Some Remarks on Honeyword Based Password-Cracking Detection

Imran Erguler

TUBITAK BILGEM, Gebze, Kocaeli, Turkey
imran.erguler@tubitak.gov.tr

Abstract. Recently, Juels and Rivest proposed honeywords (decoy passwords) to detect attacks against hashed password databases. For each user account, the legitimate password is stored with several honeywords in order to sense impersonation. If honeywords are selected properly, an adversary who steals a file of hashed passwords cannot be sure if it is the real password or a honeyword for any account. Moreover, entering with a honeyword to login will trigger an alarm notifying the administrator about a password file breach. At the expense of increasing storage requirement by 20 times, the authors introduce a simple and effective solution to detection of password file disclosure events. In this study, we scrutinize the honeyword system and present some remarks to highlight possible weak points. Also, we suggest an alternative approach that selects honeywords from existing user passwords in the system to provide realistic honeywords – a perfectly flat honeyword generation method – and also to reduce storage cost of the honeyword scheme.

1 Introduction

Disclosure of password files is a severe security problem that has affected millions of users and companies like Yahoo, RockYou, LinkedIn, eHarmony and Adobe [1,2], since leaked passwords make the users target of many possible attacks. These recent events have demonstrated that weak password storage methods are currently in place on many web sites. For example, the LinkedIn passwords were using the SHA-1 algorithm without a salt and similarly the eHarmony passwords were also stored using unsalted MD5 hashes [3]. Indeed, once a password file is stolen, by using password cracking techniques like the algorithm of Weir et al. [4] it is easy to capture most of the plaintext passwords.

In this respect, there are two issues that should be considered to overcome these security problems: First, passwords must be protected by taking appropriate precautions and storing with their hash values computed through salting or some other complex mechanisms. Hence, for an adversary it must be hard to invert hashes to acquire plaintext passwords. The second point is that a system should detect whether a password file disclosure incident happened or not to take appropriate actions. In this study, we focus on the latter issue and deal with fake passwords or accounts as simple and cost effective solutions to detect compromise of passwords. Honeypot is one of the methods to identify occurrence of password database breach. In this approach, the administrator purposely creates deceit user accounts to lure adversaries and detects a password disclosure if any one of the honeypot passwords get used [5,6]. This idea has been modifed by Herley and Florencio [7] to protect online banking accounts from password brute-force attacks. According to the study, for each user incorrect login attempts with some passwords lead to honeypot accounts, i.e. malicious behaviour is recognized. For instance, there are 10^8 possibilities for a 8-digit password and let system links 10000 wrong password to honeypot accounts, so the adversary performing the brute-force attack 10000 times more likely to hit a honeypot account than the genuine account. Use of decoys for building theft-resistant is introduced by Bojinov et al. in [8] called as Kamouflage. In this model, fake password sets are stored with the real user password set to conceal the real passwords, thereby forcing an adversary to carry out a considerable amount of online work before getting the correct information. Recently, Juels and Rivest have presented the honeyword mechanism to detect an adversary who attempts to login with cracked passwords [9]. Basically, for each username a set of sweetwords is constructed such that only one element is the correct password and the others are honeywords (decoy passwords). Hence, when an adversary tries to enter with a honeyword, an alarm is triggered to notify the system about a password leakage. The details of the method will be given in the next section.

In this study, we analyze honeyword approach and give some remarks about security of the system. Furthermore, we point out that the key item for this method is the generation algorithm of honeywords such that they shall be indistinguishable from the correct passwords. Therefore, we propose a new approach that uses passwords of other users in the system for honeyword sets, i.e. realistic honeywords are provided. Moreover, this technique also reduces storage cost compared with the honeyword method in [9]. The rest of this paper is organized as follows. In Section 2, we review the honeyword approach and discuss honeyword generation procedures. Section 3 examines security of honeywords and Section 4 gives description of our proposed model. In Section 5 we analyze its security properties and we demonstrate a comparison between our approach and the original methods in Section 6. Finally, in Section 7 we conclude this paper.

2 Honeywords

In this section, we first briefly summarize the honeyword password model proposed by the Juels and Rivest in [9]. Then, we overview the methods on generation of honeywords given in the study and discuss some points that can cause some security problems.

2.1 Review of Honeywords

Basically, the simple but clever idea behind the study is insertion of false passwords – called as honeywords – associated with each user's account. When an adversary gets the password list, she recovers many password candidates for each account and she cannot be sure about which word is genuine. Hence, cracked password files can be detected by system administrator if a login attempt is done with a honeyword by the adversary. We use the notations and definitions depicted in Table 1 to simplify the description of honeyword scheme.

Table 1: Notations.

$H()$	Cryptographic hash function used to compute hash of passwords
u_i	Username for the ith user.
p_i	Password of ith user
W_i	List of potential passwords for u_i
k	Number of elements in W_i
c_i	Index of correct password in list W_i
Gen(k)	Procedure used to generate W_i of length k of sweetwords
sweetword:	Each element of W_i
sugarword:	Correct password in W_i
honeyword:	Fake passwords in W_i

The honeyword mechanism works simply as follows: For each user u_i, the sweetword list W_i is generated using the honeyword generation algorithm **Gen**(k). This procedure takes input k as the number of sweetwords and outputs both the password list $W_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,k})$ and c_i, where c_i is the index of the correct password (sugarword). The username and hashes of the sweetwords as $< u_i, (v_{i,1}, v_{i,2}, \ldots, v_{i,k}) >$ tuple is kept in database of the main server, whereas c_i is stored in another server called as honeychecker. By diversifying the secret information in the system – storing password hashes in one server and c_i in the honeychecker – makes it harder to compromise the system as a whole, i.e. provides a basic form of distributed security [9]. Notice that in traditional password technique $< u_i, H(p_i) >$ pair is stored for each account, while for this system $< u_i, V_i >$ tuple is kept in database, where $V_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,k})$. The login procedure of the scheme is summarized below:

- User u_i enters a password g to login to the system.
- Server firstly checks whether or not $H(g)$ is in list V_i. If not, then login is denied.
- Otherwise system checks to verify if it is a honeyword or the correct password.
- Let $v(i, j) = H(g)$. Then j value is delivered to honeychecker in an authenticated secure communication.
- Honeychecker checks $j = c_i$. If the equality holds, it returns a **TRUE** value, otherwise it responses **FALSE** and may raise an alarm depending on security policy of the system.

