This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

Generation of Secure and Reliable Honeywords, Preventing False Detection

Akshima*, Donghoon Chang†, Aarushi Goel‡, Sweta Mishra†, Somitra Kumar Sanadhya§
*University of Chicago, USA, akshima@uchicago.edu
†IIIT Delhi, India, {donghoon,swetam}@iiitd.ac.in
‡ Johns Hopkins University, USA, aarushig@cs.jhu.edu
‡IIT Ropar, India, somitra@iitrpr.ac.in

Abstract—Breach in password databases has been a frequent phenomena in the software industry. Often these breaches go undetected for years. Sometimes, even the companies involved are not aware of the breach. Even after they are detected, publicizing such attacks might not always be in the best interest of the companies. This calls for a strong breach detection mechanism. Juels et al. (in ACM-CCS 2013) suggest a method called 'Honeywords', for detecting password database breaches. Their idea is to generate multiple fake passwords, called honeywords and store them along with the real password. Any login attempt with honeywords is identified as a compromise of the password database, since legitimate users are not expected to know the honeywords corresponding to their passwords. The key components of their idea are (i) generation of honeywords, (ii) typo-safety measures for preventing false alarms, (iii) alarm policy upon detection, and (iv) testing robustness of the system against various attacks.

In this work, we analyze the limitations of existing honeyword generation techniques. We propose a new attack model called 'Multiple System Intersection attack considering Input'. We show that the 'Paired Distance Protocol' proposed by Chakraborty et al., is not secure in this attack model. We also propose new and more practical honeyword generation techniques and call them the 'evolving-password model', the 'user-profile model', and the 'append-secret model'. These techniques achieve 'approximate flatness', implying that the honeywords generated using these techniques are indistinguishable from passwords with high probability. Our proposed techniques overcome most of the risks and limitations associated with existing techniques. We prove flatness of our 'evolving-password model' technique through experimental analysis. We provide a comparison of our proposed models with the existing ones under various attack models to justify our claims.

Index Terms—Keywords: Password, Honeywords, Password hash breach, Detection technique, Authentication, Security.

1 Introduction

Password based authentication is the most widely accepted and cost effective authentication technique. In general practice, passwords are never stored in clear text to ensure confidentiality. Instead they are hashed and then stored along with other user related information. The process of performing a one-way transformation on the password and to obtain another string called the 'hashed' password, is known as 'password hashing'. User selected passwords are mostly predictable, since humans have a tendency to choose non-random and easy to remember passwords [1]. 'Dictionary attack' [2] is the most widely used attack technique for retrieving a password from its hash value. In 'dictionary attack', the attacker creates a dictionary of commonly used passwords and computes their corresponding password hashes using the password hashing algorithm. Dictionaries with commonly used passwords can be efficiently created using inexpensive and massively parallelizable hardware such as Graphics Processing Units (GPUs). Any attacker with access to this precomputed dictionary, only needs to get access to the server database. He can then easily compare the entries and learn client passwords.

A *salt* for password hashing refers to an additional public random input to the password hashing algorithm. It is stored in the database along with the password hash. *Salts* help randomize the otherwise deterministic password hashing algorithm. As a result the same password can be mapped to different password hashes. Use of *salt* prevents specialized attacks like the rainbow table attack [3], when considering a large collection of hashes. For simplicity of presentation, we ignore the usage of *salt* in our constructions. However, our proposed schemes can be naturally extended to include the usage of *salts* and it is strongly recommended to use them.

There are several ways to prevent an attacker from performing a dictionary attack by increasing the complexity of this attack manifolds. Making the password hashing algorithm more resource consuming is one way to prevent the adversary from pre-computing the dictionary. This was the main objective behind the Password Hashing Competition (PHC) [4] that ran from 2013-2015. To further improve the security, use of cryptographic module for password hashing is explained in [5]. Another approach is to introduce confusion by adding a list of fake passwords along with the correct password. This would discourage the adversary to mount dictionary attack even after compromising the database. This approach, proposed by Juels et.al. [6], of using fake passwords can help in detecting password database breaches. Specifically, any login attempt with one of the fake passwords detects the breach. The idea was influenced from some other existing techniques mentioned below. The honeypot technique [7], introduced in early 90's, is a system or component which influences the adversary to attack

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

the wrong targets, namely honeypot accounts. Honeypot accounts are fake accounts created by the system administrator to detect password database breaches. Honeytoken is a honeypot that contains fake entries like social security or credit card numbers [8] to identify malicious activity. Kamouflage [9] is a theft-resistant password manager that creates multiple decoy password lists along with the correct password list.

Frequent cases of password database breaches like that of LinkedIn in 2012 [10], Adobe in 2013 [11], eBay in 2014 [12], AshleyMadison in 2015 [13] etc., are indicative of security issues in the current password based authentication systems which can fail to ensure user privacy. In the case of LinkedIn, breach of 6.5 million passwords was reported in 2012. However, in May 2016, additional 100 million passwords were found, that were reportedly leaked in the same breach in 2012 [14]. In response to this, LinkedIn invalidated all the passwords that were not changed since 2012 [10]. No efficient solution to detect such database breaches had been reported in the literature prior to [6]. Therefore, the Honeywords technique [6] is a significant contribution towards detecting breaches of the password database. In this technique, the server generates multiple fake passwords called honeywords for each user, and stores them along with the actual password chosen by the user. Even if an attacker gets access to the password database, she would not be able to distinguish the actual password from honeywords. Therefore with a very high probability, she is expected to enter a honeyword to carry out the attack. If a honeyword is entered instead of the password, the system raises an alarm, thus detecting the compromise of password database. The efficiency of this system basically depends on the ability of the honeyword generation scheme to generate honeywords that are indistinguishable from the real password. The authors in [6], provide some heuristic honeyword generation techniques, along with detailed analysis of the system implementing the honeywords technique. Continuing along the same line of research, we provide an experimental method for quantifying the flatness of honeyword generation schemes. We also implement a distance-measure between password and honeyword using 'Levenshtein distance' [15] to avoid false detection when a legitimate user makes a typing error and enters a honeyword.

Our Contribution: In this work

1) We propose a new attack model called 'Multiple System Intersection attack considering Input'. We show that the 'Paired Distance Protocol' defined in [16] is not secure against this attack model.
2) We propose efficient and practical honeyword generation techniques that can generate honeywords indistinguishable from real passwords.
3) We also suggest a typo-safety measure, to avoid false detection when a legitimate user enters a honeyword because of a typing error. We use 'Levenshtein distance' [15] to ensure that a minimum safe distance is maintained between the password and honeywords.
4) In our experiment, we use publicly available password databases leaked by hackers. Using experiments, we show that our proposed honeyword generation techniques are perfectly flat.
5) To validate our claim, we compare our proposed techniques with the existing honeyword generation techniques on various parameters. The results of these comparisons are summarized in Tables 1 and 2.

The rest of the paper is organised as follows. In Section 2 we present an overview of the 'Honeyword' technique as proposed by Juels et. al. In Section 3, we provide details of the existing attacks for analyzing the security of honeyword based authentication and also explain our proposed attack model for the same. Further, analysis of existing honeyword generation techniques is included in Section 4. Our proposed honeyword generation techniques and security analysis are presented in Sections 5 and 6 respectively. Subsequently, the details of the experiment showing the flatness of our proposed evolving password model is documented in Section 7. Finally, we provide comparison of our proposed technique with existing approaches and conclude the work in Sections 8 and 9 respectively.

