HoneyGen: Generating Honeywords Using Representation Learning

Antreas Dionysiou
University of Cyprus
Nicosia, Cyprus
adiony01@cs.ucy.ac.cy

Vassilis Vassiliades
CYENS Centre of Excellence
Nicosia, Cyprus
v.vassiliades@cyens.org.cy

Elias Athanasopoulos
University of Cyprus
Nicosia, Cyprus
eliasathan@cs.ucy.ac.cy

ABSTRACT

Honeywords are false passwords injected in a database for detecting password leakage. Generating honeywords is a challenging problem due to the various assumptions about the adversary's knowledge as well as users' password-selection behaviour. The success of a Honeywords Generation Technique (HGT) lies on the resulting honeywords; the method fails if an adversary can easily distinguish the real password. In this paper, we propose HoneyGen, a practical and highly robust HGT that produces realistic looking honeywords. We do this by leveraging representation learning techniques to learn useful and explanatory representations from a massive collection of unstructured data, i.e., each operator's password database. We perform both a quantitative and qualitative evaluation of our framework using the state-of-the-art metrics. Our results suggest that HoneyGen generates high-quality honeywords that cause sophisticated attackers to achieve low distinguishing success rates.

CCS CONCEPTS

• **Security and privacy** → *Database activity monitoring*; *Access control.*

KEYWORDS

honeyword, authentication, password, natural language processing

ACM Reference Format:
Antreas Dionysiou, Vassilis Vassiliades, and Elias Athanasopoulos. 2021. HoneyGen: Generating Honeywords Using Representation Learning. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3453092

1 INTRODUCTION

Existing password-based authentication systems maintain a sensitive file comprised of the registered users' hashed passwords [32]. This sensitive file is an attractive target for attackers as if successfully retrieved and cracked, i.e., recover the hashed passwords' plain-text formats, an adversary can undetectably impersonate a user. Several prestigious web services have been compromised, e.g., Yahoo [18], Dropbox [19], LinkedIn [23], and millions of passwords were leaked [32]. For example, the "*Have i been pwned?*" [1] database contains over 500 million real-world plain-text passwords previously exposed in data breaches. This has been made possible through sophisticated password guessing techniques [12, 27, 30, 34] and the emergence of hardware such as GPUs [13]. It is worth noting that these data breaches are often detected after several months or years, when the attackers had well exploited the data and then posted (or sold) it online [32]. For this reason, Wang et al. [32] highlight the need for active, timely password-breach detection methods to enable responsive counter-actions.

Various methods, such as machine-dependent functions [1], distributed cryptography [7], and external password-hardening services [25], make offline password guessing harder. However, all these approaches have significant limitations, e.g., they have poor scalability or they require significant changes to the server-side and client-side authentication systems, which prevent the community from applying them [32]. A promising approach, proposed by Juels and Rivest [22], is to utilise *honeywords*: false passwords associated with each user's account for detecting password leakage. Even if an attacker steals and reverts the password file F, containing the users' hashed passwords, they must still decide about the real password from a set of k distinct sweetwords [2]. Using a honeyword to log in sets off an alarm, as an adversarial attack has been reliably detected [22]. Honeywords are only useful if it is hard to differentiate them from real world passwords, otherwise a sophisticated attacker may manage to distinguish them from real world ones and subvert their security. Thus, the *honeywords generation process* is of utmost concern when incorporating this security feature to existing authentication mechanisms.

Juels and Rivest [22] mention that the *honeywords generation problem* is an interesting research direction due to the various assumptions about the knowledge of the adversary as well as the password-selection behaviour of users. In addition, any proposed

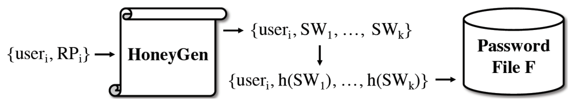

{user$_i$, RP$_i$}
HoneyGen
{user$_i$, SW$_1$, …, SW$_k$}
{user$_i$, h(SW$_1$), …, h(SW$_k$)}
Password File F

Figure 1: HoneyGen receives i_{th} user's Real Password (RP_i) and responds with an enriched with ($k-1$) honeywords passwords list containing k sweetwords (SWs) in total. Then, the returned SWs are hashed (according to each operator's hash function $h()$) and stored in the password file F.

[1]https://haveibeenpwned.com/Passwords
[2]Each user's real password and their $k-1$ honeywords are called *sweetwords*.

HGT should ensure its *non-reversibility* property, i.e., it has to be computationally inefficient (or impossible) to go from the enriched with honeywords password file to the initial password file containing only the real password for each user. In this paper, we propose HoneyGen, a practical and non-reversible honeywords generation framework (see Figure 1). To our knowledge, little to none attention has been given on the design of efficient and generic honeyword generation methods. Juels and Rivest [22] proposed four Honeyword Generation Techniques (HGTs) which have been later shown to be ineffective to meet the expected security requirements [32]. In this paper, we leverage Machine Learning (ML) to automatically generate honeywords that are indistinguishable from real passwords. Our methodology, being policy agnostic, can be applied to any potential password-based authentication system with minimal effort. Furthermore, our ML-based honeywords generation process supports the generation of honeywords with arbitrary length and structure. Our approach leverages representation learning techniques [2] to learn useful and explanatory representations from a massive collection of unstructured data, i.e., leaked password datasets, and thus generate as realistic as possible honeywords. We on purpose leverage ML for generating honeywords, since the intrinsic stochasticity of the utilised models ensure that reversing the algorithm is computationally hard (see Section 5). Finally, HoneyGen can be applied to other similar problems, such as generating decoy passwords [24] which are similar to honeywords.

So how do we tackle the honeywords generation problem? One approach would be to perform data analysis on published or leaked datasets that contain real passwords and design different *chaffing-by-tweaking* [3] techniques with respect to those datasets' password distribution. However, this approach would produce honeywords that only match the leaked datasets' password policies without being generic enough.

Another approach would be to generate honeywords using a probabilistic model that returns as honeywords the top-k nearest neighbours of a particular password from each operator's password corpus. The produced honeywords using this approach would have the following benefits: (a) they will match each operator's password policies, (b) they will accurately model the password selection behaviour of each website's users and (c) they will be realistic looking as they are essentially the actual passwords of other users. However, this technique has a limited honeywords generation spectrum as no new honeywords can be generated apart from those that already exist in each operator's password corpus. In addition, this HGT does *not* ensure the non-reversibility property as a potential (sophisticated and with large resources) attacker might reproduce the operator's model used for generating the honeywords by training k^n (k: no. of sweetwords per user, n: no. of registered users) different word embeddings models, one for each possible password file F created by considering only one sweetword per user account.

In this paper, our key insight is that it is possible to have the best of both approaches by using a *hybrid* HGT that combines the benefits of the two aforementioned generation methods, while also guaranteeing the non-reversibility property. More specifically, our approach is split into two phases. At a first phase, we train an ML model on the password corpus, which allows to learn the structure of the input and produce a *word embedding* for each password. By doing so, we are able to query the word embeddings ML model for the top-k nearest neighbours of a given password. Then, at a second phase we issue a chaffing-by-tweaking technique for perturbing (in a minimal way) the returned passwords in order to generate out-of-vocabulary (OOV) honeywords.

We employ a two-fold methodology for evaluating HoneyGen's performance on generating indistinguishable to the real passwords honeywords. In particular, we perform both a *quantitative* and *qualitative* evaluation by using the state-of-the-art metrics (proposed in [32] and [22]) and conducting a user study with human participants. As a result, we can safely conclude about the actual resistance of our framework against sophisticated distinguishing attackers.

Our contributions can be summarized as follows.

(1) We introduce HoneyGen, a *practical* and *highly performing* framework for generating indistinguishable to the real passwords honeywords. HoneyGen *outperforms* the state-of-the-art HGTs in terms of minimizing the adversary's success rate on distinguishing the real password from the honeywords and *meets* the expected security requirements, i.e., ϵ-flatness [22], flatness graph [32] and success-number graph [32].
(2) We leverage representation learning techniques to generate *high-quality*, i.e., realistic looking, honeywords. In addition, we take advantage of the stochasticity of the utilised ML models to *ensure* the *non-reversibility* of our solution.
(3) We perform both a *quantitative* and *qualitative* evaluation of our framework's performance by utilising the state-of-the-art metrics proposed in [22] and [32], and by conducting a user study with human participants. Our results suggest that HoneyGen is highly effective on *timely* detecting, in terms of minimizing the attackers' successful guesses before raising an alarm, the leakage of password file F against sophisticated attackers. In particular, HoneyGen causes adversaries to achieve low distinguishing (of the real password) success rates approximating those of the random guessing attack (optimal solution).
(4) We utilise the *largest* and *most diversified* collection of real password datasets ever used for evaluating any HGT, at least to our knowledge. In particular, our passwords datasets are composed of over *813 million* plain-text passwords and involve *13 different* web services.
(5) We experimentally demonstrate that larger length passwords, not only have increased entropy (and thus improved security), but also lead to honeywords that are more resistant against sophisticated attackers. In addition, we revisit the suggestion regarding the total number of sweetwords per user that Juels and Rivest gave in [22], i.e., 20, and show that HoneyGen allows for larger numbers of sweetwords per user, i.e., $k > 20$, without sacrificing the quality of the produced honeywords.
(6) To foster further research on this topic and ease reproducibility, we release the entire code for all of our experiments[4].

[3] *Chaffing-by-tweaking* techniques perturb certain characters of the real password, according to some heuristics, to generate honeywords.

