HoneyHash: Honeyword Generation Based on Transformed Hashes

Canyang Shi and Huiping Sun(✉)

Peking University, Beijing, China
shigle@pku.edu.cn, sunhp@ss.pku.edu.cn

Abstract. Since systems using honeywords store a set of decoy passwords together with real passwords of users to confuse adversaries, they are strongly dependent on the algorithm for generating honeywords. However, all of the existing honeyword generating algorithms are based on raw passwords of users and they either need lots of storage space or show weaknesses in flatness or usability. This paper proposes HoneyHash, a new direction of generating honeywords - generating by transforming password hashes. Analyses show that our algorithm attains expected levels of flatness, security, performance and usability.

Keywords: Password · Honeyword · HoneyHash · Transformed hash · Flatness

1 Introduction

A large number of password disclosures were reported in recent years which have been a big threat to password security. For instance, the hashed passwords of 50 million users of Evernote were exposed [1] and similar leakages of password databases also happened in LinkedIn, eHarmony, Yahoo and Adobe [2]. There are several existing mechanisms against password-related attacks including SAuth [3], PolyPassHash [4], ErsatzPassword [5] and Honeyword [6]. Among those existing mechanisms, the honeyword mechanism, which is influenced by the honeypot technique [7] and Kamouflage [8], stands out for its ability to detect attacks against hashed password databases. In a honeyword system, a set of fake passwords are stored together with real passwords in order to confuse adversaries. When an adversary attempts to log in with a fake password, the system can identify this illegal submission and an alarm may be triggered, marking a possible leakage of the password database.

The honeyword generating algorithm is important since the ability of detecting password database leakages is strongly dependent on the quality of honeywords. Until now, all existing algorithms generate honeywords based on raw passwords of users, which need to find a balance point among several factors such as flatness, performance and usability. For instance, Juels and Rivest [6] proposed *chaffing by tweaking* and *take-a-tail* when they first proposed the honeyword mechanism. *Chaffing by tweaking* brings no burden on the memorability of users and has lower time and space complexity, but it cannot generate flat honeywords. *Take-a-tail* can achieve flatness but it puts more stresses on memorability.

M. Asplund and S. Nadjm-Tehrani (Eds.): NordSec 2020, LNCS 12556, pp. 161–173, 2021.
https://doi.org/10.1007/978-3-030-70852-8_10

In this paper, we propose a new honeyword generating algorithm in which honeywords are generated by transforming hashes of original passwords. Comparing with existing algorithms, our algorithm attains higher levels of flatness, security, performance and usability. All the honeywords are transformed hashes which achieve great flatness; the algorithm shows a strong resistance to different password-related attacks including brute-force attacks, dictionary attacks, denial-of-service (DoS) attacks, targeted password guessing and multiple system attacks; the generating process is simple and only one transformed hash is stored in the password database, leading to lower time complexity and storage cost; no extra burden is put on the memorability of users.

The rest of this paper is organized as follows – in Sect. 2 we describe some other mechanisms against password-related attacks followed by the honeyword mechanism. We list existing generating methods of honeywords and analyze them from four aspects including flatness, security, performance and usability. Our new method is presented in Sect. 3 with technical descriptions and basic routines. Then analyses of the proposed method are elaborated from those four aspects thereafter in Sect. 4.

2 Related Works

2.1 Existing Mechanisms Against Password-Related Attacks

There are already several solutions to password-related threats including SAuth [3], PolyPassHash [4], ErsatzPassword [5] and Honeyword [6]. SAuth employs authentication synergy among different services and requires users to log in other servers when visiting a certain server. PolyPassHash employs a threshold cryptosystem to protect password hashes so that they cannot be verified unless a threshold of them have been decoded. ErsatzPassword utilizes a machine-dependent function at the authentication server which can prevent off-site password discovery effectively, and it also employs a deception mechanism to raise an alert if such an action is attempted.