2 Overview of the Honeywords Technique [6]

Consider an authentication system with n users $u_1, u_2, \cdots, u_n$ where u_i is the username of the ith user. Let p_i denote the password of ith user. Any typical system maintains a file F listing pairs of usernames and passwords hashed with a password hashing algorithm (PHS) H as

$$(u_i, H(p_i)).$$

To enhance confidentiality of users and to ensure detection of breaches, a system implementing honeywords creates a list W_i of distinct words called sweetwords for each user u_i, represented as

$$W_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,k})$$

where k is a small integer and recommended value of $k = 20$. Exactly one of the values in the list W_i is the actual password, i.e., $\exists j$ such that $w_{i,j} = p_i$. Let $c(i)$ denote the correct index of the password p_i for username u_i in the list W_i, for instance,

$$w_{i,c(i)} = p_i.$$

The correct password is called "sugarword" and the remaining $(k-1)$ words in the list W_i are called called honeywords. These honeywords are generated by the system. Let $Gen(k)$ be the procedure to generate the list W_i and index $c(i)$. The index $c(i)$ is stored in the *honeychecker*, an auxiliary secure server incorporated within the system (the detailed information about *honeychecker* is given later). At the time of authentication, the system interacts with the *honeychecker* to get the correct index of the ith user's password in the list W_i. The database stores the entry for each user u_i, as:

$$(u_i, H_i)$$

where H_i is a k-tuple of hash values of sweetwords. That is,

$$H_i = (H(w_{i,1}), H(w_{i,2}), \ldots, H(w_{i,k})).$$

***Honeychecker*:** It is assumed to be an auxiliary secure server where secret information can be stored. Ideally, we want

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

the *honeychecker* to maintain minimal information about the secret state. It stores the index $c(i)$ corresponding to user u_i. As explained in [6] the server communicates with the *honeychecker* by sending encrypted and authenticated messages over dedicated channels. The *honeychecker* accepts the following two commands:

- **Set** (i, j): Sets the index of sugarword for user i at position j in the array of sweetwords, which means $c(i) = j$.
- **Check** (i, j): Checks if the index of sugarword for user i is at position j. It returns the result of this check to the server. It may additionally raise an alarm if check fails.

The only job of a *honeychecker* is to securely store some information about the secret state and not the entire secret state. Note that this information is useless, unless paired with the password database. This is unlike any crypto-module that contains a key and is expected to perform cryptographic operations. Therefore, even if the *honeychecker* is compromised, the security of the system is only reduced to the level of security without incorporating honeywords and *honeychecker*.

Flatness: Let z denote the adversary's expected probability of guessing the password from the list W_i. An adversary can guess with probability $1/k$, therefore $z \geq 1/k$. A honeyword generation method is "ϵ-flat" for a parameter ϵ, if the maximum value of the guessing probability z for all the adversaries is ϵ and the technique is **perfectly flat** if z is equal to $1/k$. If the value of ϵ is very close to $1/k$ for a technique then it is called 'approximately flat'.
In [6], the authors have illustrated several flat or approximately flat $Gen(k, p_i)$ procedures that are discussed in section 4.

3 Attacks

In this section we describe various attacks for analyzing the security of honeyword generation techniques.

3.1 Denial-of-Service (DoS) Attack

This attack is an attempt to make a service unavailable to its intended users. This is a potential problem associated with the honeyword generation technique. When honeywords are used for authentication and a breach is detected, the system takes action according to the chosen policy. Typically, the policy blocks the user from logging in the system at least temporarily. To impose denial of service to multiple users, an adversary can intentionally provide a honeyword for authentication without actually compromising the complete database. This is possible when the generated honeywords are easy to predict. Therefore, the Gen(k) algorithm is an important deciding factor while analysing the security of a system implementing the honeyword technique.

3.2 Multiple System Intersection considering Output (MSIO) Attack

Humans have a tendency to choose the same password for multiple websites [17]. This behavior opens a new attack surface to compromise the user password assuming

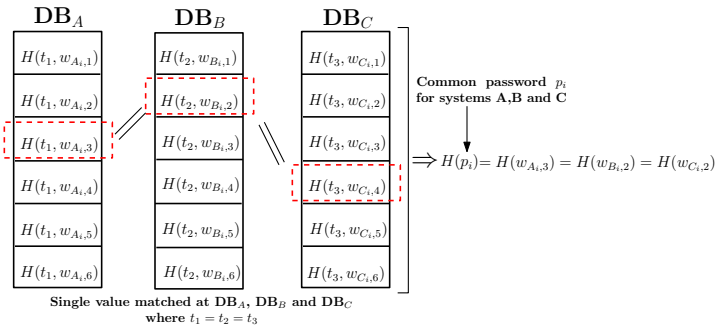

Figure 1. MSIO attack: The system A, B and C contains databases DB$_A$, DB$_B$ and DB$_C$ respectively including the sweetwords for the user i which are available to the attacker. Assuming password p_i is same across all the systems and honeywords are distinct, intersection of the database entries provides the password.

existence of zero collision on the honeywords corresponding to the same password used for multiple sites. Let the databases of three different systems A, B and C as shown in Fig. 1 represented by DB_A, DB_B and DB_C respectively, be available to the attacker. The sweetwords containing distinct honeywords corresponding to user i are represented as $(w_{A_i,j}), (w_{B_i,j})$ and $(w_{C_i,j})$ where $1 \leq j \leq 6$ for system A, B and C respectively. Let, t_1, t_2, t_3 represents the tweak values corresponding the databases DB_A, DB_B and DB_C respectively which may be associated with the input password. These tweak values are fixed for a server and represent for example, a shift by an integer value for mapping the same password to different values. We assume $t_1 = t_2 = t_3$, hence the use of tweak shows no effect on the database values. Therefore randomness of entries of databases only depends on the choice of sweetwords, not on the value of the tweak and we assume the values $t_1 = t_2 = t_3$ are known to the attacker. The list of sweetwords for each system intersects at the value $H(w_{A_i,3}) = H(w_{B_i,2}) = H(w_{C_i,2})$. Assuming honeywords are distinct, this collision helps the adversary to identify the password p_i for user i. The attack complexity is equal to the effort to get p_i from $H(w_{A_i,3})$ or $H(w_{B_i,2})$ or $H(w_{C_i,2})$. Hence, this attack defeats the detection ability of the honeyword technique, which is based on the indistinguishability of the password from the honeywords. As the intersection is taken on the output stored on multiple systems corresponding to the same user password, in this paper we specify this attack as 'Multiple System Intersection considering Output (MSIO) Attack' while it is defined as 'Multiple Systems Intersection' attack in [6].