Before discussing honeyword generation methods, we want to talk about honeyword generator algorithm **Gen**(). Note that strength and effectiveness of the method indeed is directly related to how the **Gen**() is constructed. Therefore the authors introduce a definition as flatness of **Gen**() such that it measures the chance of detecting sweetword guessing for an adversary. In other words, if a honeyword generation method is ϵ-flat, then she has at least a $1-\epsilon$ chance of picking a honeyword. For example the attacker has a chance of at most 25% of picking the correct password p_i from W_i for $\epsilon = 1/4$. In short, if the algorithm is not flat enough, real password stands out from the remaining fake passwords and an adversary can easily reveal the original one.

2.2 Honeyword Generation Methods and Discussions

The authors in [9] categorize honeyword generation methods into two groups. The first category is consisted of legacy-UI (user interface) procedures and the second one includes modified-UI procedures whose password-change UI is modified to allow for better password/honeyword generation. Take-a-tail method is given as an example of the second category. According to this approach a randomly selected tail is produced for user to append this suffix to her entered password and the result becomes her new password. For instance, let a user enter password *games01*, and then system let propose '413' as tail. So the password of the user now becomes *games01413*. Although this method strengthens the password, to our point of view it is impractical – users often forget the passwords that they determined. Therefore in the remaining parts the analysis we conducted is limited with legacy-UI procedures. Note that some discussed points are indeed mentioned in [9], but we emphasize those to address paramount importance of the selected generator algorithm.

Chaffing-by-tweaking In this method, user password seeds the generator algorithm which tweaks selected character positions of the real password to produce the honeywords. For instance, each character of user password in predetermined positions is replaced by a randomly chosen character of the same type: digits are replaced by digits, letters by letters, and special characters by special characters. Number of positions to be tweak, denoted as t should depend on system policy etc. As an example $t = 3$ and tweaking last t characters may be a method for generator algorithm **Gen**(k, t). Another approach named in the study as *"chaffing-by-tweaking-digits"* is executed by tweaking the last t positions that contain digits. For example, by using last technique for the password *42hungry* and $t = 2$, the honeywords *12hungry* and *58hungry* may be generated.

Remark 1. Many users have propensity to choose the numbers included in passwords related to a special date, e.g. birthday, anniversary or an important historical event. For example 3.6% of the hacked Adobe password hints are related to date [10]. In the light of this fact, it is highly possible that such a password involves a digit sequence like $19xx$, $20xx$ or xx where xx is last two digit of the date. For those passwords by applying the *chaffing-by-tweaking-digits* method, the date digits will be replaced with the randomly selected digits. Hence an adversary who has W_i of a user u_i may easily identify honeywords and recover the correct password. When we examine publicly available leaked passwords hacked from RockYou website (approximately 32 million entries) [11,12], we observe that passwords of numerous users include such a pattern, e.g. **june***xxxx* pattern is selected as a password by 1244 users, where *xxxx* is a date and starts with 19 or 20. Another example should be the password **alex1992** which is seen 47 times in the RockYou password list: Suppose the following honeywords are generated with $t = 4$ and $k = 9$ for this password. Note that the digits in the honeywords seem not relevant but the correct password **alex1992** makes sense for an adversary.

alex6323	*alex9058*	**alex1992**
alex1270	*alex0976*	*alex2785*
alex5469	*alex8147*	*alex9705*

Apart from use of date in passwords, many users prefer to append consecutive numbers to their password heads, like '123', '1234', due to tendency of users to choose rememberable number patterns. By considering the RockYou leaked password database, we realize that about 0.8% of all user passwords – excluding the ones in the top 1000– ends with '123' or '1234' and begins with letters at least one length. The vulnerability issued with the date patterns described above is also valid for those passwords, i.e. an adversary may distinguish the correct password from the sweetwords by just investigating the end digit patterns. Indeed, *chaffing-by-tweaking* method suffers from these type of passwords, because replacing characters of the same type randomly will give the same hint to an adversary in extracting the correct password. Similar patterns and examples of user habits in digit selections can be extended. From a broader perspective these examples show that users mostly do not choose the digits or letters in passwords randomly, so a randomly replacing technique like this model leads an adversary to make a natural selection. In particular, we believe that by deploying *"chaffing-by-tweaking"* model, it is hard to fulfill aims of the honeyword scheme, i.e. all the adversary needs to do is to have a human sense.

Chaffing-with-a-password-model In this approach, the generator algorithm takes the password from the user and relying on a probabilistic model of real passwords it produces the honeywords [9]. The authors give the model of [8] as an example for this method named as *modeling syntax*. In this model the password is splitted into character sets. For instance, *mice*3*blind* is decomposed as 4-letters + 1-digit + 5-letters $\Rightarrow L_4 + D_1 + L_5$ and replaced with the same composition like $gold5rings$.

Another example named as the *simple model* described in the study generates honeywords through a password list: Firstly a password list L is built by combining numerous real passwords and random passwords of varying lengths.

Then a random word is picked from the list with length d. Moreover, with a probability of 0.8 some honeywords are generated as "tough nuts" which will be explained in the next part. As depicted in the algorithm given below, honeyword characters are created by replacing characters of randomly selected words of L in a probabilistic manner:

Algorithm 2.1: SimpleModel(L)

```
$w \leftarrow random(L)$ – comment: randomly returns a word from $L$
$d \leftarrow length(w)$ – comment: returns length of word $w$
$honeyword(1) = w(1)$
for $j \leftarrow 2$ to $d$
  do
    comment: Probabilities of $mod1$, $mod2$ and $else$ are 0.1, 0.4 and 0.5
    if $mod1$
      then $w \leftarrow random(L)$,    $honeyword(j) = w(j)$
    comment: Add character in same position of new random word

      else if $mod2$
      then $w \leftarrow random(L)$,    $honeyword(j) = w(j)$
    comment: Select a random word such that $w(j-1) = honeyword(j-1)$

      else $honeyword(j) = w(j)$
    comment: Proceed with the same word
return ($honeyword$)
```

Remark 2. Leaked password databases has showed us that some passwords has a well known pattern. For example all of the following passwords are involved in the list of 10000 most common passwords [13].

bond007	**james007**
007bond	**007007**

Considering *modeling syntax* method, one can conclude that the honeyword system loses its effectiveness against such passwords, i.e. the correct password has become noticeably recognized by an adversary. In fact this problem seems an inherent weakness of randomly replacement based honeyword methods. Since character groups or individual characters are replaced by a picked character/characters, content integrity of such passwords would be broken and the correct password become quite salient.