[4] https://bitbucket.org/srecgrp/honeygen-generating-honeywords-using-representation-learning

2 PRELIMINARIES

Honeywords: How it Works. The emergence of high-performance computing systems, such as GPUs or large-scale distributed clusters, and the dedicated password-cracking hardware [16] made potential adversaries capable of inverting most (if not all) of the password hashes in the password file F [22]. Thus, once an adversary has successfully obtained F, it is realistic to assume that the majority of the passwords can be offline guessed [32].

Honeywords reliably detect a password file F leakage by associating each user's account with $k-1$ honeywords, i.e., false passwords [22]. This is because, even if an attacker A has retrieved a copy of the password file F, and every other values required, e.g., salt, to compute the hash function $h()$, and has successfully recovered all the passwords, e.g., using brute-force or other password guessing techniques [30], she has to first tell apart each user's real password from a set of $k-1$ intentionally generated and realistic looking honeywords [22]. An online login attempt using a honeyword will set off an alarm, while also indicating a password file compromise at the server. However, the adversary still has $1/k$ chances of selecting a user's real password and successfully (and undetectably) impersonate them. This approach is rather practical to be applied to existing authentication systems as it requires *minimal* changes to the server-side system and *no* changes to the client-side system [32]; yet it is very effective even with a modest chance of catching an adversary, e.g., for $k = 4$ the chance to catch an adversary is $3/4 = 75\%$. Although we are not aware of any web site that uses honeywords in practice, there is active research in the field, e.g., [32].

The Honeywords Generation Problem. Generating honeywords is a *challenging problem* mainly due to the various assumptions about the knowledge of the adversary as well as the password-selection behaviour of human users [7, 9, 14, 17, 22]. The success of HGTs lies on the quality of the resulting honeywords; the method fails if an adversary can easily distinguish the real password from the honeywords [22]. Juels and Rivest proposed four legacy user-interface (UI) HGTs which are heavily based on random replacement of letters, digits and symbols, and thus inherently unable to resist semantic-aware attackers [32].

Juels and Rivest proposed an evaluation metric for measuring the security of HGTs, namely ϵ-flat, which essentially measures the maximum success rate ϵ that a potential adversary A can gain by submitting *only one* online guess, when given each user's k sweetwords [22]. A perfectly flat, i.e., $1/k$-flat, HGT means that an adversary who has compromised the password file F and successfully recovered all k sweetwords of each user, has at most $1/k$ chances of selecting the correct password from each user's list of k sweetwords. However, the authors note that while a $1/k$-flat HGT is ideal, a HGT may be *effective* even if *not* perfectly flat.

Later, Wang et al. [32] proposed two evaluation metrics, namely *flatness* and *success-number graphs*, for addressing the limitations of the ϵ-flat evaluation metric (see Section 4). Among the rest, the authors reveal that generating equally probable to the real password honeywords, using random replacement techniques, is *inherently impossible* [32]. Thus, they evaluate Juels and Rivest HGTs and show that they all *fail* to achieve the expected security requirements. In addition, Wang et al. [32] demonstrate that probabilistic password guessing models *cannot* be adapted to generate high quality honeywords (which was a common belief in [22]). Thus, the community should focus on developing new HGTs that will effectively meet the expected security requirements.

Users' Password Selection Behaviour. Human selected passwords, even if long and sprinkled with extra characters, must still be easy to remember [29]. Thus, human users will always tend to include a memorable substring in their passwords despite the strict (in some cases) composition rules, that require passwords to be drawn from specific regular languages and include digits and non-alphanumeric characters [29]. For example, Weir et al. [34] observed that 50% of the passwords that contain a single digit, that digit is the number 1. Thus, a potential adversary could utilise this observation to exclude the honeyword selections that contain a different digit. However, such probability estimation techniques might *not* reflect the actual password distribution of each web service and thus, *not* achieve high-enough attack success rates on a set of different authentication systems employing dramatically different password policies.

A more intuitive approach is to bias the HGT towards the most likely to be chosen by users password space [22]. This, will decrease the adversary's ability of distinguishing the real password from the honeywords based on their unlikeness to be chosen by human users. Thus, understanding the way users select their passwords will help the community to identify and support specific assumptions and policies applied as well as generate high quality and plausible honeywords [4]. In particular, for a HGT to be effective, it should somehow incorporate the users password selection behaviour while also being generic enough (or easily tunable) to meet each operator's password policies.

Word Representation Learning. A popular idea in modern ML is to represent words by vectors which capture hidden information about a language, like word analogies or semantics. Representation learning techniques are even used to train deep architectures for CAPTCHA solving [11] or protein secondary structure prediction [10]. *Word embedding techniques* aim to map words or phrases from the vocabulary to vectors of real numbers, the `Word2Vec` [28] being the most popular one. `Word2Vec` [28] is a group of ML models used to produce *word embeddings* (typically of several hundred dimensions) given a large corpus of text with each distinct word in the corpus being assigned a corresponding vector in the latent space. Words that share common contexts and thus similar characteristics/semantics are located close to each other in the latent space [28].

`Word2Vec` [28], `FastText` [21] and other popular word embedding techniques, utilise Continuous Bag of Words (CBOW) or Skip-gram models, based on Artificial Neural Networks (ANNs), to learn the underlying word representations. In the CBOW model [28] the ANN predicts the current word given a window of surrounding context words. Contrary, the Skip-gram model [28] uses the current word to predict the surrounding window of context words by weighting nearby context words more heavily than more distant ones [28]. Generally speaking, CBOW is faster while Skip-gram has better performance (especially for rare words) [28].

`FastText` represents each word as a bag of character n-grams. A vector representation is associated to each character n-gram; words being represented as the sum of these representations [3]. Since

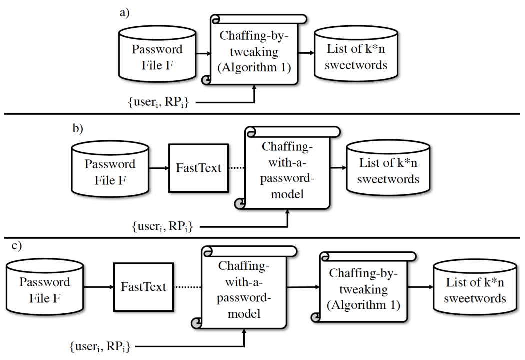


Figure 2: *HoneyGen*'s generation pipelines. Sub-figures (a), (b) and (c) represent the *chaffing-by-tweaking*, *chaffing-with-a-password-model* and *chaffing-with-a-hybrid-model* HGTs, respectively. Chaffing-with-a-hybrid-model HGT combines (a) and (b) by producing $k/2$ honeywords using (b) and then generating another $k/2$ honeywords by issuing chaffing-by-tweaking on the returned, by chaffing-with-a-password-model HGT, honeywords. For each newly registered user pair $\{user_i, RP_i\}$ (or batch of user pairs), (b) and (c) HGTs can update (retrain) their word embeddings model, i.e., $FastText$, before generating k sweetwords for $user_i$.

`FastText` exploits subword information, it can also compute valid representations for OOV words by taking the sum of its *n*-gram vectors. In our case, `FastText` is the only useful word embedding technique as the majority of users' selected passwords are OOV words due to either password polices or users' choice. As a result, `FastText` will still be able to compute a word embedding for each given password by summing up the matched *n*-gram vectors. Note that website operators store users' passwords in a hashed format for increased security [22]. However, `FastText` allows for the continuation of the model's training every time a new password (or a batch of passwords) is entered. Thus, HoneyGen will actively train the word embeddings model on each given password before storing its hashed format and deleting its plain-text one. In our experiments, we use `FastText` with `minCount=1`, `minn=2`, and `model=skipgram`.

3 HONEYGEN

The common intuition is that the users select passwords that are easy to remember [4, 35] often reusing them across multiple sites [24]. Thus, we exclusively design HoneyGen's generation pipeline to incorporate the human-memorable factor of users' password-selection behaviour. In doing so, we leverage representation learning techniques to learn useful and explanatory representations from a massive collection of unstructured data, i.e., each operator's password database, and thus generate as realistic as possible honeywords that approximate the password distribution of each authentication system. We propose (and later evaluate) three HGTs: (a) a *chaffing-by-tweaking* technique, (b) a *chaffing-with-a-password-model* technique and (c) a *hybrid* method that combines the two aforementioned techniques, for generating indistinguishable, to the real passwords, honeywords.

Figure 2 shows HoneyGen's generation pipeline for each of the aforementioned honeyword generation approaches. HoneyGen is explicitly designed to be applicable on any password-based authentication system using any kind of password policies. Thus, each operator can utilise HoneyGen for generating system-specific honeywords, that satisfy the deployed password policies, without the risk of subverting the system's security. Although our HGTs are explicitly designed to minimize the risk against *targeted-guessing* attackers [32], who can exploit the victim's Personally Identifiable Information (PII) such as their birthday or phone number, in this paper, we focus *only* on *trawling-guessing* [5] attackers for evaluating HoneyGen's performance, as all of our datasets are comprised of passwords only and not any other relevant to the users' PII.

Chaffing-by-Tweaking. Numerous heuristic-based *chaffing-by-tweaking* HGTs have been proposed in [22]. When chaffing the passwords one must be careful not to subvert the security of the generated honeywords so the attacker cannot tell the password from its tweaked versions. For example, if a chaffing-by-tweaking technique randomly perturbs the last three characters of a password, then, in the example "`57*flavors`", "`57*flavrbn`", "`57*flavctz`" the adversary can easily tell that the real password is the first one [22]. In other words, any proposed chaffing-by-tweaking technique must take into explicit consideration potential *semantic-aware* attackers in order to be useful [8, 14]. As shown in Figure 2a, HoneyGen's chaffing-by-tweaking operation is straightforward. First, it receives as input each operator's password file F containing n records (one for each user) with real passwords and later responds with a list of size $k \times n$, where each password in F is associated with $k-1$ honeywords.