3.3 Multiple System Intersection considering Input (MSII) Attack

This attack is also motivated by the tendency of humans to choose the same password for multiple websites [17]. Since different websites are assumed to have a different non-random tweak value associated with the user password, thus the same password associated with the same user is mapped to a different value depending on the tweak. Hence intersection of the databases of different websites corresponding to a fixed user does not lead to a common value. As shown in Fig. 2, let the databases of three different

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

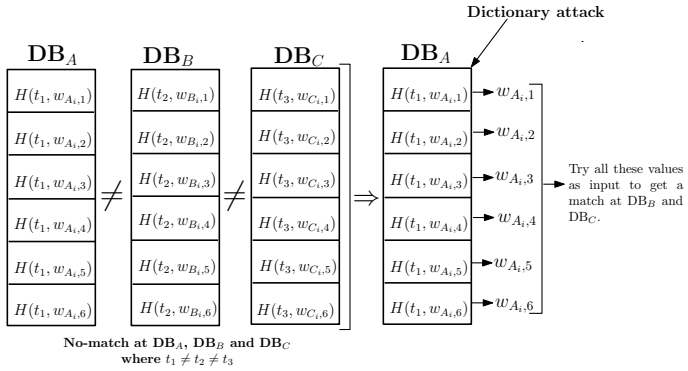

Figure 2. MSII attack: The system A, B and C contains databases DB$_A$, DB$_B$ and DB$_C$ respectively including the sweetwords for the user i which is available to the attacker. All values across all the databases are distinct. Assuming password p_i is same across all the systems, attacker implements dictionary attack on one of the systems (A for this case) and gets all the sweetwords at A. To get the password p_i attacker gives these sweetwords as input to get a match at DB$_B$ and then at DB$_C$.

systems A, B and C are DB$_A$, DB$_B$ and DB$_C$ respectively be available to the attacker. Distinct sweetwords corresponding user i are represented as $(w_{A_i,j}), (w_{B_i,j})$ and $(w_{C_i,j})$ where $1 \leq j \leq 6$ for systems A, B and C respectively. The randomness of entries in the database depend on the choice of the sweetwords, since the corresponding tweak values $t_1 \neq t_2 \neq t_3$ are easy to compute if one or more of t_1, t_2 or t_3 are known. We claim that the overall computation complexity of the MSII attack is asymptotically similar to the MSIO attack. Since the intersection of database entries does not provide any common value, the attacker can only try to guess the sweetwords for a fixed user from one of the available systems such as A. Let an attacker require effort e to get any of the $w_{A_i,j}$ from $H(w_{A_i,j})$ where $w_{A_i,j}$ is the j-th sweetword of user i on system A. For simplicity, we can assume that the effort required to compute t_1 is negligible with respect to e. Then with $k \times e$ effort all k sweetwords can be obtained where one of them is the sugarword/password and $k = 6$ for Fig. 2. When these k-values are tested on systems B and C, only the real password is expected to match the output, since honeywords are expected to be distinct on different systems. Hence, the overall effort only depends on the number of honeywords, which is constant times the effort required in MSIO attack. We refer to this attack as Multiple System Intersection considering Input (MSII) Attack, since it is based on the information obtained by getting an intersection on the inputs. In order to prevent this attack, the random secret must be hard to guess. This can be ensured if multiple systems map the same password to different outputs by ensuring that each system has a distinct random secret.

4 Existing Honeyword Generation Techniques and Their Limitations

The efficiency of database breach detection mechanism is largely dependent on the honeyword generation mechanism. Honeywords generated using perfectly flat honeyword generation techniques are indistinguishable from the real passwords. This prevents the adversary from determining the sugarword from the list of sweetwords. It is challenging to design a perfectly flat honeyword generation technique as passwords are chosen by the users and human behavior cannot be modeled using a generalised thumb rule. In this section, we discuss the limitations while suggesting possible improvements for honeyword generation and provide references when the included analysis is from the existing literatures.

1) Following are the techniques introduced under the 'Legacy-UI' category of honeyword generation. Under this category the choice of password of a user is not influenced by the User Interface (UI). The limitations of this category is that they do not prevent MSIO or MSII attacks.
   a) **Chaffing by tweaking:** In this techniques, t-positions of the password are tweaked to generate honeywords. Characters at each of the t-positions are replaced with randomly chosen characters of the same type. In "Chaffing by tweaking digits", the last t-positions containing digits in the password are tweaked. In [18], the authors acknowledge the bias in human behavior while selecting digits in their passwords. According to report [19], which is based on the analysis of hacked Adobe database, the digits chosen by users follow specific patterns like birthday, any important date, year, some consecutive numbers etc. Therefore, replacing them with some random number makes honeywords, easily distinguishable from password. Ideally such algorithms should try to ape the human behavior while replacing the characters, which is not always easy to formalize.
   If username and passwords are correlated like username:Terasa and password: mother, then it is easy for the adversary to distinguish the password from honeywords. Moreover, it is difficult to prevent such correlation between the username and the password.
   b) The **"Close-number-formation (CNF)"** method suggested in [20] is presented as an improvement over chaffing by tweaking technique. It makes use of two lists: $\{+, -\}$ and $\{1, 2, 3\}$. User chooses password along with one character from each of the lists. The digits used in honeywords are plus/minus of 1/2/3 of the password digits. For example, if the password is 28March2000 then the appropriate range for the possible honeywords can be: 26March1998 - 30March2002, etc. Even though the authors claim that this is an improvement, in this case a user can predict his/her own honeywords with high probability while ideally users should not know the honeywords. False alarms would also be frequent since the honeywords are extremely close to the passwords and there is a high chance that a legitimate user made typo error. To formalize the proposed approach, it needs to consider many different parameters before providing the range like range of dates, possible range of year and sequences etc.
   c) **Chaffing with a password model:** This model is based on publicly available password databases which might also be available to the attacker. There-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

fore, the attacker can also generate the same list of honeywords for a chosen password. The 'modeling syntax' method makes use of the same syntax as that of the password to generate honeywords. For example, the password 'mice3blind' can be decomposed token-wise as '$A_4|D_1|A_5$' where A stands for alphabet-string and D for digit-string. Lengths are also used in suffix. Following the same syntax, a possible generated honeyword could be 'rose5rings'. As mentioned in [18], the list of 10000 most commonly used passwords [21] includes the following

$$bond007 \qquad james007 \qquad 007bond$$

The authors in [18] further explain that with the modeling syntax method, it is easy to distinguish honeywords from the above mentioned passwords if honeywords are generated by randomly replacing the tokens preserving their length. However it is difficult to restrict the user from choosing passwords with such obvious correlations.

d) **Chaffing with 'tough nuts':** In this model, the system uses some tough nuts as honeywords to make it difficult for the adversary to crack the hash of password. Authors in [6], suggest using this method by combining it with some other method. However, according to [18], most of the passwords are simple combinations of digits, alphabets and special characters and not the tough nuts. Therefore, adversary can easily skip to search for the tough nuts. As a result, this strategy will reduce the overall complexity required to compromise the system, thereby defeating one of the purposes of honeywords which is to increase the complexity required to break the system. However, the use of tough nuts as explained in [6] was introduced to discourage the adversary to mount dictionary attack which is not possible in real scenario if it is preferable for the attacker to skip them.