Remark 3. Besides the previous point, we want to discuss another issue: If there is a correlation between the username and the password, then the password can be easily distinguished from the honeywords. For example, the password *johndoe*123 with a username *johndoe* can be easily distinguished from corresponding honeywords. The password policy and guidelines should dictate users not to create passwords that is correlated with the username. Unfortunately, some correlations are inevitable like username *peterparker* and the password *spiderman*1992.

Chaffing with "tough nuts" In this method, the system intentionally injects some special honeywords, named as tough nuts, such that inverting hash values of those words is computationally infeasible, e.g. fixed length random bit strings should be set as hash value of a honeyword. Moreover, it is noted that number and positions of tough nuts are selected randomly. By means of this, it is expected that the adversary cannot seize whole sweetword set and some sweetwords will be blank for her, thereby deterring the adversary to realize her attack. In [9], it is discussed that in such a situation the adversary may pause before attempting login with cracked passwords.

Remark 4. Tough nuts are recommended to be used together with other methods to render the adversary's work more challenging and exhaust the attacker. Nevertheless, it has remained an open question in [9] what is the optimal strategy for an adversary when tough nuts are experienced. We believe that "tough nuts" method is a double-edged-sword: Numerous unknowns in the password list may discourage an adversary to proceed mounting her attack. On the other hand, an adversary may suppose that most of the passwords made up of simple words and digit combinations, not a tough nut. Hence, it is reasonable for this adversary to conduct her classic attack with skipping tough nuts contrarily to authors' expectations. Note that for this attack strategy, entropy contributed by the honeywords is decreased, because the tough nuts are ignored by the adversary. For example if in average 2% of all honeywords are tough nuts, apparently this rate will be redundant according to this approach.

Hybrid Method Another method discussed in [9] is combining the strength of different honeyword generation methods, e.g. *chaffing-with-a-password-model* and *chaffing-by-tweaking-digits.* By using this technique, random password model will yield seeds for tweaking-digits to generate honeywords. For example let the correct password be *apple1903.* Then the honeywords *angel2562* and *happy9137* should be produced as seeds to *chaffing-by-tweaking-digits.* For $t = 3$ and $k = 4$ for each seed, the sugarword table given below may be attained:

happy9679	*apple1422*	*angel2656*
happy9757	**apple1903**	*angel2036*
happy9743	*apple1172*	*angel2849*
happy9392	*apple1792*	*angel2562*

Remark 5. Feeding from the strength of *chaffing-with-a-password-model*, this method cuts down chance of adversary in guessing the correct password from the sugarwords. Nevertheless, previous remarks are also valid for this case, e.g. in the above example an adversary may make plausible guesses.

3 Security Analysis of Honeywords

In this part, we investigate security of the honeyword system against some possible scenarios.

3.1 Denial-of-service Attack

In [9], a denial-of-service (DoS) attack is discussed for the following scenario: Adversary knows the used **Gen**() procedure and can produce all possible honeywords for a given a password. For example if *chaffing-by-tweaking-digits* is employed in the system and with a small t adversary may generate whole possible honeywords from a known password. Consider the case, let password of a user be **test42**, then for $t = 2$ she can generate 100 possible honeywords and k of these honeywords are stored in the system password list. Let $\Pr(g = w_i|p_i)$ denote the probability of correctly guessing a valid honeyword of W_i, where correct password p_i is available to the adversary. Hence if this probability is a non negligible value, the adversary may attempt to login with the guessed honeyword to trigger an alarm condition. In fact, this may be serious, if a strong policy is set by the administrator e.g. a global password reset in response to a single honeyword hit. In the above example for $k = 20$ and $t = 2$, $\Pr(g = w_i|p_i) = (k - 1)/99 = 0.19$. In order to mitigate this risk, the authors suggest to choose a relatively small set of honeywords randomly from a larger class of possible sweetwords. For the previous example, success probability of the attacker is about 19% for $k = 20$, while this chance is decreased to 2% by only changing $t = 3$.

Nevertheless, we want to consider the case that an adversary knows m username–password pairs. Perhaps, she previously created these accounts in the system to make a DoS attack. Also suppose that there exists a limit for unsuccessful login attempts as n and success probability of guessing a valid honeyword for a known password is $\Pr(g = w_i|p_i) = \frac{1}{\alpha}$. Then it is more likely that the adversary can succeed in DoS attack, if she makes about α trials. Notice that the adversary can make at most $m \cdot n$ attempts. For the above example $\Pr(g = w_i|p_i) = 0.02$, so it is highly possible to raise an alarm condition if an adversary make about 50 trials. That is to say if the false attempt limit n is (say) five, 10 known account/passwords pairs will be enough to realize the mentioned attack.