After performing some data analysis on the 13 datasets of leaked passwords (shown in Table 3) we observe the following: on average, (a) *89.85%* of the symbols included in a given password are the *same* and (b) *94.77%* and *5.23%* of the letters contained in a given password are lower-case and upper-case, respectively. Thus, in our chaffing-by-tweaking strategy we choose to replace all the occurrences of a particular symbol in a given password with a randomly selected and different one. In addition, we choose to lower-case each letter of a password with a higher probability than upper-casing it. Furthermore, we choose to replace each digit in a given password with a small probability in order to mitigate the risk against *targeted-guessing* attackers [32] who can exploit the victim's PII. Note that HoneyGen adaptively increases those probabilities in case the algorithm produces the same honeywords for a number of (predefined) times. As a result, HoneyGen can be even applied on authentication systems that contain passwords with low entropy.

We choose to chaff the generated passwords in a minimal way in order to avoid any obvious discrimination between the real passwords and the honeywords. Our chaffing algorithm, shown in Algorithm 1 (Appendix A), perturbs a honeyword in 3 different levels. First, the algorithm lower-cases and upper-cases each letter with respect to a probability p and f, respectively. Second, it replaces each digit occurrence with respect to a probability q. Third, for each symbol found in the given password, it replaces all of its occurrences with a randomly selected and different one with respect to a probability p. Selecting the initial values for the

[5] *Trawling-guessing* attackers *do not* have access to the victim's PII.

Table 1: Honeyword samples generated using our chaffing-by-tweaking HGT (see Algorithm 1 - Appendix A).

Passwords:	littleDog	timmy3531	cri;cri;
Chaffed Honeywords:	LittledoG	timmY3531	Cri@cri@
	littledog	Timmy3531	cri!cri!
	littleDoG	TImmy6531	cri#CRi#
	LIttledog	timmy3731	cri@cri@

probabilities f, p and q is of major importance as they largely affect the produced honeywords' deviation from the real password. According to the analysis results mentioned in the previous paragraph and based on experimentation with different initial values we set $p = 0.3$, $f = 0.03$ and $q = 0.05$.

Our chaffing-by-tweaking algorithm produces honeywords by limiting the perturbations made to the real password and taking into explicit consideration the user's password selection behaviour in order to generate realistic honeywords. In particular, we choose to lower-case or upper-case each letter occurrence instead of replacing it with a different letter in order to avoid any potential security degradation of the generated honeywords such as the one introduced by the chaffing-by-tail-tweaking method proposed by Juels and Rivest [22]. In addition, we choose to replace each digit occurrence with a small probability to mitigate the risk against targeted attackers that exploit the victim's PII. Finally, we replace each occurrence of each particular symbol in a given password with a randomly selected and different one with a larger probability as symbols usually do not exhibit any significant semantic information to the attacker. Some honeyword samples, generated using our chaffing-by-tweaking HGT, are shown in Table 1.

Chaffing-With-a-Password-Model. Chaffing-with-a-password-model techniques generate honeywords using probabilistic models based on high volume lists of real passwords [22]. This HGTs produce honeywords that are more realistic looking compared to chaffing-by-tweaking HGTs as they approximate the users' password selection behaviour in a more sensible way [32].

As shown in Figure 2b, the first step in HoneyGen's chaffing-with-a-password-model technique requires a password corpus. This password corpus acts as the training dataset for our word embeddings technique, i.e., `FastText`. After successfully training our word embeddings model [6] we can query it by giving a real password as input and receiving as a response an x-dimensional (in our case 100-dimensional) vector representing the *word embedding* of the given password. Next, for constructing the list of $k \times n$ sweetwords, we iterate over each password included in our passwords' corpus (n records in total) and return its top-k nearest neighbours in decreasing order of *cosine similarity*. As a result, we create a list containing the k most similar passwords, i.e., *honeywords*, for each password included in the password file F. Note that the generated passwords are different for different password databases.

In order to develop a *proof-of-concept* for our chaffing-with-a-password-model HGT we utilise *RockYou* dataset [6] containing 14, 341, 497 leaked passwords as the training dataset. However, generating honeywords solely based on published or leaked password databases is *not* a good idea as such lists may also be available to potential adversaries, who might use them to identify the honeywords in the password file F [22]. Thus, we underline that each operator should train our chaffing-with-a-password-model on their password database, not only for mitigating the aforementioned risk, but also for the produced honeywords to comply with the deployed password policies. Some honeyword samples, generated using our chaffing-with-a-password-model HGT, are shown in Table 2.

[6]Note that a detailed analysis regarding the influence of the hyper-parameters' values to the final model is out of the scope of this paper.

Table 2: Honeyword samples generated using our chaffing-with-a-password-model HGT. The word embeddings ML model has been trained on the *RockYou* dataset [6].

Passwords:	loverose1	lavonda1	marino40
Chaffed Honeywords:	loverose21	lafonda1	marino54
	loverme23	lagonda1	marina40
	loveernie1	lavonta1	marino25
	loverose01	lavenda1	marinos3

Chaffing-With-a-Hybrid-Model. Chaffing-with-a-hybrid-model techniques combine the benefits of multiple honeyword generation strategies [22]. Despite the efficiency as well as the sophistication of chaffing-by-tweaking techniques, they *cannot* generalize well on dramatically different password corpus from the ones used for forming those functions. A potential attacker is capable of subverting the security of honeywords that only deploy chaffing-by-tweaking techniques by calculating cumulative statistics about the password file F. On the other hand, the probabilistic chaffing-with-a-password-model technique suffers from the limitation that it cannot produce honeywords that are not included in the initial password corpus. Although this is *not* a direct implication for the security of the produced honeywords it dramatically limits the available honeywords generation spectrum and thus, the computational time needed for recovering the plain-text versions of the hashed passwords included in the password file F. As a result, in order to avoid the individual defects of our chaffing-by-tweaking and chaffing-with-a-password-model HGTs, we choose to combine them and create a *hybrid* model that produces high-quality, indistinguishable to the real passwords, honeywords. As a result, we will further limit the adversary's success rates on distinguishing the real passwords from the honeywords.

Figure 2c shows the honeyword generation pipeline for our hybrid method. First, we train the word embeddings model, i.e., `FastText`, on the password corpus. Second, we utilise the trained word embeddings model to produce l ($l < k$) honeywords for each real password included in the password corpus. Third, we issue our chaffing-by-tweaking technique to perturb the l honeywords retrieved from our word embeddings model in order to finally produce k sweetwords in total for each user pair $\{user_i, RP_i\}$.

HoneyGen's Applicability. HoneyGen can be applied to any modern authentication system with *minimal* effort. In particular, for already running password-based authentication systems the operators can invite users to re-enter their passwords in order to collect the plain-text versions of the already stored hashed passwords issuing, afterwards, the HoneyGen's honeywords generation

Table 3: The real password datasets used for HoneyGen's performance evaluation in chronological order of data breach year (derived from temp.hashes.org).

Dataset	*Total Entries*	*Data Breach Year*
phpbb	183440	2009
rockyou	14341497	2009
linkedin	60649463	2012
dropbox	10278958	2012
yahoo	5953057	2013
myspace	51283076	2013
last.fm (2016)	20510674	2016
adultfriendfinder	36892926	2016
youku	47633248	2016
chegg	20103871	2018
dubsmash	22106265	2018
have-i-been-pwned-v2	501444649	2018
zynga	42343670	2019

process. In addition, in order to minimize the computational overhead for retraining the word embeddings model, i.e., `FastText`, and computing the top-k nearest neighbours for each password, we advice systems' operators to retrain their word embeddings model every $1,000$ newly registered users, i.e., $1,000$ newly created passwords.

4 HONEYGEN EVALUATION

In this section, we evaluate HoneyGen's performance on creating indistinguishable to the real password honeywords. To do so, we deploy the evaluation metrics proposed in [22] and [32], and conduct a user study asking human participants to distinguish the real password from a list of k sweetwords. We choose to use two different methodologies for evaluating HoneyGen's performance, i.e., quantitative and qualitative, in order to avoid any bias regarding the datasets used as well as the adversary's knowledge and capabilities when trying to distinguish the real passwords from the honeywords. Note that we run all of our attacks on a 4-core Xeon machine with 64GB of memory.

Datasets. We evaluate HoneyGen's performance based on 13 datasets containing real-world passwords (see Table 3). In total, our passwords datasets are composed of over *813 million* plain-text passwords and involve *13 different* web services. To our knowledge, this is the *largest* and *most diversified* collection of real passwords ever used for evaluating any HGT. The majority of the utilised datasets may disclose the recent password selection behaviour of users as some of them have recently leaked by hackers or insiders. Note that we process those datasets and select only the passwords with length ≥ 8 as the majority of the *top-50* popular websites ranked by *alexa.com* as of November 2020, including Google, Microsoft, Facebook, and many others require the use of at least 8 characters for any selected password. In addition, we randomly select $50,000$ real passwords from each leaked dataset volume in order to facilitate the evaluation process of our HGT, without loss of generality. Table 3 shows the password datasets used for evaluating HoneyGen's performance in chronological order of data breach year.

Quantitative Evaluation. The ϵ-flat evaluation metric proposed by Juels and Rivest [22] measures the maximum success rate ϵ that an attacker can gain by submitting only *one* online guess to a system. However, this method falls short in two cases: (a) when the attacker is allowed to make more than one guess per user and (b) on identifying a system's most vulnerable honeywords, i.e., those that can be easily distinguished [32]. Thus, Wang et al. [32] proposed two new metrics, namely *flatness graph* and *success-number graph*, that measure a HGT's resistance against honeyword distinguishing attackers, for tackling the two aforementioned problems.