2) Following are the techniques introduced under the 'Modified-UI' category of honeyword generation. In this category the choice of password of a user is influenced by the UI. The motivation behind this category is to introduce randomness in user chosen password to prevent MSIO and MSII attacks.

a) **Take-a-tail [6]:** This method appends system generated digits at the end of user chosen passwords. This approach introduces randomness in user selection. This technique helps to mitigate the correlation problem that could exist between username and password. This is because all honeywords will differ from the password only in the appended digits. It also prevents multiple system intersection attack. However, as explained in [16] it is difficult for users to remember random digits and especially in the case when multiple sites implement this technique and each site gives the user a unique number that needs to be memorized. Also, it is highly likely for the system to raise false alarms, especially in cases where random digits are selected from a small space.

b) The **"Modified-tail"** is proposed in [20] as an improvement over the take-a-tail technique. Number of possible strings of length m, with m different characters is $m!$. The prefixes of each of these strings is also unique. Therefore, in this technique, the user needs to select $m-1$ characters from the provided set of m special characters (to fix the prefix of his own choice) and system appends the remaining one at the end to form the tail of the password. The honeywords are generated by taking all possible combinations of the tail (which are m!-1). For example (from [20]), let the user chosen password be 'tea' and permutation of special characters chosen from the set $S = \{@, ?, |\}$ be '@?|'. Then the list of honeywords are:

$$tea?|@, \ tea?@|, \ tea|?@, \ tea|@?, \ \textbf{tea@?}|, \ tea@|?$$

It is difficult to prevent false detection when a legitimate user makes a typing error in this scenario as honeywords are very close (distance-wise) to the password. Apart from the issue of typing errors, the burden of memorizing extra $m-1$ characters from a system generated list is placed on the user. Therefore, this technique is not at all user-friendly.

c) The **"Caps-key based approach"** proposed in [20] also clams to be an improvement over the 'Take-a-tail' method. In this technique, the user is asked to choose any two characters from the password and convert them to upper case. To generate honeywords, the system then selects any two characters from the password and converts it to upper case. For a password of length l, total $\binom{l}{2} - 1$ such honeywords are possible. For example (from [20]), the password chosen by user is 'animal' and the two characters selected by the user are 'ni'. Therefore the new password becomes 'aNImal'. The honeyword list contains (for $k = 6$),

$$AnImal \ AnimaL \ aNimaL \ \textbf{aNImal} \ ANimal \ anImAl$$

This technique makes the system prone to false detection in case of typing errors. It also increases the possibility of a DoS attack by making the guessing of honeywords easier.

d) The **Paired Distance Protocol** or **PDP** method proposed in [16] creates a secure circular list containing alphabets and digits with default 36 entries in a random order. User selects a random string RS of length l from the set of alphabets and digits along with the password. The distance between any two characters of the RS string is called the 'paired distance'. This paired distance is measured for the RS string considering the position distance (clock-wise) between the same characters of the secure circular list. If the first character of RS is not known then it is difficult to guess RS if only the paired distances are stored in the database. For a circular list of size n, it provides total $(n-1)$ possible values. This idea is used in this technique and honeywords are generated randomly by choosing different paired distances for each password. The *honeychecker* keeps the index of the correct password alongwith the record of first character of the RS string.

Attack: As mentioned in [16]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

- Same circular list could be shared among m different systems and
- User may use same RS for different accounts

MSIO attack is possible in both the above scenarios. Suppose the user chooses the same RS for different systems and the circular list is available alongwith the database entries to the attacker. In this case, the intersection of entries from multiple systems can reveal the actual password and the attacker only needs a maximum of $n-1$ additional trials (where n is the total number of entries in the circular list) to predict the password when the first character is not known. Hence this technique fails to provide security against MSIO attack.

Now suppose different circular lists are provided to different systems and the attacker has complete access to these lists and the databases corresponding to the m-different systems. In this case, assuming that the user chooses the same RS, the attacker has to find all k sugarwords for one of the systems. Trying these sugarwords on other systems, the attacker can find a match for the password, thereby successfully mounting MSII attack. Hence, the proposed method is not secure against MSIO and MSII attack. Apart from the above attacks it forces the user to randomize the choice of RS which is an overhead for the user to memorize.

5 The Proposed Techniques

In this section we propose approximately flat honeyword generation techniques considering typo-safety and other policy choices.

5.1 Password Policy

It is difficult to impose strict rules to enhance the entropy of user chosen passwords. However, users should avoid using dictionary words as passwords, selecting same password on multiple websites, using commonly used passwords, or choosing passwords correlated to the username. For example, if username is 'titanic' and password is 'rose@1997', then there exists a correlation which can help the attacker to uniquely identify the password. A study on password policy as explained in [22] shows that the most restrictive password policies do not provide greater security. Therefore our system imposes the following less restricted and practical to implement conditions on password selection.

1) Username or it's sub-string should not appear in the password.
2) The password should contain at least 8 characters including alphabets, special symbols and digits.

5.2 Typo-Safety

According to an analysis given in [23], the error patterns in mini-QWERTY keyboard show the rate of human errors as: 'Substitution' errors - 40.2%, 'Insertion' errors - 33.2%, 'Deletions errors - 21.4% and 'Transposition' errors - 5.2%. There is a possibility that a legitimate user may enter a honeyword because of typing errors if honeyword is very similar to the password. Such issues can be resolved with a very high probability by maintaining a minimum distance between the password and each generated honeyword. We suggest using 'Levenshtein distance' [15] to compute the distance between password and honeywords. 'Lavenshtein distance' is calculated by counting the number of deletions, insertions, or substitutions required to transform one string into another. It can be used to calculate distances between variable length strings. In this way, all types of human typing errors can be taken into account.

5.3 Proposed Honeyword Generation Techniques

We propose the following techniques for honeyword generation.

1) Legacy-UI password changes: The password selected by the user is not influenced by the UI
   a) **Evolving password model:** We first define the key terms for the better understanding of the scheme. These terms are defined with respect to the available disclosed password databases.
      - **Token:** We consider token as a sequence of characters that can be treated as a single logical entity. In our context, for a given password, tokens are the alphabet-strings(A), digit-strings(D) or special-character-strings(S).
      - **Pattern:** The different combinations of tokens form patterns for a password, e.g., ADS_1, AS_2D, S_1AS_1D etc.
        Note: To create honeywords indistinguishable from user password we do not preserve length of alphabets and digits, however we preserve the length of special-characters. Therefore the length of the special-character is mentioned as a subscript of S in the representation of pattern.
      - **Frequency:** It is the number of occurrences of the tokens or the password pattern in the available password database.

      This evolving password model uses the probabilistic model of real passwords to generate honeywords. Specifically, it pre-computes the frequencies of each password enlisted in an existing database and also computes the frequency of each individual tokens (alphabets-strings, digits-strings, special-characters-strings) of the password following the technique suggested in [24]. The frequencies of the password pattern and its tokens are stored in a file. The database of these frequencies is updated with each new user registration, therefore the frequency list evolves with each new registration. Honeywords are generated by first matching the frequency of the password pattern as a whole and then matching the frequency of the tokens of the password. To match the frequency considering the tokens, we follow the technique proposed in [24], where the probability of the honeyword is the product of the probabilities of the tokens used to derive it. The whole procedure can be covered with two different steps of computation as listed below.
      i) Algorithm to compute frequencies of password pattern and tokens

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

ii) Algorithm to generate honeywords with the help of pre-computed frequencies and update the frequency lists, which means evolving the frequency lists with each new registered user password.