Remark 6. In fact, a user should deploy the described attack even she possesses a single account by following the procedure:

Algorithm 3.1: DosAttack($p_i, T(p_i), n$)

```
for $j \leftarrow 1$ to $|T(p_i)|$
    do { if $mod(j, n) = 0$
           then $Login(p_i)$ – comment: To reset unsuccessful login attempts

           else $Login(Guess_j)$ – comment: Make $j^{th}$ guess; $Guess_j \in T(p_i)$
```

In this case, an adversary solely knows a single username and password u_i and p_i respectively. Also, we suppose that system limits for unsuccessful login attempts as n, i.e. after n consecutive wrong password trials the account will be blocked. Nonetheless, if the correct password is entered before n is reached, then system resets the wrong password counter. Hence, as illustrated in the procedure, the adversary logins with the correct password at each n^{th} attempt to avoid block of the account. For example if the used technique for honeyword generation is *chaffing-by-tail-tweaking* and the honeywords are produced by tweaking the characters in the selected last t positions, e.g. $t = 3$, then the adversary should select password such that last t positions only involve digits to reduce entropy about possible characters. For this example $|T(p)| = 1000$, where $T(p)$ stands for the set of sweetwords producible by tweaking p for the selected character positions. Also, we assume that system uses CAPTCHA or a similar mechanism [14,15] to prevent automated login attempts and the adversary is patient to try all guesses manually each of which needs about 5 seconds. Then, she hits a honeyword in about 1.5 hours.

3.2 Brute-force Attack

In previous attack, we point out that if a strict policy is executed in a honeyword detection, system may be vulnerable to DoS attacks affecting the whole system. On the other hand, a soft policy weakens the influence of honeywords. In this regard, we describe the following attack to demonstrate an adversary can capture an amount of accounts in case of a light policy.

We suppose an adversary has obtained a password file F and cracked numerous user passwords. Then, she tries to login with any accounts in the list instead of compromising a specific account. Furthermore, we assume that the adversary has no advantage in guessing correct password by analyzing corresponding honeywords, i.e. $\Pr(g = p_i) = 1/k$. Last, if one of the user's honeywords is entered, system takes the appropriate action according to one of the example policies as follows:

– Login proceeds as usual,
– User's account is shutdown until the user establishes a new password.

The common point of the above policies is that even a honeyword entrance is detected, system gives a local or no response. As a result of this, an adversary can carry out a brute-force search until a successful login is obtained. For example, even a user's account is locked due to a honeyword attempt, she continues to search with another user's account, i.e. single guess for each user. She likely makes a correct guess after k trials, since $Pr(g = p_i) = 1/k$. As an illustrative example for $k = 20$, it is highly possible that the adversary finds a correct password after 20 attempts. It is equivalent to say that if there exists N users in the system, the adversary may recover genuine passwords of N/k users by using brute-force search.

3.3 Choosing Policy

By considering the described attacks and discussions, one can infer that there are two major issues about honeywords. The first issue is flatness of the generator algorithm such that it is directly related to chance of distinguishing the correct password out of respective sweetwords. Thus, if the method is not flat enough, it undermines the main task of the honeywords and an adversary can easily perceive the correct password. Second issue is that what is the chance of an adversary in hitting a honeyword intentionally and triggering a false alarm to render the system in a DoS state. Significance of this issue depends on the adapted policy, e.g. what would be done in case of a false alarm. Under these points, one can see that selection of **Gen**() procedure and an appropriate policy is critically important. Indeed, these security issues are mentioned in [9]. However, the authors propose to adapt factors that reduce the potency of DoS attacks, e.g. increasing t value for chaffing-by-tweaking method instead of insisting on strong policies. Since the main purpose behind the introduction of honeywords is to overcome password-crack detection problem, we believe that security policies should not be loosened to mitigate DoS attacks. In order to hinder DoS attacks, **Gen**() is chosen such that $\Pr(g = w_i | p_i) = \varepsilon$ must be satisfied, where ε a negligible value. Also a limit, as λ, for maximum number of honeyword attempts in a period should be set to prevent brute-force attack. When the limit is exceeded a major appropriate action should be taken, e.g. forcing users to refresh their passwords.

4 A New Approach

Our proposed model is still based on use of honeywords to detect password-cracking. However, instead of generating honeywords and storing them in the password file, we suggest to benefit from existing passwords to simulate honeywords. In order to achieve this, for each account $k-1$ existing password indexes, which we call *honeyindexes*, are randomly assigned to a newly created account of u_i, where $k \geq 2$. Moreover, a random index number is given to this account and the correct password is kept with the correct index in a list. On the other hand, in another list u_i is stored with an integer set which is consisted of the *honeyindexes* and the correct index. So, when an adversary analyzes the two lists, she recognizes that each username is paired with k numbers as *sweetindexes* and each of which points to real passwords in the system. The tentative password indexes hampers an adversary to make a correct guess and she cannot be easily sure about which index is the correct one. It is equivalent to say that to create uncertainty about the correct password, we propose to use indexes that map to valid passwords in the system. The contribution of our approach is twofold. First, this method requires less storage compared to the original study. Second, in previous sections we argue that effectiveness of the honeyword system directly depends on how $Gen()$ flatness is provided and how it is close to human behavior in choosing passwords. Within our approach passwords of other users are used as fake passwords, so guess of which password is fake and which is correct becomes more complicated for an adversary.

4.1 Initialization

Firstly, T fake user accounts (honeypots) are created with their passwords. Also an index value between $[1, N]$, but not used previously is assigned to each honeypot randomly. Then $k-1$ numbers are randomly selected from the index list and for each account a *honeyindex* set is built like $X_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,k})$; one of the elements in X_i is the correct index (*sugarindex*) as c_i. Now, we use two password files as F_1 and F_2 in the main server: F_1 stores username and *honeyindex* set, $< hu_i, X_i >$ pairs as shown in Table 2, where hu_i denotes a honeypot accounts. On the other hand F_2 keeps index number and corresponding hash of password, $< c_i, H(p_i) >$, as depicted in Table 3. Let S_I denote index column and S_H represent the corresponding password hash column of F_2. Then the function $f(c_i)$ that gives password hash value in S_H for the index value c_i can be defined as: $f(c_i) = \{H(p_i) \in S_H :< c_i, H(p_i) > \text{ stored pair of } u_i \text{ and } c_i \in S_I\}$. In order to make points clear, the initialization process is shown within the following example.