Flatness graph plots the probability of distinguishing the real password versus the number of allowed sweetword login attempts per user x ($x \leq k$) [32]. The flatness graph showcases the *average resistance* of a distinguishing attacker for each allowed number of guesses per user. A *perfect* HGT allows for a maximum of $x \times 1/k$ success rate for each allowed sweetword login attempts per user x.

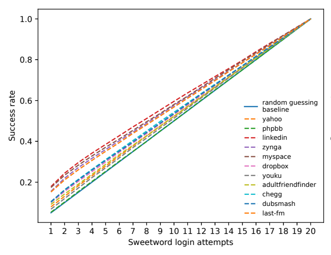
Success-number graph plots the total number of successful login attempts, i.e., login with a real password, versus the total number of failed login attempts, i.e., login with a honeyword [32]. The success-number graph measures to what extent a method will produce vulnerable honeywords that could be easily distinguished. A *perfect* HGT produces k sweetwords per user with the same probability of being the real password. For the success-number graphs we only consider the *worst case* scenario where the attacker can perform k online guesses, i.e., use all the given sweetwords until the real password is found.

Wang et al. conduct a honeyword distinguishing attack, namely *Normalized Top-PW*, in order to gather the appropriate statistics for plotting the two aforementioned graphs. In particular, given an adversary A and a password file F of $n \times k$ sweetwords (where n is the total number of users and k the number of sweetwords per user, e.g., 20 as suggested in [22]), A tries to find as many as possible real passwords before making T_2 failed honeyword login attempts in total. Note that A can only make at most T_1 login attempts per user. A tries each of these k sweetwords per user in decreasing order of *normalized* probability, where the probability of each sweetword $sw_{i,j}$ ($1 \leq i \leq n$ and $1 \leq j \leq k$) comes directly from a known probability distribution of a leaked password dataset D like the *RockYou* [6] or LinkedIn [23] and is calculated as follows. For each sweetword that exists in D then $Pr(sw_{i,j}) = P_D(sw_{i,j})$ else $Pr(sw_{i,j}) = 0$. $\forall x \in D$, $P_D(x) = Count(x)/|D|$, where $Count(x)$ is the number of occurrences of x in D and $|D|$ is the size of the leaked passwords dataset D. Note that the normalized Top-PW adversary first attacks the *vulnerable* user accounts, i.e., the user accounts for which their most probable honeyword is closest to 1. For doing so, we normalize each user's k sweetwords as follows. $\forall sw_{i,j} \in n \times k$ sweetwords, $Pr(sw_{i,j}) = Pr(sw_{i,j})/\sum_{t=1}^{k} Pr(sw_{i,t})$. If the system allows more than one honeyword login attempt, i.e., $T_1 > 1$, after a sweetword has been attempted, the probability of all the other unattempted sweetwords should be re-normalized. Our naming conventions are inspired by the formalism of Wang et al. [32].

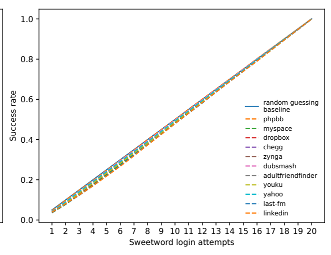
For evaluating the 4 HGTs proposed by Juels and Rivest [22], Wang et al. [32] employ a *half-half* strategy by using the first half as the *target* dataset, i.e., to generate honeywords for each real password and later try to distinguish the real password from the k sweetwords, and the second half as the *attacker* dataset, i.e., the leaked dataset D to be used for launching their attack described in the previous paragraph. In contrast to Wang et al. [32], for evaluating HoneyGen's performance, we *do not* deploy this *half-half* strategy and we instead use two *different* datasets one for the target and one for the attacker. By doing so, we are able to explore the *worst case* scenario where the target and attacker datasets are drawn from a *different distribution.* This scenario is the most realistic one as *different* websites/systems usually make use of *different* password policies thus having *different password distributions* [30, 32, 33]. Note that we *do not* consider targeted-guessing attackers [32] who can exploit the victim's personal information, for evaluating HoneyGen's performance, as all of our datasets are comprised of passwords only and not any other relevant to the user PII.

Results. Figure 3 shows the flatness (3(a)-3(c)) and success-number graphs (3(d)-3(f)) when attacking each HGT proposed as part of HoneyGen framework, having the *RockYou* as the target dataset (for which the attacker has to distinguish the real password from the list of $n \times k$ sweetwords) and each of the remaining datasets as the attacker dataset D (used to compute the probability of each sweetword as described in the normalized Top-PW attack). Sub-figures 3(a)-3(c) reveal that our chaffing-by-tweaking, chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs are 0.17^{+}-flat, 0.04^{+}-flat and 0.08^{+}-flat, respectively, the optimal HGT being $1/k = 0.05$-flat. Those results indicate that our chaffing-by-tweaking HGT is $\approx 3\times$ weaker than expected in [22], whereas the two other two HGTs are very close to achieving the optimal flatness. Sub-figures 3(d)-3(f) show that the normalized Top-PW attacking strategy can distinguish 120, 1 and 4 real passwords (the optimal HGT allowing for a total of $\lfloor T_2/(k-1) \rfloor = \lfloor 61/19 \rfloor = 3$ successful password guesses) against our chaffing-by-tweaking, chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs, respectively, when allowed $T_2 = 61$ honeyword logins, which is the equivalent threshold of the total number of allowed login attempts using a honeyword that Wang et al. used in [32] with respect to each target dataset's size. However, we set $T_1 = 20$ to explore the worst-case scenario in contrast to Wang et al.'s evaluation where they only allow 1 login attempt per user account, i.e., $T_1 = 1$. We conduct the same attack, i.e., normalized Top-PW, using each of the 13 datasets (shown in Table 3) as the the target dataset and the remaining ones as the attacker dataset D. Nonetheless, due to space constrains and because we observe similar results we omit including each one of them. However, these results are taken into consideration in a later graph, i.e., Figure 4, which shows the *average* flatness and success-number graphs for each HGT.

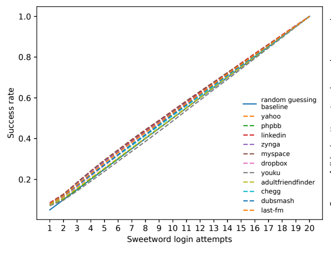
Table 4 shows the average attack success rates achieved on each HGT, using each of the 13 datasets shown in Table 3 as the target dataset and each of the remaining ones as the attacker dataset D. When allowing 61 failed attempts, i.e., logins using a honeyword, $0.1196\% \sim 0.9953\%$ accounts of the 13 target datasets can be successfully guessed. Observing the results reported in Table 4, one can easily see that the *chaffing-with-a-password-model* HGT performs better than the other HGTs as it causes the attacker to achieve the lowest attack success rates on average. The 2nd best performing HGT (by a small margin, i.e., 0.0669%, from the highest performing HGT) is the chaffing-with-a-hybrid-model HGT. Finally, the worst performing, i.e., least secured, HGT is the chaffing-by-tweaking.



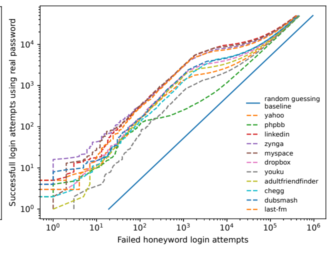
(a) Chaffing-by-tweaking



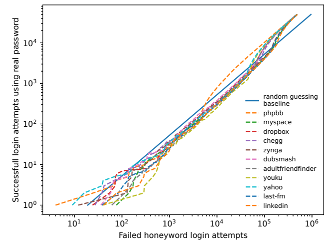
(b) Chaffing-with-a-password-model



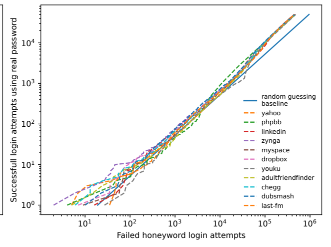
(c) Chaffing-with-a-hybrid-model



(d) Chaffing-by-tweaking



(e) Chaffing-with-a-password-model



(f) Chaffing-with-a-hybrid-model

Figure 3: The *flatness (3(a)-3(c))* and *success-number (3(d)-3(f))* graphs using *RockYou* as the target and each of the other datasets as the attacker dataset D, for each HGT. The closer to the random guessing baseline, the better the corresponding HGT is.

Table 5 shows the ϵ-flatness results for each HGT, using each of the 13 datasets shown in Table 3 as the target dataset. Again, the *chaffing-with-a-password-model* HGT performs consistently well achieving the lowest ϵ-flatness on average, i.e., 0.0866, compared to the other two HGTs. In particular, chaffing-with-a-password-model HGT almost achieves the optimal ϵ-flatness, which is $1/k = 1/20 = 0.05$. However, even the ϵ-flatness results achieved from chaffing-with-a-hybrid-model and chaffing-by-tweaking HGTs are not bad at all being $2\times$ and $3\times$ weaker, respectively, than the optimal solution. This is because a HGT may be effective even if not perfectly flat [22]. Finally, the worst performing, i.e., least secured, HGT is again the chaffing-by-tweaking.