The main idea of the honeyword mechanism is to store a set of passwords (sweetwords) for each account which contains several decoy passwords (honeywords) and the real password (sugarword), so that even if adversaries obtain the password-hash database and recover the original passwords, they cannot discern the real one. When an adversary tries to log in with a honeyword, an alarm may be triggered, informing administrators of a potential leakage of the password database.

2.2 Existing Honeyword Generating Algorithms

Juels and Rivest [6] provided four methods of generating honeywords when they first proposed the honeyword mechanism in 2013. *Chaffing by tweaking* generates honeywords by replacing letters and numbers with other letters and numbers. *Chaffing-with-a-password-model* applies a probabilistic algorithm based on publicly available password databases. *Chaffing with "tough nuts"* generates honeywords which are much harder to crack than the average, e.g., 256-bit, random bit-strings. *Take-a-tail* asks users to add short suffixes to their raw passwords. Then honeywords are generated by changing the suffix of the sugarword.

Imran Erguler [9] proposed another honeyword generating method which maintains a set of integers for each user, corresponding to a set of existing passwords stored in another list. One of the passwords is the sugarword and the others are honeywords. The index of the sugarword is saved in the honeychecker.

Nilesh Chakraborty and Samrat Mondal [11] proposed three new algorithms including *modified-tail*, *close-number-formation* and *caps-key based approach. Modified-tail* is an extension of *take-a-tail* which allows users to have the freedom to choose tails without diluting the security standards. *Close-number-formation* changes the numbers in original passwords slightly. *Caps-key based approach* changes several letters from lower case to upper case. In another paper, Nilesh Chakraborty and Samrat Mondal [13] proposed *paired distance protocol* approach which not only attains a high detection rate, but also reduces the storage cost to a great extent.

Akshima, etc. [18] proposed two legacy-UI models, *evolving password model* and *user-profile model*, and one modified-UI model, *append-secret model*. *Evolving password model* utilizes a probabilistic model of real passwords. *User-profile model* generates honeywords by combining details from user profiles. *Append-secret model* generates honeywords by calculating and appending a secret suffix to the passwords.

Several examples of aforementioned algorithms are presented below (Table 1).

Table 1. Examples of existing generating algorithms

Generating algorithm	Sugarword	Possible honeyword(s)
Chaffing by tweaking	BG+7y45	BG+7q03, BG+7m55, BG+7y45
Chaffing-with-a-password-model	mice3blind	gold5rings
Chaffing with "tough nuts"	/	9,50PEe]KV.0?RIOtc&L-:IJ"b + Wol<*[!NWT/pb
Take-a-tail	RedEye2413	RedEye2582, RedEye2766 (413 is the tail generated by the system randomly)
Modified-tail	tea@?|	tea?|@, tea?@|, tea|?@, tea|@?, tea@|? (@? is the tail chosen by the user from the set of special characters {@, ?, |}.)
Close-number-formation	28May2000	26 May 1999, 25 May 1997, 29 May 2001, 22 May 1998,
Caps-key based approach	aNImal	AnImal, aNimaL, Animal, anImAl
Paired distance protocol	secrettp7	secretk8b, secretekx (tp7 is the tail chosen by the user)
Evolving password model	abcde123%	secret_9
User-profile model	/	Wood = 1995, Alice_19, Jerry#19wood
Append-secret model	abcde1998	abcde4e7j@ (1998 is an extra entry chosen by the user)

2.3 Analysis of Existing Algorithms

The effectiveness of the honeyword mechanism is strongly dependent on the honeyword generating method. In this part, we focus on several factors, including flatness, security, performance and usability, when evaluating existing generating algorithms.

Flatness
Flatness marks the probability of each honeyword to be regarded as the true password from the view of an adversary. A flatter generating method makes it harder for adversaries to discern the sugarword. Among all existing ideas, those algorithms which generate honeywords by changing suffixes achieve better flatness, while tweaking algorithms may not generate flat honeywords in some cases, especially when the sugarword contains a unique pattern and stands out among fake passwords.