Username	Honeyindex Set
agent-lisa	$(93, 16626, \ldots, 94931)$
alexius	$(15476, 51443, \ldots, 88429)$
baba13	$(3, 62107, \ldots, 91233)$
$\vdots$	$\vdots$
zack_tayland	$(1009, 23471, \ldots, 47623)$
zoom42	$(63, 51234, \ldots, 72382)$

Table 2: Example password file F_1 for the proposed model. Each entry has two elements: First one is the username of the account and second element is *honeyindex* set for the respective account. Note that table is sorted alphabetically by the username field.

S_I	S_H
3	$H(p_3)$
7	$H(p_7)$
85	$H(p_{85})$
$\vdots$	$\vdots$
100000	$H(p_{100000})$
100004	$H(p_{100004})$

Table 3: Example password file F_2 for the proposed model. Each entry has two elements: First one is the *sugarindex* of the account and second element is hash of the corresponding password. Note that table is sorted according to the index values.

Example 1. Suppose that a honeypot username/password pair is generated like $< macbeth, master2014 >$ by the system. Then an index number is randomly selected, for instance 1008, and assigned as the correct index of this account. Now F_2 file is updated according to this information as shown below:

Index No	**Hash of Password**
$\vdots$	$\vdots$
1008	$H(master2014)$
$\vdots$	$\vdots$

Then, $k-1$ numbers are randomly chosen from S_I of F_2 and combined with correct index 1008 in a random manner to produce the index group. For instance if $k=5$, such a group $(42, 96104, \mathbf{1008}, 7201, 23008)$ may be generated. In this case F_1 file is seen as below:

Username	**Honeyindex Set**
$\vdots$	$\vdots$
macbeth	$(42, 96104, \mathbf{1008}, 7201, 23008)$
$\vdots$	$\vdots$

4.2 Registration

After the initialization process, system is ready for user registration. In this phase, a legacy-UI is preferred, i.e. a username and password are required from the user as u_i, p_i to register the system. We use the *honeyindex* generator algorithm $\mathbf{Gen}(k, S_I) \rightarrow c_i, X_i$, which outputs c_i as the correct index for u_i and the *honeyindexes* $X_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,k})$. Note that $\mathbf{Gen}(k, S_I)$ produces X_i by randomly selecting $k-1$ numbers from S_I and also randomly picking a number $c_i \notin S_I$. So c_i becomes one of the elements of X_i. One can see that the generator algorithm $\mathbf{Gen}(k, S_I)$ is different from the procedure described in [9], since it outputs an array of integers rather than a group of honeywords. Note, however, that the index array X_i is indeed represents which honeywords are assigned for u_i. In other words, the corresponding honeyword will be the real password whose hash value is $f(x_{i,j})$. After c_i, X_i are obtained, u_i, c_i pair is delivered to the honeychecker and F_1, F_2 files are updated as shown below:

S_I	S_H
3	$H(p_3)$
$\vdots$	$\vdots$
c_i	$H(p_i)$
$\vdots$	$\vdots$
$\vdots$	$\vdots$
100000	$H(p_{100000})$
100004	$H(p_{100004})$

Username	Honeyindex Set
agent-lisa	$(93, 16626, \ldots, 94931)$
alexius	$(15476, 51443, \ldots, 88429)$
baba13	$(3, 62107, \ldots, 91233)$
$\vdots$	$\vdots$
u_i	X_i
$\vdots$	$\vdots$
zack_tayland	$(1009, 23471, \ldots, 47623)$
zoom42	$(63, 51234, \ldots, 72382)$

Table 4: After a registration process, how F_2 file is changed is illustrated on the left, while update of F_1 is shown on the right.

Last, periodically *honeyindexes* of each account should be regenerated. As the number of users in the system increases to provide uniform distribution of *honeyindexes* across S_I, fresh *honeyindex* set must involve numbers from this new larger list. Otherwise, passwords of newly created accounts would not be used as honeywords in the system and it may give a clue to adversary to in guessing the correct password of these new accounts. Note that within a uniform distribution each password is assigned as a honeyword about k times, because there are N passwords but Nk honeywords are needed.

4.3 Honeychecker

In our approach, the auxiliary service honeychecker is employed to store correct indexes for each account and we assume that it communicates with the main server through a secure channel in an authenticated manner. Indeed, it can be assumed that security enhancements for honeychecker and the main server presented in [16] are applied, but it is out scope of this study.

The role and primary processes of the honeychecker are the same as described in the original study [9], except that $< i, c_i >$ pair is replaced with $< u_i, c_i >$ pair in our case. The honeychecker executes two commands sent by the main server:

Set: c_i,u_i

Sets correct password index c_i for the user u_i.

Check: u_i, j

Checks whether c_i for u_i is equal to given j. Returns the result and if equality does not hold, notifies system a honeyword situation.

Thus, the honeychecker only knows the correct index for a username, but not the password or hash of the password. In the following part, functions of the honeycheker is described

4.4 Login Process

System firstly checks whether entered password, g, is correct for the corresponding username u_i. To do this, the hash values stored in F_2 file for the respective indices in X_i are compared with $H(g)$ to find a match. If a match is not obtained, then it means that g is neither the correct password nor one of the honeywords, i.e. login fails. On the other hand, if $H(g)$ is found in the list, then the main server checks whether the account is a honeypot. If it is a honeypot, then it follows a predefined security policy against the password disclosure scenario. Notice that for a honeypot account there is no importance of the entered password is genuine or a honeyword, so it directly manages the event without communicating with the honeychecker. If, however, $H(g)$ is in the list and it is not a honeypot, the corresponding $j \in X_i$ is delivered to honeychecker with username as $< u_i, j >$ to verify it is the correct index. Honeychecker controls whether $j = c_i$ and returns result to the main server. At the same time if it is not equal then it assured that the proffered password is a honeyword and adequate actions should be taken depending on the policy.

5 Security Analysis of the Proposed Model

In this section, we investigate security of the proposed model against some possible attack scenarios. Before, however, we elaborate on the attack strategies, we will first state a set of reasonable assumptions about our approach and related security policies. We suppose that the adversary is able to invert most or many of the password hashes in file F_2. Notice that introduction of this scheme comes with a DoS attack sensitivity in which an adversary deliberately tries to login with honeywords to trigger a false alarm. Hence, suggested policies given below mostly focuses on minimizing DoS vulnerabilities.