The average flatness and success-number graphs for each HGT proposed in this paper are shown in Figure 4. As shown in sub-figure 4(a), on average, the chaffing-by-tweaking HGT is 0.18-flat and $\approx 3\times$ weaker than the optimal ϵ-flatness, i.e., 0.05. The chaffing-with-a-hybrid-model HGT is 0.12-flat and $\approx 2\times$ weaker than the optimal ϵ-flatness. Finally, the chaffing-with-a-password-model

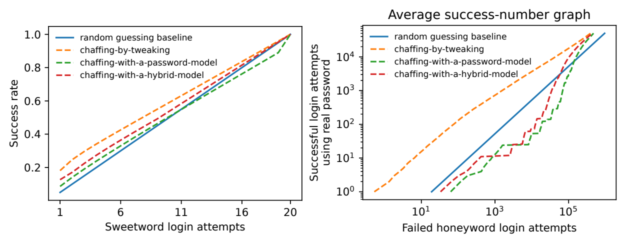
Table 4: The average attack success rates (%) achieved on our chaffing-by-tweaking (tweaking), chaffing-with-a-password-model (model) and hybrid HGTs for each target dataset ($T_1 = 20$, $T_2 = 61$, $k = 20$). A value in bold means that the corresponding HGT performs best, while a value in *italics* means that it performs the worst.

Target Dataset	*Tweaking*	*Model*	*Hybrid*
phpbb	*0.8639%*	**0.0239%**	0.0279%
rockyou	*0.2399%*	**0.0019%**	0.0079%
linkedin	*0.1859%*	**0.0039%**	0.0079%
dropbox	*0.6439%*	**0.1499%**	0.0899%
yahoo	*8.1398%*	**0.4519%**	1.4139%
myspace	*0.5839%*	**0.0719%**	0.0459%
last.fm (2016)	*0.2619%*	**0.0039%**	0.0099%
adultfriendfinder	*0.3779%*	**0.0039%**	0.0079%
youku	*0.0759%*	**0%**	**0%**
chegg	*0.0899%*	**0.0019%**	**0.0019%**
dubsmash	*0.8899%*	**0.6679%**	0.6779%
have-i-been-pwned-v2	*0.2019%*	**0%**	0.0019%
zynga	*0.3839%*	0.1739%	**0.1319%**
Average	*0.9953%*	**0.1196%**	0.1865%

Table 5: ϵ-flatness results for our chaffing-by-tweaking (tweaking), chaffing-with-a-password-model (model) and hybrid HGTs for each target dataset ($T_1 = 1$, $k = 20$). A value in bold means that the corresponding HGT performs best, while a value in *italics* means that it performs the worst.

Target Dataset	*Tweaking*	*Model*	*Hybrid*
phpbb	*0.2803*	**0.1222**	0.1792
rockyou	*0.1141*	**0.0402**	0.0777
linkedin	*0.0855*	**0.0323**	0.0559
dropbox	*0.2832*	**0.1443**	0.2052
yahoo	*0.1790*	**0.0599**	0.0986
myspace	*0.2285*	**0.1163**	0.1702
last.fm (2016)	*0.1175*	**0.0383**	0.0661
adultfriendfinder	*0.1349*	**0.0432**	0.0786
youku	*0.0545*	**0.0201**	0.0345
chegg	*0.0831*	**0.0284**	0.0542
dubsmash	*0.4359*	**0.2975**	0.3572
have-i-been-pwned-v2	*0.0700*	**0.0230**	0.0374
zynga	*0.2976*	**0.1611**	0.2371
Average	*0.1589*	**0.0866**	0.1270

HGT is 0.08-flat with a minor difference, i.e., 0.03^+, from the optimal ϵ-flatness. The ϵ-flat results are by default with $T_1 = 1$. However, the flatness graph/metric proposed by Wang et al. [32] explores the attack success rate when the adversary is allowed to make more than 1 guesses. In particular, the flatness graph provides a view of the average resistance against a distinguishing attacker with varied guess numbers per user all the way to k [32]. Thus, the closer to the random guessing baseline, the better the corresponding HGT is. As shown, the chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs perform comparably well on approximating the random guessing baseline.



(a) Flatness graph (b) Success-number graph

Figure 4: The *average flatness* and *success-number* graphs for each HGT. The closer to the random guessing baseline, the better the corresponding HGT is.

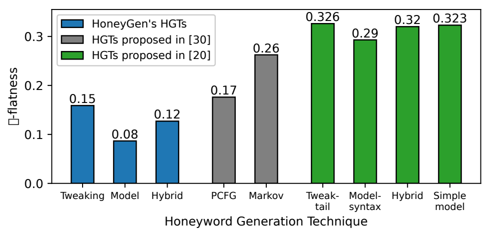
Sub-figure 4(b) shows the average success-number graphs for our three HGTs. The experimental results show that the chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs cause the normalized Top-PW attacker to achieve lower attack success rates than the random guessing baseline HGT. This is a clear evidence regarding the resistance of these two HGTs against sophisticated distinguishing attackers.

All the results reported in this section showcase that the chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs generate *high-quality* honeywords which are indistinguishable from the real passwords. We came to this conclusion by utilising the three state-of-the-art evaluation metrics, namely ϵ-flatness [22], flatness graph [32] and success-number graph [32], for HGTs. The soundness of the reported results is further strengthened by the fact that we utilise a large pool, i.e., 813^+ million, of real passwords and by involving 13 different web services which inherently have different password distributions. Finally, our results suggest that the normalized Top-PW attacking strategy is by *no means* optimal.

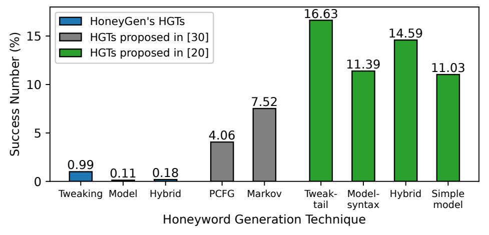
Figure 5 shows the ϵ-flatness and success-number comparison of our HGTs and those proposed in [32] and [22]. As shown, our HGTs achieve both a lower ϵ-flatness and lower success-number of successful password guesses, thus, offering increased security, compared to the HGTs proposed by Juels and Rivest [22] and Wang et al. [32]. More specifically, our framework, i.e., HoneyGen, offers HGTs that achieve $1.7^+\times$ and $2.5^+\times$ lower ϵ-flatness and $13.3^+\times$ and $30.9^+\times$ lower successful password guess rate for the equivalent total number of allowed guesses with a honeyword T_2, compared to the HGTs proposed in [32] and [22], respectively.

Implications of Normalized Top-PW attack. The normalized Top-PW attack assumes that the recovered password file F shares common features and characteristics with the leaked password dataset D used to compute the probability $Pr(sw_{i,j})$ for each sweetword. Nonetheless, in reality, password distributions differ greatly for each website/authentication system [30, 33]. Thus, the normalized Top-PW distinguishing attack, proposed by Wang et al. [32], is inherently unable to achieve high attack success rates against authentication systems that deploy dramatically different password polices from the policies used for forming each password in the leaked password dataset D.

However, in case of significant overlap between the password file F and the leaked dataset D a potential attacker can achieve



(a) ϵ-flatness



(b) Success-number

Figure 5: Comparison of the ϵ-flatness and success-number of our HGTs and these proposed in [32] and [22]. The *lower* ϵ-flatness and success-number a HGT reports the *better*.

significantly high attack success rates given that the deployed HGT does not produce honeywords in respect to the distribution of the password file F (e.g., chaffing-by-tweaking). The chaffing-with-a-password-model (and as a consequence chaffing-with-a-hybrid-model) HGT described in Section 3, takes into explicit consideration the distribution of the password file F by producing honeywords that are actual passwords of other users of a particular authentication system. This is the main reason for observing a decrease of the attack success rates on the chaffing-with-a-password-model and chaffing-with-a-hybrid-model HGTs compared to the attack success rates achieved on the chaffing-by-tweaking HGT (see Figure 4).

The results reported in this section demonstrate that the chaffing-with-a-password-model and chaffing-with-a-hybrid-model perform comparably well. However, chaffing-with-a-hybrid-model dramatically increases the honeywords generation spectrum compared to chaffing-with-a-password-model HGT, as it can generate honeywords that do not exist as real passwords in each operator's password corpus. As a consequence to increasing the honeywords generation spectrum, chaffing-with-a-hybrid-model also decreases the attacker's efficiency on recovering the plain-text format of the hashed passwords in the password file F, compared to the chaffing-with-a-password-model HGT. The larger honeywords generation spectrum of chaffing-with-a-hybrid-model increases the probabilities of this HGT to be effective even against other, more sophisticated, adversaries that may employ a completely different attacking strategy compared to the normalized Top-PW attack proposed in [32]. Thus, chaffing-with-a-hybrid-model HGT is expected to surpass the performance of chaffing-with-a-password-model HGT, in terms of producing indistinguishable to the real passwords honeywords, when tested on a variety of adversaries (especially in the case that a system has a limited number of registered users).

Due to the lack of a formal evaluation method for HGTs one should employ a two-fold evaluation strategy, i.e., quantitative and qualitative, for effectively concluding about their framework's resistance against distinguishing attackers. So far, we have presented a large-scale and detailed quantitative analysis utilising a series (13 in total) of leaked real-password datasets that exhibit up-to-date user password behaviours. Note that for the time being, this is the *largest* corpus of real-world password datasets and most *comprehensive* quantitative evaluation of any proposed HGT. In addition, our real-world password corpus is among the largest and most diversified ones ever collected for use in a password study, at least to our knowledge. In our paper, we take one step further into effectively concluding about the actual robustness of HoneyGen's performance by also conducting a *qualitative* analysis, i.e., user study, asking human participants to indicate 5 choices in decreasing order of probability of being the real password given a list of 20 sweetwords. Note that we are the first to conduct a user study for concluding about the effectiveness of our HGT.