Security
The security of an algorithm represents its resistance to password-related attacks such as brute-force attacks, dictionary attacks, denial-of-service (DoS) attacks, targeted password guessing and multiple system attacks. Algorithms like *user-profile model* show lower resistance to targeted password guessing since their honeywords are highly related to personal information. On the other hand, if honeywords are highly predictable, adversaries can use DoS attacks by keeping submitting honeywords deliberately with the help of available true passwords. Some algorithms implement extra mechanisms to defend attacks, but other factors are weakened at the same time.

Performance
Performance measures time and space costs, including time complexity of the generating algorithm and storage space needed by both the password database and the honeychecker. Compared with complex algorithms, those algorithms with simple ideas such as tweaking or changing suffixes have lower time complexity, but nearly all existing algorithms have to store extra k honeywords together with the sugarword, or maintain a huge database of existing passwords which takes a lot of storage space.

Usability
Usability includes some user-related factors. For example, does the system interfere the password choice of the user? Do users need to memorize extra information? What is the possibility of inputting a honeyword by mistake? Among existing methods, generating honeywords by changing suffixes requires users to memorize extra tails, bringing more burdens to users; some other methods, such as the *caps-key based approach*, add extra limits to legal passwords which interferes the password choices of users; for tweaking methods, the typing mistake of a user may be recognized as a submission of honeyword, leading to an alarm which is not expected to be triggered.

3 A New Direction

3.1 Main Ideas

Most of the existing generating algorithms are based on original passwords. They generate honeywords by directly transforming the original password, or by making up a new

password according to the original pattern. Thus, existing methods may have a huge storage cost, and the honeywords may not be flat enough so that adversaries can easily discern the real passwords.

Our algorithm – "transformed-hash", generates a honeyword from the hash of the raw password, and actually, the honeyword is just a transformed hash. The information of the transformation is stored in the honeychecker. The comparison between concerns of existing models and our algorithm is showed below (Fig. 1). There are two main improvements of our algorithm. Firstly, we only store one password hash for each user in the password database, which reduces the storage cost to a large degree. Secondly, instead of generating honeywords based on raw passwords, we focus on hashes and use a transformed hash as a honeyword. Therefore, our method attains expected levels of flatness, security, performance and usability.