– As described in Section 4.4 when a user logins with a wrong password, but not a honeyword, the login fails. If this wrong password is the password of another account in the system and the same user hits this situation more than once, the system should turn on additional logging of the user's activities to detect a possible DoS attack and to attribute the adversary, besides the incorrect login attempt case proceeds as usual.

- If a password in the list is entered in wrong login attempts for more than once, the system should take actions against a possible DoS alarm. These attempts may be done with a single username or with different usernames.
- In order to increase number of unique passwords in the system, i.e. reduce common passwords, users should be forced to adhere to a password-composition policy like basic8 (8 or more characters), comprehensive8 (at least 8 characters including an uppercase and lowercase letter, a symbol, and a digit and not contain a dictionary word), basic16 (16 or more characters) in password creation [17]. The importance of this item is addressed in Section 5.1.
- A username should not be correlated with its password, Remark 3 should be considered.
- If a created password is in a list of 1000 most common passwords, the user should be driven to choose another password.

5.1 DoS Attack

Under this attack scenario as described in Section 3.1, the adversary does not have the password files and their contents. Her main purpose is to trigger a false alarm and to raise a honeyword alarm situation, i.e. depending on the policy some or all parts of the system may be out of service or disabled unnecessarily. We suppose that the adversary has knowledge $m + 1$ username and respective passwords in the system as $(u_a, p_a, \ldots, u_{a+m+1}, p_{a+m+1})$; maybe she intentionally created all of these accounts. In this case, a plausible method for attacking the system is creating m accounts with the same password as p_z, while a single account, u_y, has different password like p_y and entering system with username u_y with password p_z. If p_z is assigned by the system as a honeyword, then the adversary mounts a DoS attack by entering with the system $< u_y, p_z >$ pair. Let $\Pr(p_z \in W_y)$ denote the probability that p_z is assigned as one of the honeywords for u_y; it is also success probability of the adversary for this attack. Since there are $N - m$ passwords different from p_z[1] and k honeywords are assigned to each account:

$$\Pr(p_z \in W_y) = 1 - \left(\frac{N-m}{N}\right)^k. \tag{1}$$

As an illustrative example for $N = 1000000$, $k = 20$ and $m = 100$, from Eq. 1 an adversary succeeds in realizing the described attack with a probability of 0.002. Note that, adversary would like to perform the attack with more accounts like p_y such that in each trial p_z is tested. However, from our assumptions we know that when a password in the system is entered incorrectly by the same username or different usernames for more than once, a DoS attack alarm should

[1] In fact, an adversary may select a common password p_z such that it is already selected by another users, i.e. more than m passwords would become same with p_z. Nevertheless it seems unlikely to find a match with a common password, if a strong password-composition policy is used in the system.

be triggered. Hence, the adversary cannot increase her chance by making more trials for the same known password p_z without being noticed by the system.

5.2 Password Guessing

In this attack, we assume that an adversary has plundered password files F_1 and F_2 from the main server and also obtained plaintext passwords by inverting the hash values. Extracted F_2 file gives $< indexnumber, password >$ pairs to the adversary but they are not directly connect to a specific username. By just analyzing this she cannot exactly determine which password belongs to which user. On the other hand, F_1 gives $username, indexset$ pairs such that for each username k possible passwords exist. Also, we suppose that the adversary has no advantage in guessing the correct password by using specific information of the user such as age, gender and nationality. If the adversary randomly picks an account from the list in F_1 and then tries to login with a guessed password, then her success will depend on: First, the selected account is not a honeypot (decoy) account. Second guessing the correct password p_i out of k sweetwords. Otherwise, the adversary will caught by the system due to a honeyword or a honeyspot. Let $\Pr(success)$ represent the probability that the adversary makes a correct guess for a randomly picked username. Below, we express the probability that the adversary, who makes random trials, is not detected by the system, where we suppose the number of honeypots in the system is T:

$$\Pr(success) = \frac{N - T}{N} \cdot \frac{1}{k}. \tag{2}$$

A convenient choice for T should be $\sqrt{N}$. For $k = 20$ and $N = 1000000$, she picks the correct password p_i with 5% probability. Conversely, the adversary will caught by the system in password guessing attack with a chance of 95%, as long as the password does not carry any information about the username. In contrast to the guess probability in [9] which depends on number of honeywords, the chance depends on two factors – number of honeywords and honeypots. Thus, one can create a higher number of honeypots than $\sqrt{N}$, to increase detection probability of the adversary.

5.3 Brute-force Attack

In this case, we consider the attack described in Section 3.2. We suppose that if a honeypot entrance is detected by the system, it responds with a strong reaction, while a light policy (not suggested) is executed in case of a honeyword detection. So, we assume that even in a honeyword detection the adversary may proceed to make her trials due to light local policies. If, however, a honeypot account is attempted then system follows a marginal policy e.g. demanding all users to renew their passwords. From binomial distribution the probability that the adversary hits at least one honeyspot in her α trials is $Pr(hit \geq 1) = 1 - (N - T/N)^{\alpha}$. Even in this case our approach provides resistance against such an attack, because for $\alpha = 700$, $T = 1000$, $N = 1000000$ values this probability tends to 0.5. It is equivalent to say that in brute-force guess attack, it is likely that the adversary hits a honeypot and system detects the password disclosure situation.

5.4 Same User in Multiple Systems

In [9], the attack scenarios such that a user reuse passwords on two different systems as A and B are investigated. For example suppose A uses honeywords and B has prevalent password storage techniques and a target user u_i shares her password across these two systems. In this case, if an adversary compromises B, it is apparent that honeywords assigned for this user in A contributes nothing at all. Conversely, if the adversary pilfers passwords from A, she can try all sweetwords of the common user u_i in A to verify which is the correct password by submitting to B. If a honeyword is entered to B, it results in an incorrect password screen, while the adversary successfully logins in case of the correct password. Notice that our proposed model is also vulnerable these scenarios: Indeed, if the password is not same but correlated for a user in two distinct domains, then first scenario may be still valid. For example a user has password *bond*007 in B which does not use honeywords. On the other side same user has password $james$007 in domain A which assigns honeywords to these user. Then it is highly possible that an adversary extracts the correct password from the sweetwords, if she has knowledge of *bond*007. So, both of the original method and our approach can not provide resistance against such conditions, as long as users select same or highly correlated passwords in different domains.