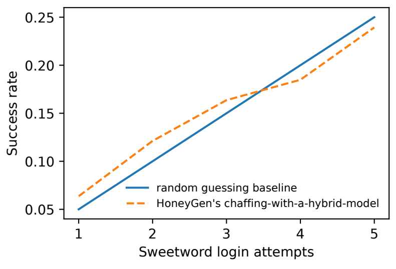
Qualitative Evaluation. In order to qualitatively evaluate our framework's performance we conduct a user study asking human participants to distinguish the real password from a list of 20 sweetwords. By doing so, we can effectively conclude about whether or not our HGT produces realistic looking honeywords that cannot be easily distinguished by human users and thus, highly intelligent adversary agents. All data have been collected in an anonymous way so we are not obliged to get an IRB approval as we do not collect any demographic information which might be sensitive.

We compose a questionnaire with 10 questions in total, where each question contains a list of 20 randomly selected sweetwords (one of them being the real password). We ask the participants to select 5 sweetwords in decreasing order of probability of being the real password, i.e., $T_1 = 5$, from a list of 20 sweetwords, along with their confidence level when answering each question. After gathering the answers from 33 randomly selected human respondents we came up with the following graphs: (a) percentage of correct guesses per allowed number of login attempts, i.e., flatness graph for $T_1 = 1$ up to $T_1 = 5$, and (b) distribution of confidence when answering each question, i.e., Likert scale questions with 5 points [5]. Note that the honeywords for each question have been produced using the chaffing-with-a-hybrid-model HGT.

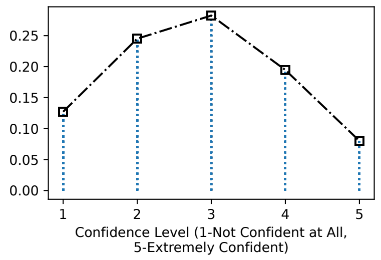
As shown in Figure 6(a) HoneyGen achieves very close to the random guessing baseline (optimal solution) flatness graph, while also being 0.06^+-flat, the optimal ϵ-flatness being 0.05. Furthermore, as shown in Figure 6(b) the confidence graph peaks at *"neither confident nor not confident"* and *"not confident"* answers. In particular, *40.6%* of the respondents felt *"not confident"* or *"not confident at all"* when distinguishing the real password from the honeywords. Observing all the results reported in this section, one can easily see that even highly intelligent adversaries, i.e., humans, *cannot* distinguish, with high success rates, the real passwords from the lists of k sweetwords. This fact demonstrates the *resistance* of our HGT against highly sophisticated attackers.

5 DISCUSSION

HoneyGen's Reversibility. In order for a honeyword generation framework to be useful it has to satisfy two goals. First, it has to

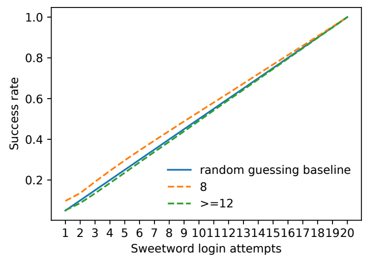


(a) Flatness graph

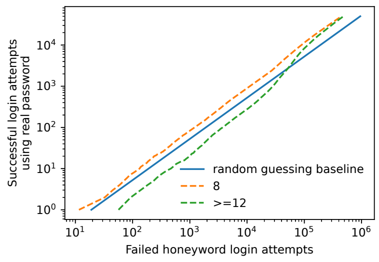


(b) Distribution of confidence

Figure 6: The flatness graph and the distribution of confidence for the user study's collected results for $T_1 = 1$ up to $T_1 = 5$, using chaffing-with-a-hybrid-model.



(a) Flatness graph



(b) Success-number graph

Figure 7: The *flatness* and *success-number* graphs, using *RockYou* as the target dataset, considering passwords with length 8 and ≥ 12.

generate high-quality honeywords that are indistinguishable to the real passwords. Second, it must be *non-reversible*, i.e., it has to be computationally inefficient (or impossible) to go from the enriched with honeywords password file to the initial password file, which only contains the real password for each user. Similar to the extensively used encryption algorithms, the HoneyGen's robustness against distinguishing attackers *does not* rely on its algorithm secrecy (which we make public). In contrast, the non-reversibility property of our HGT is guaranteed by the *stochasticity* of the ML-based honeywords generation model, i.e., `FastText`. In fact, this is the *main reason* for applying ML technologies to generate realistic looking honeywords. The continuous batch retraining of our ML-based word embeddings model every 1,000 newly registered users, dramatically changes its weights' values and thus, its previous operation mechanics.

However, HoneyGen ensures its non-reversibility property even in the case that an operator does not update its initial word embeddings model, i.e., HGT, for each batch of newly registered users. In this case, an adversary can reproduce the operator's trained word embeddings model, which meant to be private, by training k^n different word embeddings models for each possible list created by considering only one sweetword for each user account. This attack, although *computationally expensive*, e.g., for a password file F with 1 million registered users and $k = 20$ an adversary has to train $20^{1,000,000}$ different ML models for reproducing the one used by the operator for generating the honeywords, it may be possible with more sophisticated attacking strategies. However, chaffing-with-a-hybrid-model HGT eliminates this problem in an efficient and effective way. More specifically, chaffing-with-a-hybrid-model HGT offers an additional step of non-reversibility by perturbing the characters of the generated, through our ML-based model, honeywords in a random way (see Algorithm 1 - Appendix A). As a result, any potential adversary is *unable* to reproduce the word embeddings model of a particular operator, thus, *guaranteeing* the non-reversibility property of this HGT.

Long Passwords vs. Short Passwords. Long passwords will most probably have larger entropy than smaller ones. Increasing a password by 1 bit doubles the number of guesses the attacker has to perform in order to crack the password. For example, Shay et al. [31] found that some policies that require longer passwords provide better usability and security compared with traditional policies. Thus, we advice the different website operators to deploy password policies that will force users to select larger (in terms of length) passwords. This will not only decrease the adversary's performance on recovering the hashed passwords but it will also increase the security of the generated honeywords. In order to validate this assumption we conduct two experiments by attacking two different password files F_1 and F_2 that contain sweetwords of length 8 and ≥ 12, respectively, using the normalized Top-PW attacking strategy. We utilise *RockYou* as the target dataset and each of the remaining datasets as the attacker dataset.

As shown in Figure 7, F_2 is more robust compared to F_1 as it causes the attacker to achieve both lower flatness and lower success-number results. In particular, F_2 is 0.05^+-flat whereas F_1 is 0.09^+-flat. In addition, F_2 allows only 1 successful guess whereas F_1 allows 4 successful guesses, when $T_2 = 61$, on average. As a result, the length of users selected passwords not only increases their entropy but also improves the robustness of the generated honeywords.

Number of Sweetwords per User Account. Increasing the number of sweetwords per user (k) *increases* the chance that an

Table 6: Standard deviation (σ) of the achieved ϵ-flatness ($T_1 = 1$) and the success-number ($T_1 = 20, T_2 = 5,000$) from the optimal values, i.e., $1/k$ for ϵ-flatness and $\lfloor T_2/(k-1) \rfloor$ for success-number. The *smaller* the σ the *better*.

No. of sweetwords per user (k)	*ϵ-flatness σ*	*Success-number σ*
20	2.78%	18
40	0.01%	5
60	0.01%	2
80	0.06%	2
100	0.04%	6
160	0.02%	2
200	0.02%	2

adversary gets caught. For instance, if there are k sweetwords per user, an adversary has $1/k$ chances of selecting the real password. However, if there are y sweetwords per user and $y > k$, then an adversary has $1/y < 1/k$ chances of selecting the real password. For this reason, and because storage capacity is cheap, an operator might use more than 20 sweetwords per user, as suggested by Juels and Rivest [22], in order to further limit adversaries' attack success rates on distinguishing the real password from the $k-1$ honeywords.

However, in such case, where $k > 20$, a HGT may produce *unrealistic* looking honeywords that can be easily excluded from the list of possible real passwords. Thinking one step further, they may even leak significant information regarding the way that a HGT produces honeywords. This fact holds especially for HGTs that are solely based on random replacement of characters, which are either based on heuristics or not. Thus, a potential adversary may utilise sophisticated techniques to reduce the pool of possible real passwords by excluding a large amount of sweetwords that are marked as honeywords by the deployed distinguishing technique. We, for the first time, revisit the suggestion of $k = 20$ that Juels and Rivest made in [22], and offer a HGT that produces honeywords that meet the expected security requirements even when using more than k sweetwords per user.

Table 6 shows the standard deviation (σ) of ϵ-flatness and success-number from their optimal values, for each number of sweetwords per user (k). As shown, our chaffing-with-a-hybrid-model HGT approximates the optimal values having only a minor deviation. This means that HoneyGen produces high-quality honeywords even when using $k > 20$ sweetwords per user. As a result, the different authentication systems' operators are encouraged to deploy HoneyGen for producing more than $20 - 1 = 19$ honeywords per user to further limit the adversaries' chances of selecting the real password from the list of sweetwords. Figures 8 and 9 (Appendix B) show the flatness and success-number graphs using different k values. As shown, our chaffing-with-a-hybrid-model HGT approximates the random guessing baseline (optimal solution) in all cases, thus, validating our conclusion that HoneyGen produces realistic-looking honeywords even for larger than 20 k values.

6 FUTURE DIRECTIONS

Despite the empirical evaluation metrics proposed in [22] and [32], namely ϵ-flat and flatness/success-number graphs, a theoretical/formal evaluation is largely absent from the related literature. Such a theoretical evaluation method will enable the accurate security assessment of HGTs as well as the direct comparison between them. In addition, developing empirical standards for HGTs will dramatically boost the interest of the scientific community on inventing HGTs that effectively meet these security requirements. Furthermore, we plan to conduct a user study with a larger pool of participants (more than 100) in order to reduce the uncertainty of our results. Moreover, an interesting future direction is to examine whether or not attackers can identify the passwords database based on the generated honeywords. In such case, a portion of the passwords included in the password corpus may be low hanging fruits, i.e., vulnerable to distinguishing attacks, so a potential adversary may choose to attack them first.