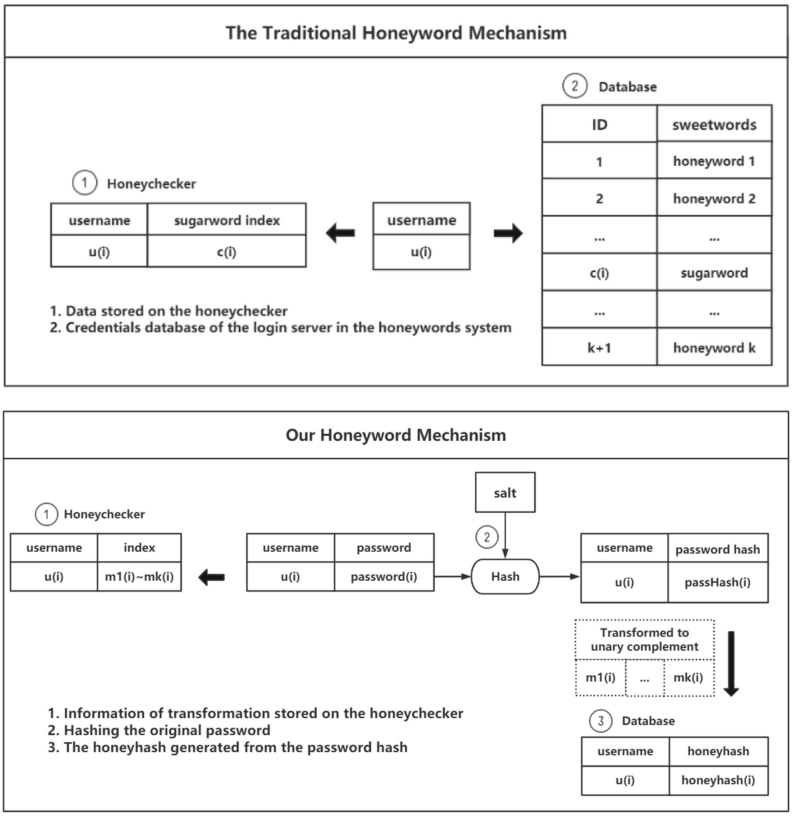

Fig. 1. Comparison between concerns of existing models and our algorithm

3.2 Transforming Methods

Transforming methods are applied to turn the hash of the real password to a fake hash. In this paper, we propose a relatively simple strategy to illustrate our idea. We suppose that the password hashes are 256-bit long. The algorithm transforms a hash by flipping k bits, namely, it selects k bits of the password hash randomly, and then changes them to their unary complements.

It is noteworthy that some transformed hashes may be excluded by adversaries since they do not seem to be hashes of user-generated passwords, so the space of decoy hashes must be huge enough so that enough deceptive keys are incorporated. Let p stand for the ratio of the theoretical key space to the actual key space. For a 256-bit hash, the number of deceptive transformed hashes is

$$\frac{1}{p} \times \binom{256}{\mathrm{k}}$$

To find a proper value for k, we should focus on the number of deceptive hashes (Table 2). The values of parameters should be set properly basing on actual situations. In this paper, as an example, we assume $p = 10^8$ and set the value of k to 5, and the number of deceptive hashes is 88 in this case. Therefore, the system generates a transformed hash by flipping 5 different bits of the original hash.

Table 2. The number of deceptive transformed hashes for different values of k and p

k	p		
	10^7	10^8	10^9
4	17	2	0
5	881	88	9
6	36853	3685	369

3.3 Technical Descriptions

Symbols

u_i: the ith user of the computer system
p_i: the raw password of u_i
H: the cryptographic hash function used in the computer system
$H(p_i)$: the password hash of u_i
$H'(p_i)$: the transformed password hash of u_i
$m_{i1} \sim m_{i5}$: five integers that mark the indexes of the changed bits.

Password Database
The system maintains a file F storing information of usernames and passwords. File F lists the pairs of usernames and transformed password hashes which have the following form:

$$(u_i, H'(p_i))$$

Thus, file F can be described as $\{(u_i, H'(p_i))\}$.

Honeychecker
Like the original honeyword generation methods proposed by Juels and Rivest, this new method also needs a server called honeychecker to check whether the inputted password is a sugarword. For each user u_i, the honeychecker maintains $m_{i1} \sim m_{i5}$ which represent the indexes of the changed bits of $H(p_i)$. Records of the honeychecker database have the following form:

$$(u_i, m_{i1} \sim m_{i5}) = (u_i, m_{i1}, m_{i2}, m_{i3}, m_{i4}, m_{i5})$$

And the honeychecker database can be described as $\{(u_i, m_{i1} \sim m_{i5})\}$.
Our honeychecker receives messages of the following two types:

$$Set\colon i, m_1 \sim m_5$$

Store the indexes of the changed bits for u_i, namely, set the values of $m_{i1} \sim m_{i5}$ to $m_1 \sim m_5$.

$$Check\colon i, m_1 \sim m_5$$

The honeychecker queries its database to get $m_{i1} \sim m_{i5}$. If $m_{i1} \sim m_{i5}$ equals to $m_1 \sim m_5$, then the check succeeds, otherwise the check fails and the honeychecker may raise an alarm.

3.4 Algorithm Routines

Registration
A new honeyword is generated in the process of registration. When a user u_i inputs the expected password p_i, the computer system calculates the password hash $H(p_i)$ at first. Then five random integers $m_1 \sim m_5$ are generated, marking the indexes of bits that will be changed. Later, the system transforms the password hash and gets $H'(p_i)$. Finally, the honeychecker is informed of this operation and the pair $(u_i, H'(p_i))$ is stored into file F.
The routine of registration is presented below:

(1) Read u_i and p_i inputted by the user
(2) Calculate the password hash $H(p_i)$
(3) Generate five different random integers $m_1 \sim m_5$ that are greater than or equal to 0 and smaller than 256