Next we provide an example of generating honeywords from a given password: 'abcde123 %'. To generate the honeywords compute the frequency of the pattern of abcde123 % (ADS_1) and the frequency of the tokens: 'abcde', '123' and '%'. Let frequency of the pattern of 'abcde123 %' matches with the pattern of type AS_1D. Now choose tokens with frequencies similar to the frequencies of 'abcde', '123' and '%' respectively. The token '123' matches with '9', '%' with '_' and 'abcde' with 'secret'. Therefore one of the honeywords is 'secret_9'.

In this technique, the *honeychecker* keeps the index of the correct password. At the time of authentication the index value is verified by the server through a secure communication with the *honeychecker* as explained in [6].

b) **User-profile model:** It generates honeywords by combining some details from the user profile and checks the threshold of minimum distance with the password. One technique to generate honeywords is to create different sets containing tokens of each type, for instance, alphabet-strings, digit-strings and special-character-strings from provided user details. Then make possible combinations of the elements from each sets of tokens. The combinations thus generated are used as honeywords. This technique is best suited when the password contains some substring correlated to the user profile. According to a 2013 report [25] by Google which includes top 10 worst password ideas, users tend to select passwords related to themselves. Hence, this technique generates honeywords indistinguishable from most of the user passwords. To provide typo-safety, a minimum distance between the honeywords and the password should be maintained and we suggest the default value to be 3. One possible approach to create honeywords is following the steps below (considering the password policy mentioned in Section 5.1). For listing the special characters, we can have the list of commonly used special characters.

i) Create separate list of tokens named, token_digits, token_alphabet, token_specialChar from the information provided in user profile.
ii) To create k honeywords, take k different combinations of elements from each token lists, satisfying the password policy of the service.
iii) Compare the tokens of the password with the tokens of the honeyword. Reject the honeyword if more than one token matches with password.

As an example, let the following information about a user profile be known:
Name: Alice Wood; **DoB:** 19/07/1995; **Address:** 54 west 28 street;
Name of the first pet: Jerry
Password: Tom!54street
Then the system can generate the following for this user:
Token_digit={19, 07, 1995, 54, 28}
Token_alphabet={Alice, Wood, west, street, Jerry}
Token_specialChar={/, !}
Honeywords: Wood/1995; Alice_19; Jerry#19wood, Alice@28street
Flatness: As per the report [25] of Google, the top 10 worst password ideas are related to the personal details of the user. Another work on targeted user password guessing [26] leverages the tendency that majority of population uses personal details for choosing password. Therefore, it is easy to interpret that honeywords generated from user profile will be indistinguishable from user passwords for majority of the population. Thus this technique is expected to provide approximate flatness. However, when passwords are not related to the user profile, it is easy to differentiate a password from honeywords. As many users incorporate personal information in their passwords we conjecture that the user profile model will provide approximate flatness for many users though we leave this as an open question for future research.

In this technique, the *honeychecker* keeps the index of the correct password. At the time of authentication the index value is verified by the server through a secure communication with the *honeychecker* as explained in [6].

2) Modified-UI password changes: The password selected by the user is influenced by the UI, while minimizing the usability of the system.

a) **Append-secret model:** In this technique, at the time of registration, the user provides his/her password. The system asks for an extra entry, say l where l ranges from two to four characters in length. System generates a random string r of default length of 3 considering digits, alphabets and special characters. It computes $f(pwd \parallel l \parallel r)$ and outputs x where f is a collision-resistant one-way function. Honeychecker stores both the index of correct password and the value r. Considering the *honeychecker* as a secure database which manages minimum storage, we define the random secret to be a small value and recommend the default length as 3. We also recommend that a different random r is used for each user. This is to prevent the disclosure of (remaining) secrets even when the communication between the server and the *honeychecker* is compromised and the communicated secret is disclosed. The database stores '$H(pwd \parallel x)$' instead of $H(pwd \parallel l)$, where H is a password hashing algorithm. Even after successful implementation of dictionary attack it is still challenging for the attacker to identify the real password. Honeywords generated by this technique differ in the choice of l which influences the value x and the final hash. To provide typo-safety we recommend the length of l from $2-4$ to maintain 'levenshtein distance ≥ 2. As r is randomly selected for each site, even if user selects the same l for multiple sites the intersection of information from multiple sites does not reveal

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

the actual password. Therefore, it provides security against both MSIO and MSII attack.
Example:
Enter Password: abcde
Use any string of length (2-4): 1998
System generated secret: $\$8d$
System computes: $f(abcde \parallel 1998 \parallel \$8d) = 4e7j@$
Database stores: H(abcde$\parallel 4e7j@$)
Therefore, the dictionary attack may reveal the password: $abcde4e7j@$ but not the actual input: $abcde1998$. To get the password, the value of r is required.

Honeychecker: Due to the minimal storage requirement for the *honeychecker*, we do not recommend using random secret r of length 128 bits or above which enhances security. As *honeychecker* is only for secure storage and not for secure computation, we do not allow cryptographic computation inside the *honeychecker* where use of HMAC or other crypto-primitive was possible. If distinct key $\geq$ 128 bit is allowed then it requires more storage. Using a single key for all user is possible but in that case compromising a single communication compromises the whole database. To avoid the communication of the key, a possible approach is to perform all computation inside the *honeychecker*. However, *honeychecker* is not supposed to be a crypto-module. Therefore, we recommend a small and distinct value of r for each user.
We assume the communication between the *honeychecker* and the server is over dedicated channels and/or encrypted and authenticated as explained in [6]. At the time of authentication the secret r is first communicated and then the *honeychecker* verifies the command, 'Check: i, j' as explained in Section 2.

Usability Issues: In our proposed technique the user needs to select and memorize a string l of length 2-4 apart from the initially chosen password. The choice of length from 2-4 for l is for enhancing the security of existing password based authentication with minimal burden on the user. Authors in [27], [28] show that humans are capable of learning random strings over time through spaced repetition. Thus, contrary to the general perception, memorizing the string l should not be a difficult task. However, this could create a usability issue when this technique is implemented in multiple systems and the user is expected to remember a different random string of length 2-4, for each of these systems. A way to get around this issue is to implement the proposed scheme in single sign-on systems [29] [1], the user would only be required to remember a single random string of length 2-4 characters, which would not affect the usability of the system immensely.

1. a mechanism where a single action of authentication permits the authorized user to access multiple applications without having to log-in again during a particular session

5.4 Alarm Policy

Different alarm policies are implemented for passwords based on different levels of hardness to guess the password. This would prevent strong actions in case of DoS attacks targeting the vulnerability of high frequency passwords. As mentioned in [6], policies need not be uniform across a user population. Following are some of the policies suggested in existing literature.