For conducting the normalized Top-PW attack we have calculated the probability of each password in the password file F, based on the probability distribution from a leaked password dataset D. We calculate the probability of each password by counting the number of its occurrences and dividing by the total number of records. However, we should also consider the case where an adversary calculates the probability of each password based on a probabilistic password model such as Markov [26].

Wang et al. [32] explored a series of targeted-guessing attackers who can exploit the victims' PII. These adversaries achieve better attack success rates compared to non-targeted, i.e., PII unaware, ones. In our case, we do not consider such adversaries as all of our datasets are comprised of passwords only and not any other relevant to the user PII. However, HoneyGen's performance, in terms of generating indistinguishable to the real passwords honeywords, should be also evaluated under such attacking strategies that may exploit similar information. We leave this as a future research direction.

HoneyGen's chaffing-with-a-password-model technique can be utilised for password recovery. This is because it returns the $k - 1$ nearest neighbours of a given password according to their cosine similarity. In other words, it returns the $k-1$ most similar passwords that exist in the password corpus. Thus, an authentication system may offer a "forgot your password?" option requiring users to enter the last password they remember. After that, the chaffing-with-a-password-model HGT will retrieve and show its $k - 1$ nearest neighbours. Note that in order to avoid any security implications, the returned passwords should be partly obfuscated with $*$ so that potential attackers cannot utilise this feature for password recovery. Nonetheless, further experimentation is required for accurately concluding about any security implications that this application may have and we leave this as future research direction.

7 RELATED WORK

Bojinov et al. [4] were the first to discuss the use of honeywords to protect a user's list of passwords in a client-side password manager in case the device hosting that list is compromised. They do this by generating several false password lists which look similar, i.e., the contain honeywords, to the real password file F. The authors present, a syntax-based HGT in which honeywords are generated using the same syntax as the real password. In particular, each password is parsed into a series of tokens containing consecutive characters of letters, digits or special characters (symbols).

Then, each character in the tokens is replaced with a randomly selected one that matches the token's type. For example, the password `blue3ball` is modelled as $w_4|d_1|w_4$. A potential honeyword would be $w_4 \longrightarrow red$, $d_1 \longrightarrow 3$, and $w_4 \longrightarrow sword$. However, Bojinov et al.'s approach requires significant changes to the client-side authentication system, which largely affects usability [4]. Furthermore, it cannot generate honeywords of different length or structure which dramatically reduces the honeywords generation spectrum. In contrast, our approach requires minimal changes to the server-side authentication system, while also being capable of generating honeywords with different length and/or structure. While it is trivial to generate honeywords for users that make use of password managers, it is realistic to assume that the largest portion of users avoid using such systems.

Kontaxis et al. [24] proposed SAuth, a protocol which employs authentication synergy among different services. In this context, the authors suggest the use of decoy passwords to tackle the problem of password reuse which can cause their system to fail if a user recycles the same password across all vouching services [24]. However, the key difference between the decoy passwords and honeywords (as suggested in [4, 22]), is that any of those decoys can successfully authenticate the user to the service, whereas the use of a honeyword sets off an alarm as an adversarial attack has been reliably detected. However, even in that case, HoneyGen can be applied for generating decoy passwords which are similar to the user's selected password.

Juels and Rivest [22] suggested the use of honeywords for reliably detecting the disclosure of a password file F on the server-side. At the expense of increasing the storage requirement by 20 times (as they suggest to use 20 sweetwords per user), the authors introduce a simple yet effective solution to sense impersonation. Their approach is more practical compared to [4] as it requires minimal changes to the existing server-side authentication systems and no changes to the client-side system. The authors proposed four legacy UI HGTs and one modified UI HGT. The legacy UI methods are preferred over the modified UI methods due to usability advantages [32]. Thus, we also focus on HGTs for the legacy UI. The four legacy UI HGTs that Juels and Rivest proposed in [22], namely tweaking-by-tail, modelling-syntax, hybrid and simple model, are heavily based on random replacement of characters, thus, inherently unable to generalize to dramatically different password distributions. In contrast, our ML-based HGT leverages representation learning techniques to learn any system-specific underlying password distribution for generating realistic looking honeywords.

Erguler suggested an alternative approach that selects the honeywords from existing user passwords in order to provide high-quality and realistic looking honeywords [14]. However, their HGT is heuristic-based and lacks of systemic evaluation using the relevant evaluation metrics (see [22] and [32]). Furthermore, as outlined in [32], Erguler, I.'s approach has several limitations some of them being the following: (a) it suffers from the "peeling-onions style" distinguishing attack, (b) has critical deployment issues, and (c) dramatically decreases the potential honeywords generation spectrum as it only returns honeywords that already exist in the password corpus as real passwords. Our HGT deals with all the aforementioned problems by combining the benefits of our chaffing-by-tweaking and chaffing-with-a-password-model methods.

Chakraborty and Mondal [8] proposed a modified-UI based HGT, namely Paired Distance Protocol (PDP), that requires a user to remember an extra piece of information, apart from the username and password, for successfully login into a website, namely the password-tail. The password-tail is selected from a list of randomly chosen password-tails created by the system. However, this HGT dramatically decreases the already low user experience of existing password-based authentication systems by forcing users' to remember extra bits of information. In our case, we do not require any modification to the client-side authentication systems, thus, preserving the current usability.

Recently, Fauzi et al. [15] proposed a two-stages PassGAN-based HGT. PassGAN was originally developed for password guessing [20]. PassGAN, being a probabilistic ANN-based model, it automatically approximates the training data password distribution without requiring any heuristic-based password rules. Thus, in [15], Fauzi et al. demonstrated a potential way of adapting a password guessing model, inspired from the common belief in [22] that such models could be potentially produce high-quality honeywords. However, Fauzi et al.'s approach has significant overhead as it generates several lists of possible honeywords for each user account and finally picks the ones that share high proximities to the real passwords (by issuing a discriminator network). Furthermore, it requires that a leaked password dataset is used for training their HGT; potential adversaries that have access to the utilised leaked passwords dataset can dramatically improve their guessing performance. In contrast, our HGT directly generates the $k - 1$ honeywords for each user account. In addition, we suggest the websites' operators to train our HGT on their password corpus to generate system-specific honeywords and avoid any security implications. Finally, Fauzi et al.'s approach should be also evaluated using the normalized Top-PW attacker (as done in this paper) in order to conclude about whether or not their HGT produces vulnerable honeywords for specific user accounts that could be easily distinguished [15].

Wang et al. [32] observed that the relevant literature on honeywords generation merely provide heuristic security arguments when evaluating a HGT without any empirical (or theoretical) evaluation using datasets containing real passwords. Thus, they proposed two evaluation metrics, namely flatness and success-number graphs, for estimating the actual performance of any proposed HGT. Using those metrics, Wang et al. showed that the four legacy UI HGTs proposed in [22] can survive neither PII-unaware nor PII-aware (targeted) attackers. Furthermore, the authors demonstrated that probabilistic password guessing models cannot be readily employed to generate high-quality honeywords, rather than serving as a mean to evaluate the resistance to password distinguishing attacks of other HGTs.

8 CONCLUSION

We propose HoneyGen, a resistant to distinguishing attackers HGT that produces high-quality honeywords according to each authentication system's password distribution. HoneyGen can be applied to any existing password-based authentication system with minimal effort. We evaluate HoneyGen's performance using the state-of-the-art evaluation metrics as well as by conducting a user study with human participants. Our results suggest that HoneyGen is highly effective on timely detecting the leakage of password file F.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for helping us to improve the final version of this paper. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct), No. 830929 (CyberSec4Europe), No. 826278 (Serums), and No. 739578 (RISE) and the Government of the Republic of Cyprus through the Directorate General for European Programmes, Coordination and Development.

REFERENCES

[1] Mohammed H Almeshekah, Christopher N Gutierrez, Mikhail J Atallah, and Eugene H Spafford. 2015. Ersatzpasswords: Ending password cracking and detecting password leakage. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 311–320.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[4] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. 2010. Kamouflage: Loss-resistant password management. In *European symposium on research in computer security*. Springer, 286–302.

[5] Harry N Boone and Deborah A Boone. 2012. Analyzing likert data. *Journal of extension* 50, 2 (2012), 1–5.

[6] William J. Burns. 2019. RockYou Password Leak. https://www.kaggle.com/wjburns/common-password-list-rockyoutxt

[7] Jan Camenisch, Anja Lehmann, and Gregory Neven. 2015. Optimal distributed password verification. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 182–194.

[8] Nilesh Chakraborty and Samrat Mondal. 2017. On designing a modified-UI based honeyword generation approach for overcoming the existing limitations. *Computers & Security* 66 (2017), 155–168.

[9] Nilesh Chakraborty, Shreya Singh, and Samrat Mondal. 2018. On Designing a Questionnaire Based Honeyword Generation Approach for Achieving Flatness. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 444–455.

[10] Antreas Dionysiou, Michalis Agathocleous, Chris Christodoulou, and Vasilis Promponas. 2018. Convolutional Neural Networks in Combination with Support Vector Machines for Complex Sequential Data Classification. In *International Conference on Artificial Neural Networks*. Springer, 444–455.

[11] Antreas Dionysiou and Elias Athanasopoulos. 2020. SoK: Machine vs. machine – A systematic classification of automated machine learning-based CAPTCHA solvers. *Computers & Security* 97 (2020), 101947. https://www.sciencedirect.com/science/article/pii/S0167404820302236

[12] Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Abdelberi Chaabane. 2015. OMEN: Faster password guessing using an ordered markov enumerator. In *International Symposium on Engineering Secure Software and Systems*. Springer, 119–132.