(4) Get $H'(p_i)$ by changing the five bits of $H(p_i)$ to their unary complements
(5) Send *Set*: $i, m_1 \sim m_5$ to the honeychecker
(6) Store the pair $(u_i, H'(p_i))$ into file F

Login

When a user tries to login with a username u_i and a password w_i (the password may be incorrect), the computer system calculates the password hash $H(w_i)$. Then the system queries the file F database and get $H'(p_i)$. In order to detect possible leakages of password databases, an alarm will be triggered when a similarly-transformed password (we still call it a honeyword for convenience) is submitted. According to our transforming method, compared with the hash of the sugarword, all those passwords whose hashes have exactly five different bits are regarded as honeywords. Therefore, if $H(w_i)$ and $H'(p_i)$ have exactly five different bits, then the system sends the indexes of the different bits to the honeychecker and waits for it to have a further check.

The routine of login is presented below:

(1) Read u_i and w_i inputted by the user
(2) Calculate the password hash $H(w_i)$
(3) Look for $H'(p_i)$, the transformed password hash of u_i, in file F. Then compare $H(w_i)$ with $H'(p_i)$. If $H'(p_i)$ is not found in F, or $H(w_i)$ and $H'(p_i)$ do not have exactly five different bits, the login routine fails.
(4) Get the indexes of the different bits $m_1 \sim m_5$
(5) Send *Check*: $i, m_1 \sim m_5$ to the honeychecker
(6) If the check succeeds, then the user login successfully, otherwise the login routine fails. Besides, an alarm may be raised when the check fails, informing an administrator or other party of a possible leakage of the password hash database.

Modification

The routine of modifying the password is almost the same as that of registration. When a user u_i inputs the modified password p_i', the computer system calculates the password hash $H(p_i')$ at first, then generates five new random integers $m_1' \sim m_5'$, marking the five bits of $H(p_i')$ that will be changed. Later, the system transforms the password hash and get $H'(p_i')$. After informing the honeychecker of this operation, the system stores the pair $(u_i, H'(p_i'))$ into file F.

The routine of modifying the password is presented below:

(1) Read u_i and p_i' inputted by the user
(2) Calculate the password hash $H(p_i')$
(3) Generate five different random integers $m_1' \sim m_5'$ that are greater than or equal to 0 and smaller than 256
(4) Get $H'(p_i)$ by changing the five bits of $H(p_i)$ to their unary complements
(5) Send *Set* : $i, m_1' \sim m_5'$ to the honeychecker
(6) Store the pair $(u_i, H'(p_i'))$ into file F

4 Analysis

4.1 Flatness and Security Analysis

Flatness
Flatness influences the difficulty of detecting the sugarword from honeywords. As our model is based on transforming hashes, the honeyword and the sugarword only have similar hashes and their original forms are totally different. For each account, adversaries must find how the hash is transformed before looking for the sugarword.

However, it is really difficult to find the transforming way, and detecting the sugarword is nearly impossible. We suppose that the password hash is transformed by flipping k bits. Then there are $256! \div (251! \times 5!) = 8809549056$ possible original password hashes for $k = 5$, each of which can be regarded as a honeyword. Comparing with existing generating algorithms which generally store about 20 sweetwords for each account, our algorithm has a huge decoy-key space. Most importantly, adversaries cannot rely on any pattern to help them discover the real hash of the original password because each sweetword is a 256-bit hash and shows nothing special.

Brute-Force Attacks and Dictionary Attacks
Adversaries need to enumerate all possible passwords for a brute-force attack. Because of the huge number of honeywords which may cause alarms, adversaries can easily be detected while submitting guesses. Therefore, the proposed algorithm has a strong resistance to brute-force attacks. For attackers, the computational expense of cracking the password database is also higher comparing with that of attacking other existing honeyword systems.