Analysis on Different Alarm Policies

- As suggested in [18], we can limit the number of login attempts that can be made using honeywords in a certain period of time. Based on the limit, an action like refreshing all existing passwords can be taken.
  A single login attempt with honeywords suggest either the compromise of the password database from server or weakness in honeyword generation technique implemented by the server. However, when a threshold limit of attempts is crossed, there is a high chance that it is due to a database breach. In such a situation, refreshing the passwords is the most suitable solution even though it puts burden on the users. Moreover, to ensure security and privacy, enforcing change of passwords after regular intervals of time is a common practice.
- Per-user Policy: As suggested in [6] the system can generate fake accounts called honeypot accounts known only to the *honeychecker*. Any login attempt to the honeypot account with honeyword suggests a breach of password database file. An alarm must be raised in this case. Note that this case requires stronger measures as compared to the measures taken when login attempts to legitimate accounts are made using honeywords. This is because an attacker could have guessed the honeyword for a valid account, or a legitimate user could have made a typing error. The policy for handling such cases must be additionally included with the policies for other cases.
  A system with dummy accounts makes it vulnerable to DoS attacks as explained in [6]. However, we can have a system where the probability of guessing the honeyword is low. If we have a system which allows k honeywords and possible options of honeywords is s for any password then we can restrict this DoS attack. This is possible if the probability to hit the honeyword is very low, which means the value of $(k-1)/s$ is very low.
- Selective alarm Policy: This policy as mentioned in [6] requires the system to raise an alarm if honeyword for a sensitive account such as administrator account is used for login. Such a case can also be exploited to launch a DoS attack. Therefore, this case must also be handled differently from the general policy.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

Proposed Alarm Policy

Analyzing the existing suggestions for deciding the effective alarm policy, we propose the following.

a) If a honeyword, corresponding to a password with very high frequency (e.g., ≥ 0.2) is entered, only inform that user to change his/her password. When honeyword corresponding to a relatively uncommon password is entered, ask all users to change password.
b) If there are repeated attempts within a short span of time, count the number of attempts and send notifications to users, to change passwords, following the approach given in point (a).

On Measuring the Password Probability: The work [24] suggests a method to compute the probability based on the frequency of each token in the password. The frequency can be analysed from compromised real databases which capture the distribution of real user chosen passwords. We can compute the probability of the tokens such as alphabet-strings(A), digit-strings(D) and special-character-strings(S) for a given password from the frequency list computed on real data. The probability of a password can be estimated as the product of the probabilities of all its tokens.

5.5 Managing Old Passwords

As suggested in [6] there is no need to store the hash of the old passwords. Even if the old passwords are required to be stored, they may be stored in a separate database file. This database can be accessed, to ensure that the user does not reuse an old password.

6 Security Analysis

In this section we provide the security analysis of the proposed models against the attacks explained in Section 3, and some other common attacks.

6.1 Brute-Force Attack

For this attack scenario, we assume that the *honeychecker* is not compromised and the adversary gets access to the password database from the server. Then the success probability for the attacker to get the real password depends on the flatness of the honeyword generation technique. In the case where 'user-profile' model is used for honeyword generation, the technique will provide approximately perfect flatness since the probability of a password being related to the user profile is very high. Therefore, in this case, brute-force attack does not help to uniquely identify the password. In the case of 'evolving-password' model, the honeywords are generated such that their frequency is similar to that of the password selected by the user. Hence adversary gets no advantage in guessing the real password without additional information that can be gathered from other attacks such as social engineering attacks. In the case of 'append-secret' model, the attacker can get the value x from brute force attack as explained in Section 5.3. To get the actual input, the adversary needs to guess both the user and system selected random secrets. This sums up to all possible combinations considering user choices for l of length 2-4 and the random choice of length 3 made by the system. This requires extra effort of

$$(\binom{I}{2} + \binom{I}{3} + \binom{I}{4}) \times \binom{I}{3}$$

computations for each password, where I is the input space for random secret and the user input.

6.2 Targeted Password Guessing

Personal information of a user may help an adversary in distinguishing the actual password from honeywords. However, if user-profile model is used for honeyword generation, it increases the difficulty for adversary, provided that passwords of the same user from multiple sites are unavailable to the adversary. In case of evolving-password model, if the password is correlated to the user profile then it is highly likely that adversary will be able to distinguish the password from honeywords. Password guessing is an effective attack against weak popular passwords, passwords reused across multiple website and passwords related to user profile as mentioned in [26]. Therefore, a hybrid approach can be taken by merging both models to generate honeywords. In that case, tokens can be generated from both user profile and password frequency and honeywords can be generated with different combination of the tokens. However, this hybrid approach cannot prevent MSIO or MSII attacks. The append-secret model can provide security against MSIO and MSII attacks and hence prevents the disclosure of the password for a targeted user.

6.3 Denial-of-Service Attack

Honeyword generation from user profile model requires details of the personal information of each individual. It is not unusual for an adversary to collect personal information of any user through social engineering attacks [30]. This makes it easy for an adversary to guess honeywords and carry out a DoS attack on the system. Therefore predicting honeywords is not very challenging and hence DoS attack is easy to implement in such cases. Moreover, targeting some specific individuals would be easier, even though honeyword generation techniques provide flatness close to uniform. This is because honeywords generated from user profile are indistinguishable from password. Therefore considering the possibility of social engineering this technique is not DoS resistant. However, probability of hitting the honeyword can be decreased as explained in Section 5.4 under 'Per-user Policy' by increasing the information space from the user profile. We can say that for user-profile model, given a password, an adversary can provide a honeyword with negligible probability. This is possible when space of honeyword is large for a targeted user else it is difficult to guess the honeywords in general. We categorize this technique as 'moderate' DoS resistant. Therefore, we suggest that the system must take strong actions only if the count of accounts

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

being attacked by DOS attacks is more than a threshold value.

The evolving-password model updates the database of frequencies with each new registration. Therefore the updated database differs from any existing openly available password database files, depending on the number of new registrations. The update of database prevents the adversary from guessing the honeywords with a high probability. Further, the similarities in the frequencies of honeywords with the password provides a nearly perfect flatness. We experimentally show that guessing the password from the honeywords is difficult and the model satisfies perfect flatness.

The append-secret model makes the task of guessing honeywords more challenging, as it implements a one-way function taking input from the user and a random secret from the system to generate a value that is appended to the password.

DoS attack with Fake Account: There could be two preventive approaches to handle the case where an attacker is registering a dummy account and trying to guess the honeyword. One is a preventive approach where the solution is to make the honeyword generation perfectly flat. Hence, guessing honeyword is difficult and DoS attack can be prevented. Other possible solution is to distinguish fake accounts from real ones. This can be implemented through classification using machine learning, for instance, by tracing some common features for fake accounts. For example, fake accounts are inactive after creation, few features that are provided by the service could commonly be untouched for fake accounts etc. Once an account is classified as fake, there is a high probability that a login using honeyword for this account could lead to a DoS attack.

6.4 Multiple System Intersection considering Input/Output Attack

It is common human tendency to use the same password for multiple sites [17]. This facilitates the adversary when attacking multiple distinct systems. The adversary can take an intersection of sugarwords from various systems to guess the password, which means it can mount MSIO attack. If same password maps to distinct values without applying distinct random secrets to different websites, MSII attack is possible. Our proposed append-secret model modifies the password chosen by the user based on a random secret generated by the system. Therefore, even if the adversary gets access to the password files from distinct systems, the same password maps to different values and hence intersection does not reveal the password. Even if the adversary can get all the sugarword corresponsing to a specific system, it is not possible for her to match the entries to another system without knowing the random secret of the two systems. Hence, both MSII and MSIO attacks are not possible against our proposed 'Append-Secret model'.