[13] Markus Dürmuth and Thorsten Kranz. 2014. On password guessing with GPUs and FPGAs. In *International Conference on Passwords*. Springer, 19–38.

[14] Imran Erguler. 2016. Achieving flatness: Selecting the honeywords from existing user passwords. *IEEE Transactions on Dependable and Secure Computing* 13, 2 (2016), 284–295.

[15] Muhammad Ali Fauzi, Bian Yang, and Edlira Martiri. 2019. PassGAN-Based Honeywords System. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 179–184.

[16] J Goldberg. 2015. Bcrypt is great, but is password cracking infeasible?

[17] Yimin Guo, Zhenfeng Zhang, and Yajun Guo. 2019. Superword: A honeyword system for achieving higher security goals. *Computers & Security* (2019), 101689.

[18] Robert Hackett. 2017. Yahoo raises breach estimate to full 3 billion accounts, by far biggest known. *fortune.com* (2017).

[19] Patrick Heim. 2016. Resetting passwords to keep your files safe. *dropbox.com* (2016).

[20] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. 2019. Passgan: A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security*. Springer, 217–237.

[21] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[22] Ari Juels and Ronald L Rivest. 2013. Honeywords: Making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 145–160.

[23] Poul-Henning Kamp, P Godefroid, M Levin, D Molnar, P McKenzie, R Stapleton-Gray, B Woodcock, and G Neville-Neil. 2012. LinkedIn password leak: salt their hide. *ACM Queue* 10, 6 (2012), 20.

[24] Georgios Kontaxis, Elias Athanasopoulos, Georgios Portokalidis, and Angelos D Keromytis. 2013. SAuth: protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 187–198.

[25] Russell WF Lai, Christoph Egger, Dominique Schröder, and Sherman SM Chow. 2017. Phoenix: Rebirth of a cryptographic password-hardening service. In *26th USENIX Security Symposium (USENIX Security 17)*. 899–916.

[26] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. 2014. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 689–704.

[27] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2016. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*. 175–191.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[29] Arvind Narayanan and Vitaly Shmatikov. 2005. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM conference on Computer and communications security*. 364–372.

[30] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, and Mauro Conti. 2019. Improving Password Guessing via Representation Learning. *arXiv preprint arXiv:1910.04232* (2019).

[31] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2014. Can long passwords be secure and usable?. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2927–2936.

[32] Ding Wang, Haibo Cheng, Ping Wang, Jeff Yan, and Xinyi Huang. 2018. A Security Analysis of Honeywords. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. 1–16.

[33] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. 2016. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1242–1254.

[34] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. 2009. Password cracking using probabilistic context-free grammars. In *30th IEEE Symposium on Security and Privacy*. IEEE, 391–405.

[35] Roman V Yampolskiy. 2006. Analyzing user password selection behavior for reduction of password space. In *Proceedings 40th Annual 2006 International Carnahan Conference on Security Technology*. IEEE, 109–115.

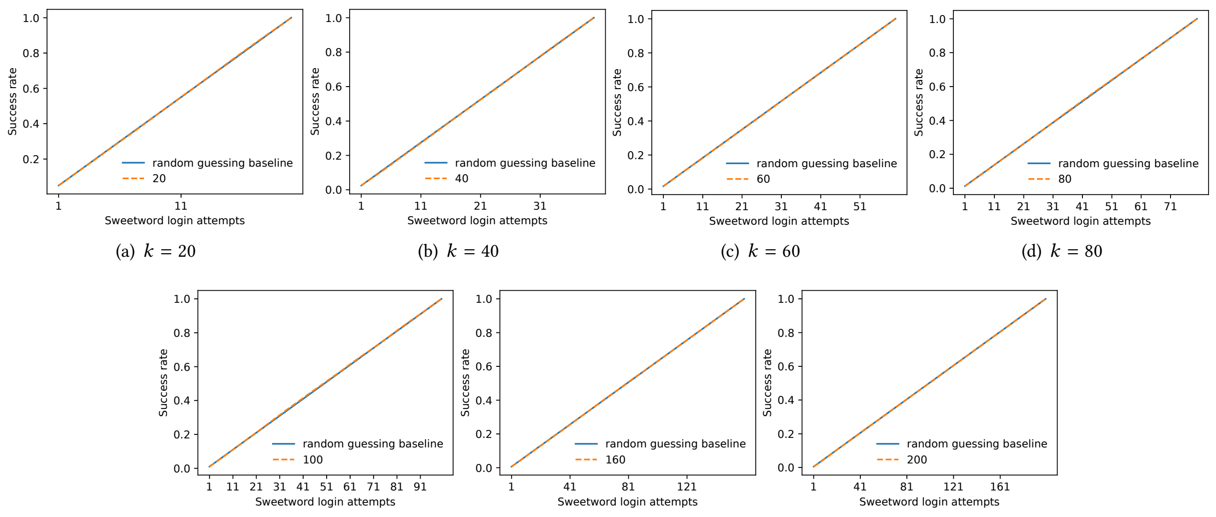
A CHAFFING-BY-TWEAKING PSEUDOCODE

In this part, we provide supplementary material for the *chaffing-by-tweaking* HGT. In particular, Algorithm 1 shows the pseudocode for the detailed operation of this HGT (see Section 3 for a detailed description). Later, Appendix B shows the flatness and success-number graphs for different numbers of sweetwords per user, i.e., k (see Section 5 for more details).

Algorithm 1 Honeywords Chaffing Pseudocode

```
procedure CHAFFING_BY_TWEAKING($real\_password, k, f, p, q$)
    $honeywords = []$                                        ▷ Initialize the honeywords list.
    while $Size(honeywords) < k$ do                          ▷ Repeat until $k$ honeywords are produced.
        $temp$ = ""                                          ▷ Initialize honeyword.
        for $i \leftarrow 0; i < Size(real\_password)$ do    ▷ Loop on each character of the real password.
            if $real\_password[i].is\_lowercase()$ then      ▷ Check if character is a lower-case letter.
                $temp+ = real\_password[i].uppercase(f)$     ▷ Upper-case the character with a probability $f$.
            else if $real\_password[i].is\_uppercase()$ then ▷ Check if character is an upper-case letter.
                $temp+ = real\_password[i].lowercase(p)$     ▷ Lower-case the character with a probability $p$.
            else if $real\_password[i].is\_digit()$ then     ▷ Check if character is digit.
                $temp+ = real\_password[i].change\_digit(q)$ ▷ Replace the digit with a different one with probability $q$.
            else                                             ▷ The character is a symbol.
                $temp+ = real\_password[i].change\_symbols(p)$ ▷ Replace each occurrence of the symbol with a different one.
            end if
        end for
    end while
    if $honeywords.contains(temp)$ then                      ▷ Check if honeyword already exists in the list of honeywords.
        $duplicates = duplicates + 1$                        ▷ Increase duplicates counter.
        if $duplicates$ mod 4 = 0 then                       ▷ Duplicates limit reached so increase the probabilities.
            $p+ = 0.1$
            $q+ = 0.1$
            $f+ = 0.1$
        end if
    else                                                     ▷ New honeyword produced.
        $honeywords.append(temp)$                            ▷ Include honeyword in the honeywords list.
    end if
    return $honeywords$                                      ▷ Return the honeywords list.
end procedure
```

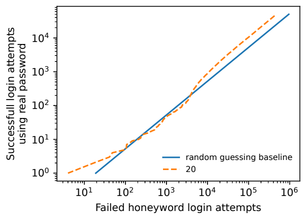
B FLATNESS AND SUCCESS-NUMBER GRAPHS FOR DIFFERENT K



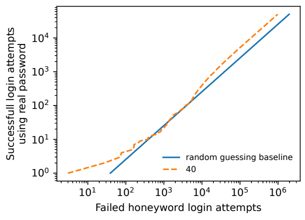
(a) $k = 20$ (b) $k = 40$ (c) $k = 60$ (d) $k = 80$

(e) $k = 100$ (f) $k = 160$ (g) $k = 200$

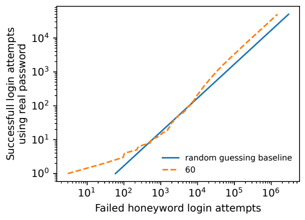
Figure 8: The *flatness* graphs for each number of sweetwords per user (k) using *RockYou* as the target dataset and *Phpbb* as the attacker dataset D.



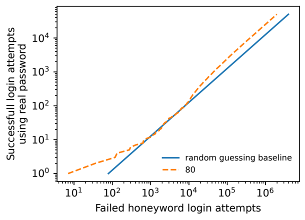
(a) $k = 20$



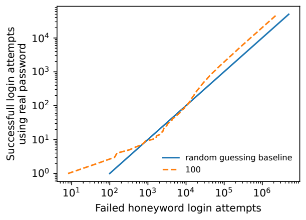
(b) $k = 40$



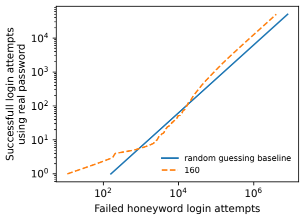
(c) $k = 60$



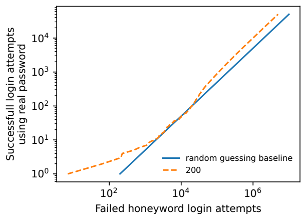
(d) $k = 80$



(e) $k = 100$



(f) $k = 160$



(g) $k = 200$

Figure 9: The *success-number* graphs for each number of sweetwords per user (k) using *RockYou* as the target dataset and *Phpbb* as the attacker dataset D.