6 Comparison of Honeyword Generation Models

In this section, we give comparison of the generation methods including our proposed model with respect to storage cost, DoS resistance and flatness of each algorithm. The results are also depicted in Table 5

6.1 Storage Cost

In this part, we compute storage requirement of our method and compare it with that in [9]. A typical password file system requires hN plus storage for usernames, where N stands for number of users in the system and h denotes length of password hash in bytes. On the other hand this is khN for [9], where k denotes number of sweetwords assigned to each account. Notice that we ignored the storage cost stemmed from usernames, since it is not changed after adaptation of honeywords. The authors also propose a storage optimization technique for the *chaffing-by-tweaking* model such that keeping only hash of a single sweetword, $v_{i,r}$ in database would be enough, because the main server can compute all possible honeywords from an entered proffered password g, e.g. $T(g)$ then check hash of each element in $T(g)$ with stored value $v_{i,r}$ in run time. The authors claim that for a small value of t, $|T(g)|$ will be reasonable. For example if $t = 2$ is selected in case of *"chaffing-by-tweaking-digits"*, $|T(g)|$ becomes 100. Although the solution works and it is an affordable computation cost for the main server, we argue about its applicability, e.g. for each login attempt the server makes 100 more hash computation just to save some storage space.

For our approach we assume that each index is requires 4 bytes and the storage cost becomes[2]:

$$4kN + hN + 4N. \tag{3}$$

To measure the gain in storage compared to original method, we give the ratio as:

$$\frac{4kN + hN + 4N}{khN} = \frac{4k + h + 4}{kh}.$$

Notice that this ratio is independent from number of users and it is less than one for realistic values of k and h. For example let used hash function be SHA-1, i.e. $h = 20$ bytes and $k = 20$ as mentioned in [9], then this ratio will be about 0.25. In other words, for this case our approach needs 1/4 of storage of the original method. Also note that, as k increases storage cost of our scheme is affected by the term $4kN$, while this is hkN for the methods of [9]. So for practical values of h, such as 16 for MD5, 20 for SHA-1 and 32 for SHA-256, growth in storage cost of our method will be less than those of the original ones.

6.2 DoS Resistance

In Section 3.1, we show that *chaffing-with-tweaking-model* may suffer from a DoS attack, due to predictability of the honeywords. Unlikely, *chaffing-with-a-password-model* provides resistance against such an attack, because honeywords are generated by using a list of passwords such that they may be independent from the correct password. In this context, a detailed security analysis of our proposed model is presented in Section 5.1 and we claim that our scheme also thwarts a realizable DoS attack as long as the password policies in Section 5 are adapted and the users obey these tenets in password creation. Note that the authors in [9] avoids direct use of a password list to eliminate a DoS attack threat in case of very common passwords exist in the list. As opposed to this idea, our proposed scheme uses password list in the system as honeywords of a user. However as stated in Section 5, adaptation of a strong password composition policy likely prevents occurrence of common passwords in high numbers, i.e. probability of a common password is assigned as a honeyword for a specific user will be negligibly low. Although, an adversary may hit a real password using a common password in the system, it is not necessarily a honeyword for the corresponding account. Thus, use of real passwords as honeywords does not cause a DoS weakness. Last but not least issue is that in our proposed model addition to honeywords honeypots are employed to detect a password disclosure.

[2] In order to make comparable results, we discarded the storage cost for honeypots– it needs $(4kT + hT + 4T)$ bytes of storage for T honeypots

This facilitates showing a strong response to actions of an adversary, because entering with a honeypot account ensures occurrence of a password leakage. In other words, in our approach administrator should take stronger actions in case of a honeypot attempt compared to entering with honeywords in order to diminish DoS vulnerability.

6.3 Flatness

Remark 1 demonstrates that *chaffing-with-tweaking-model* may leave traces to an adversary in distinguishing the genuine password from the honeywords. As can be inferred from this analysis, the superior method of [9] is the *chaffing-with-a-password-model*, because produced honeywords may seem like user passwords from the perspective of the adversary. Success of the method in flatness depends on how password-model is constructed, for instance the *modeling syntax* yields honeywords depending composition of the user password, thereby a perfect user like behaviour cannot be provided. On the other hand, the *simple model* described in the study may satisfy the distribution of honeywords like user passwords by using a list of real passwords. For our proposed model as described previously passwords of other users become honeywords for a user. Hence, our model satisfies perfect flatness as long as the correct password is not correlated with username as pointed in Remark 3 and investigation of a target user profile (age, gender, religion etc.) gives no advantage to an adversary in password guessing. Comparing our method with the *simple model*, one can see that our method is better than the latter in terms of flatness: The honeywords in the former carry all characteristics of the real passwords in the same system, while the *simple model* generates honeywords artificially despite using real passwords of different list. For example it is well known that users choose segregate their passwords for more-secure and low-secure sites [18,19]. In [20], it is presented that reuse rate of weaker passwords is higher than those of stronger passwords, since the stronger ones are usually created for higher-security sites e.g. banking accounts. Consequently, a password list from a lower-security site password list which is used in the *simple model* for a higher-security site may not be natural. Also, just consider the user passwords for football or movie fan websites. Intuitively, it is likely that many passwords will be related to the context, e.g. passwords include names of heroes, actors, football players or team clubs for movie and football fan sites respectively. Hence, honeywords generated by relying on a general real password list may not exactly match the context of the such a specific website, i.e. an unequivocal pattern incompatibility may exist. This eventually may lead to advantage of an adversary in distinguishing the honeywords.