7 Experimental Data

We set-up an experiment to check flatness of the technique of honeyword generation and the effectiveness of implementing Levenshtein distance to prevent false detection. We simulate our 'evolving password model' using publicly available password files. We generate honeywords and compare the lists of honeywords generated for two passwords of the same frequency. We also check the ease with which the honeywords can be guessed. To generate honeywords, the frequency of all tokens is preserved, while only the length of special characters is preserved.

Following is a detailed description of the implemented algorithms. For our experiment, we use multiple databases containing real user passwords which were compromised by hackers and were made publicly available. Passwords are considered as private data and hence we keep the databases used in this study confidential. Respecting ethical and legal issues in distribution of user private information, we can provide the list to the legitimate researcher who agrees to follow the moral and legal standards of handling private information. The databases can be obtained by contacting the authors, after agreeing to standard ethical practices in handling the same. Our combined database contain more than 4 million unique user passwords. This database is used to pre-compute the frequencies of password patterns as well as frequencies of individual tokens. These computed frequencies are stored in separate files. Our experimental process is described below. The two step procedures to generate the honeywords as explained in Section 5.3 are specified in Algorithm 1 and Algorithm 2.

Let a password in the input database is 'password_123'. Applying Algorithm 1 on the password, we first decide its pattern (syntax). For instance, 'AS_1D' where A= string of alphabet, S_x= special character of length x and D = string of digits. We then store the frequency in a file named patterns_map. The algorithm then splits the password into tokens: {'password', '_', '123'}. The files alphabets_map, digits_map and specialChars_map stores the frequency of the alphabet-string 'password', the frequency of digit-string '123'; and the frequency of the special character '_'.

Using the generated files by implementing the Algorithm 1, we generate the honeywords following Algorithm 2. To test Algorithm 2, we combine the list of leaked password files of two different large organizations. These files were leaked by hackers in 2006 and 2010, respectively. Just like the training database which we already mentioned, we maintain the confidentiality of this test database as well. This database can also be provided to academic researchers after due diligence that they will use standard ethical practice in handling the same. Anyone needing the files can contact the authors.

As we generate honeywords by randomly choosing entries from list_1, list_2, list_3 and list_4 (see Algorithm 2) which are created from an evolving dataset, we get

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2824323, IEEE Transactions on Dependable and Secure Computing

Algorithm 1 To pre-compute the frequencies of password pattern and tokens

Input: A database containing n number of user passwords
for each n in database **do**
    patterns_map = {Store the frequency of the password pattern}
    Split the password into separate tokens and store in different files as
        alphabets_map = {Store the frequency of the string of alphabets}
        digits_map = {Store the frequency of the string of digits}
        specialChars_map = {Store the frequency of the special character}
end for

Algorithm 2 To generate honeywords

Find the frequency of the password pattern from **patterns_map** and update the respective frequency.
List$_1$ = {store all password patterns that have same frequency}
Split the password into individual tokens to compute frequency and also update the respective token frequencies.
List$_2$ = {store all strings of alphabets that have same frequency from **alphabets_map**}
List$_3$ = {store all strings of digits that have same frequency from **digits_map**}
List$_4$ = {store all strings of special symbols that have same frequency from **specialChars_map**}
List_honeywords={}
for $i = 1$ to k (where k=number of honeywords) **do**
    pattern_x=randomly choose a pattern from List$_1$
    for each token in pattern_x **do**
        honeyword = replace the token with elements chosen randomly from
        List$_2$, List$_3$ and List$_4$ depending on the type of token
    end for
    if levenshtein_distance (password, honeyword) ≥ 3
        add the honeyword to List_honeywords
    else do not update the value of k
end for
return List_honeywords

different set of honeywords even for same frequency passwords, with a very high probability. To test the probability, we generate honeywords for different pairs of passwords of same frequency. For our chosen set of password pairs, we observe the following number of entries (the minimum entries obtained over password-pairs of having different frequencies from very high to low):

$$|\text{List}_1| = 10; \;\; |\text{List}_2| = 20; \;\; |\text{List}_1| = 10; \;\; |\text{List}_1| = 10$$

Therefore, number of possible honeywords for passwords of the above case are 20000 and the probability that the two sets of honeywords for pair of similar frequency passwords collide is .001, which is very low. This assures that it is difficult to predict honeywords for two passwords of same frequency. With evolving frequency counts, it is also difficult to create similar list of honeywords for an adversary even if the adversay has the access of same leaked password database.

The Flatness: Honeywords are generated analysing the frequency of the real user passwords. It is very difficult to model human behavior to generate passwords. Therefore, techniques based on the distribution of real user password are the best way to map human tendency towards password generation. Algorithm 2 randomly generates honeywords maintaining a threshold distance (≥ 3) with the password. Hence, the technique is perfectly flat.

8 Comparison with Existing Work

There exist a few works [16], [18], [20] on the analysis of honeyword based detection techniques. A comparison of existing honeyword generation techniques and our proposed methods, considering various advantages and drawbacks, is shown in Table 1 and Table 2. Table 1 compares the existing protocols defined under the 'Modified-UI' category. It is clear from the comparison and the explanation in Section 5.3 that our proposed technique, 'Append-Secret model' is user friendly and provides protection against 'multiple system intersection considering Input/Output' attack. The advantage of 'Modified-UI' technique is that it introduces randomness in user chosen password and prevents the MSII/MSIO attack. The disadvantage of Take-a-tail method under this category is that the randomness is generated by the system and user needs to memorize it. Another technique PDP is an approach where randomness is selected by the user as explained in Section 4. However, introducing the new MSII attack model we have shown that the technique is completely broken. Specifically, it fails to provide security against both MSII and MSIO attacks. Further, this method forces users to randomize their passwords and this is a memory overhead for users. Hence, our proposed technique 'Append-secret model' tries to balance the disadvantages of the above mentioned techniques under 'Modified-UI' approach and the comparison is tabulated in Table 1. Table 2 compares all the existing

honeyword generation techniques considering flatness and significant attack models. The Denial-of-Service (DoS) attack resistance is defined as 'weak' or 'strong' where 'weak' signifies that given a password, an adversary can provide a honeyword with non-negligible probability and 'strong' signifies that honeywords are indistinguishable from passwords. 'Multiple system intersection considering Output (MSIO) and 'Multiple system intersection considering Input (MSII)' attack resistance means that even if accounts of the same user on different systems are compromised, it will not reveal the password. The typo-safety is considered to be taken care of only if it is experimentally performed or explicitly mentioned while describing the technique. Flatness is defined as the probability to distinguish $(k-1)$-honeywords from the password. A technique is user-friendly if the choice of password is not influenced by the UI.

9 Conclusions

In this work, we propose new honeyword generation techniques which overcome several limitations of the existing honeyword generation techniques. Our proposed methods produce honeywords that are indistinguishable from the password and hence achieve 'approximate flatness'. To prevent false detection, in cases where legitimate user unintentionally enters a honeyword, we implement Levenshtein distance to maintain minimum required distance (3 for our experiment) between password and honeywords. We propose a new attack model and show that the 'Paired Distance Protocol' defined in [16] is completely broken in our attack model. The detailed analysis of existing honeyword techniques and their comparison with our proposed techniques is also provided.

References

[1] Jerome H. Saltzer. Protection and the control of information sharing in MULTICS. In *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*.

[2] Robert Morris and Ken Thompson. Password Security: A Case History, 1979. http://cs-www.cs.yale.edu/homes/arvind/cs422/doc/unix-sec.pdf.

[3] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[4] Password Hashing Competition (PHC), 2014. https://password-hashing.net/index.html.

[5] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for password hashing. In Dongdai Lin, Moti Yung, and Jianying Zhou, editors, *Information Security and Cryptology - 10th International Conference, Inscrypt 2014, Beijing, China, December 13-15, 2014, Revised Selected Papers*, volume 8957 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2014.

[6] Ari Juels and Ronald L. Rivest. Honeywords: making password-cracking detectable. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013.

[7] Fred Cohen. The Use of Deception Techniques: Honeypots and Decoys. http://all.net/journal/deception/Deception_Techniques_.pdf.

[8] Lance Spitzner. Honeytokens: The Other Honeypot, 2003. http://www.symantec.com/connect/articles/honeytokens-other-honeypot.

[9] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 286–302, 2010.

[10] Wikipedia contributors. 2012 LinkedIn hack. Wikipedia, The Free Encyclopedia, Date retrieved: 29 May 2016. Available at: https://en.wikipedia.org/w/index.php?title=2012_LinkedIn_hack&oldid=722095159.

[11] Bruce Schneier. Cryptographic Blunders Revealed by Adobe's Password Leak. Schneier on Security, 2013. Available at: https://www.schneier.com/blog/archives/2013/11/cryptographic_b.html.

[12] Swati Khandelwal. Hacking any eBay Account in just 1 Minute, 2014. Available at: http://thehackernews.com/2014/09/hacking-ebay-accounts.html.

[13] Wikipedia contributors. Ashley Madison data breach. Wikipedia, The Free Encyclopedia, Date retrieved: 29 May 2016. Available at: https://en.wikipedia.org/w/index.php?title=Ashley_Madison_data_breach&oldid=721001290.

[14] Troy Hunt. Observations and thoughts on the LinkedIn data breach, 2015. Available at: https://www.troyhunt.com/observations-and-thoughts-on-the-linkedin-data-breach/.

[15] Michael Gilleland. Levenshtein Distance, in Three Flavors. Available at: http://people.cs.pitt.edu/~kirk/cs1501/assignments/editdistance/Levenshtein%20Distance.htm.

[16] Nilesh Chakraborty and Samrat Mondal. A new storage optimized honeyword generation approach for enhancing security and usability. *CoRR*, abs/1509.06094, 2015.

[17] Dinei A. F. Florêncio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 657–666, 2007.

[18] Imran Erguler. Achieving flatness: Selecting the honeywords from existing user passwords. *IEEE Trans. Dependable Sec. Comput.*, 13(2):284–295, 2016.

[19] Mark Burnett. The Pathetic Reality of Adobe Password Hints, 2013. Available at: https://xato.net/the-pathetic-reality-of-adobe-password-hints-bb40fd92220f#.u8deumwif.

[20] Nilesh Chakraborty and Samrat Mondal. Few notes towards making honeyword system more secure and usable. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015*, 2015.

[21] Most common passwords list. Available at: http://www.passwordrandom.com/most-popular-passwords/page/89.

[22] Dinei A. F. Florêncio and Cormac Herley. Where do security policies come from? In *Proceedings of the Sixth Symposium on Usable Privacy and Security, SOUPS 2010, Redmond, Washington, USA, July 14-16, 2010*, 2010.

[23] Edward Clarkson, James Clawson, Kent Lyons, and Thad Starner. An empirical study of typing rates on mini-qwerty keyboards. In *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005*, pages 1288–1291, 2005.

[24] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, 2009.

[25] Techlicious Fox Van Allen techlicious. Google Reveals the 10 Worst Password Ideas, 2013. http://techland.time.com/2013/08/08/google-reveals-the-10-worst-password-ideas/.

[26] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1242–1254, 2016.

[27] Joseph Bonneau and Stuart E. Schechter. Towards reliable storage of 56-bit secrets in human memory. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 607–623, 2014.

[28] Jeremiah Blocki, Saranga Komanduri, Lorrie Faith Cranor, and Anupam Datta. Spaced repetition and mnemonics enable recall of

Table 1
Comparison of Modified-UI honeyword generation techniques.

Take-a-tail [6]	PDP [16]	Append-sectret **[this work]**
System selects the random value for user	User selects the random value	User and system selects the random value
For n different sites n-different values for each user	For n different sites may use same value for each user	For n different sites n-different values for each user
Prevents MSIO attack	MSIO attack Possible	Prevents MSIO attack
Prevents MSII attack	MSII attack Possible	Prevents MSII attack

Table 2
Comparison of honeyword-generation methods. 'weak' DoS Resistant means that given a password, an adversary can provide a honeyword with non-negligible probability; 'moderate' DoS Resistant means that given a password, an adversary can provide a honeyword with non-negligible probability on special scenarios else indistinguishable from passwords; 'strong' DoS Resistant means honeywords are indistinguishable from passwords. MSIO attack and MSII attack resistance means that even if accounts of the same user on different systems are compromised, it will not reveal the password. The typo-safety is considered to be taken care of only if it is experimentally performed or explicitly mentioned while describing the technique. Flatness is defined as the probability to distinguish $(k-1)$-honeywords from the password.

Honeyword Method	Flatness	DoS Resistant	Typo Safety	Legacy UI	MSIO Resistance	MSII Resistance	user-friendly
Tweaking [6]	$< 1/k$	weak	no	yes	no	no	yes
Password-model [6]	no	strong	no	yes	no	no	yes
Tough nuts [6]	N/A	strong	no	yes	no	no	yes
Take-a-tail [6]	1/k	weak	no	no	yes	yes	no
Close-number [20] formation	1/k	weak	no	yes	no	no	yes
Caps-key based [20] approach	1/k	weak	no	no	yes	yes	no
Modified-tail [20] approach	1/k	weak	no	no	yes	yes	yes
PDP [16]	1/k	strong	no	no	no	no	no
Evolving password model **[this work]**	1/k	strong	yes	yes	no	no	yes
User-profile model **[this work]**	$\approxeq$ 1/k	moderate	yes	yes	no	no	yes
Append-secret model **[this work]**	1/k	strong	yes	no	yes	yes	yes

multiple strong passwords. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[29] Bo Li, Sheng Ge, Tianyu Wo, and Dian-fu Ma. Research and implementation of single sign-on mechanism for ASP pattern. In *Grid and Cooperative Computing - GCC 2004: Third International Conference, Wuhan, China, October 21-24, 2004. Proceedings*, pages 161–166, 2004.

[30] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing social networks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 173–187, 2009.