6.4 Usability

In this part we compare our approach with the *simple model* in terms of practicality and ease of use. By considering the *simple model* whose password list is constructed with composition of numerous real passwords and randomly generated passwords, one can argue about how the real password source is provided. If same resource of real passwords is used in different sites, similar inherited weaknesses related to honeyword generation may be observed. Nonetheless, if use of publicly available password lists is forbidden (as suggested by the authors), then it will not be easy to get required large number of real passwords. Conversely, our approach does not need to use an external real password resource in honeyword generation, rather it just feeds itself. Therefore, we claim that our approach is simpler and more practical for implementation, e.g. an admin does not have to deal with these details.

Method	DoS Resistance	Flatness	Storage Cost
Tweaking	*weak*	*weak*	hN^*
Password-model	*strong*	*strong*†,‡	khN
Our model	*strong*	*strong*‡	$4kN + hN + 4N$

Table 5: Comparison of honeyword generator models. Same expressions of [9] are used for table entries: By *weak* DoS resistance we mean an adversary who knows the password can hit the one of corresponding honeywords with a non-negligible chance; while by strong we mean that this chance is ignorably small. The † is used for condition that its strength depends on how the real password list is used, e.g. the *modeling syntax* may fail as noted in Remark 2. The ‡ is used to mean that condition is satisfied except the case of Remark 3. Also $*$ indicates optimization technique is considered in storage cost calculation.

7 Conclusion

In this study, we have analyzed security of the honeyword system and addressed a number of flaws that need to be handled before successful realization of the scheme. In this respect, we have pointed out that the strength of honeyword system directly depends on how the generation algorithm selected, i.e. flatness of the generator algorithm determines chance of distinguishing the correct password out of respective sweetwords. Another point that we would like to stress is that defined reaction policies in case of a honeyword entrance can be exploited by an adversary to realize a DoS attack. This will be a serious threat if chance of an adversary in hitting a honeyword given the respective password is not negligible. To combat such a problem, also known as DoS resistance, low probability of such an event must be guaranteed. This can be achieved by employing unpredictable honeywords or altering system policy to minimize this risk. Hence, we have noted that the security policy should strike the balance between DoS vulnerability and effectiveness of honeywords. Furthermore, we have demonstrated weak and strong points of each method introduced in the original study. It has been shown that DoS resistance of *chaffing-by-tweaking* method is weak and also its flatness can be questioned by regarding Remark 1. Although some weaknesses of *chaffing-by-tweaking* techniques are accepted by their creators, we believe that it should not be considered as alternative method due to its guessable nature and a potential DoS weakness. Moreover, *chaffing-with-tough nuts* model has been investigated and we have doubted about its favour as opposed to ideas of Juels and Rivest. On the other hand, *chaffing-with-a-password-model* can fulfill its claims provided that the generator algorithm is flat. Nevertheless, how the source of real passwords is attained for this model should be answered before judging its applicability. Finally, we have presented a new approach to make the generation algorithm as close as to human nature by generating honeywords with randomly picking passwords that are belonging to other users in the system. We have compared the proposed model with other methods with respect to DoS resistance, flatness, storage cost and usability properties. The comparisons have indicated that our scheme has advantages over *chaffing-with-a-password-model* in terms of storage, flatness and usability.

References

1. Mirante, D., Justin, C.: Understanding Password Database Compromises. Technical Report TR-CSE-2013-02, Department of Computer Science and Engineering Polytechnic Institute of NYU (2013)
2. Vance, A.: If your password is 123456, just make it hackme. The New York Times **20** (2010)
3. Brown, K.: The dangers of weak hashes. Technical report, SANS Institute InfoSec Reading Room (2013)
4. Weir, M., Aggarwal, S., de Medeiros, B., Glodek, B.: Password cracking using probabilistic context-free grammars. In: Security and Privacy, 2009 30th IEEE Symposium on, IEEE (2009) 391–405
5. Cohen, F.: The use of deception techniques: Honeypots and decoys. Handbook of Information Security **3** (2006) 646–655
6. Almeshekah, M.H., Spafford, E.H., Atallah, M.J.: Improving security using deception. Technical Report CERIAS Tech Report 2013-13, Center for Education and Research Information Assurance and Security, Purdue University (2013)
7. Herley, C., Florencio, D.: Protecting financial institutions from brute-force attacks. In: SEC'08. (2008) 681–685
8. Bojinov, H., Bursztein, E., Boyen, X., Boneh, D.: Kamouflage: Loss-resistant password management. In: Computer Security–ESORICS 2010, Springer (2010) 286–302
9. Juels, A., Rivest, R.L.: Honeywords: Making password-cracking detectable. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13, New York, NY, USA, ACM (2013) 145–160
10. Burnett, M.: The pathetic reality of adobe password hints. `https://xato.net/windows-security/adobe-password-hints`
11. Bonneau, J.: The science of guessing: analyzing an anonymized corpus of 70 million passwords. In: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE (2012) 538–552

12. Malone, D., Maher, K.: Investigating the distribution of password choices. In: Proceedings of the 21st International Conference on World Wide Web. WWW '12, New York, NY, USA, ACM (2012) 301–310
13. Burnett, M.: 10000 top passwords. `https://xato.net/passwords/more-top-worst-passwords/`
14. Ahn, L.V., Blum, M., Hopper, N.J., Langford, J.: Captcha: Using hard ai problems for security. In: Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques–EUROCRYPT'03. Volume 2656 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer-Verlag (2003) 294–311
15. Zhao, L., Mannan, M.: Explicit authentication response considered harmful. In: Proceedings of the 2013 Workshop on New Security Paradigms Workshop–NSPW '13, New York, NY, USA, ACM (2013) 77–86
16. Genc, Z.A., Kardas, S., Sabir, K.M.: Examination of a new defense mechanism: Honeywords. Cryptology ePrint Archive, Report 2013/696 (2013)
17. Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Lopez, J.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE (2012) 523–537
18. Bonneau, J., Preibusch, S.: The password thicket: Technical and market failures in human authentication on the web. In: WEIS. (2010)
19. Notoatmodjo, G., Thomborson, C.: Passwords and perceptions. In: Proceedings of the Seventh Australasian Conference on Information Security–AISC 2009, Australian Computer Society, Inc. (2009) 71–78
20. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th international conference on World Wide Web, ACM Press (2007) 657–666