An adversary may also carry out a dictionary attack with the help of a dictionary of user-generated passwords. If the adversary knows that the stored hashes have been transformed by tweaking k bits, he can keep calculating hashes of passwords from the dictionary offline until he discovers a password whose hash value has exactly k different bits comparing with the stored hash, and then he may submit the discovered password. However, if we suppose that the dictionary contains 10^{-8} of all theoretically possible passwords, then there are $256! \div (251! \times 5!) \times 10^{-8} \approx 88$ confusing honeywords when 5 bits are changed. Therefore, if k is chosen properly, the adversary can probably find many confusing answers when carrying out a dictionary attack, leading to a high possibility of being detected when logging in. In sum, our algorithm can defend dictionary attacks effectively.

Denial-of-Service Attacks
Denial-of-service (DoS) attacks can be a potential problem and threat for the honeyword mechanism, especially when the generated honeywords are highly predictable. If an adversary has not compromised the password database F but successfully knows the original password of the user in some way, he has a great chance to guess honeywords and submits them to the system deliberately. The system may force a global password reset or blocking the whole web-server in response to the submission of one or more honeywords.

The key point to mitigating DoS attacks is reducing the chance of triggering an alarm maliciously. One way is to increase the difficulty of guessing honeywords with the help of known sugarwords or honeywords. According to our algorithm, the sugarword and honeywords can be totally different since they only have similar hashes. Knowing a sugarword or a honeyword brings no benefit to adversaries when trying to discover other honeywords, so the discovery of each honeyword needs a dictionary attack. Besides, the alarming mechanism can be changed so that alarms cannot be triggered unless an enough number of different honeywords are submitted to the system. For those adversaries who have obtained the original password in some way, they have to use dictionary attacks repeatedly until they have found enough different honeywords, so they almost have no chance to trigger an alarm on purpose. Therefore, comparing with existing generating algorithms, our model can help the system to defend DoS attacks to a large degree.

Targeted Password Guessing

Adversaries may also use targeted password guessing attacks by detecting the sugarword with the help of the personal information of users, which can be easily obtained based on usernames or the social network graphs, especially for those users whose passwords are highly related to their personal information.

The best way to prevent targeted password guessing attacks is using irrelevant honeywords so that personal information brings no benefit to adversaries. In our model, only one honeyword, the transformed password hash, is stored for each account and no personal information is involved because of the transformation, adversaries cannot expect to gain any advantage of detecting the sugarword.

Multiple Systems

Users prefer setting the same password across different systems. In that case, adversaries may get advantages for discovering the sugarword. Juels and Rivest described *intersection attacks* and *sweetword-submission attacks* which are related to multiple systems. If a set of distinct honeywords are stored for each account, an adversary can compromise the password database on several different systems and learn the real password from the intersection of those sweetword sets. On the other hand, if a part of those systems do not use honeywords in order to avoid *intersection attacks*, adversaries can submit sweetwords as password guesses to the honeyword-absent systems without risks of detection.

One way to make the system resistant to such attacks is enlarging the intersection of sweetword sets among different systems. If the intersection has many sweetwords instead of one, then adversaries cannot identify the sugarword from it. In our model, when a user employs the same password on two different systems which both transform original hashes by flipping k bits, then the intersection of the two sets has at least $\binom{2k}{k}$ sweetwords (6 sweetwords when $k = 2$ and 252 sweetwords when $k = 5$). Therefore, even if an adversary has compromised both systems and has found the intersection of sweetword sets, he still cannot discover the exact sugarword. Thus, our algorithm has a higher resistance to multiple system attacks.

4.2 Performance and Usability Analyses

Performance

Time Complexity
Our algorithm has a comparatively lower time complexity. The idea and routines of our algorithm are simple and the whole generating process can be divided into two parts, calculating a hash and transforming a hash. Transforming a hash can be done easily and quickly with the use of bit operation. Calculating the hash is the most time-consuming part whose time cost depends on the hash method. Therefore, the time complexity of our algorithm is nearly the same as that of calculating a hash, which is necessary for every generating method.

Storage Cost
The hash-based generating method also has a low storage cost. Nearly all existing methods store k sweetwords for one account. If we suppose that $k = 20$ and sweetwords are 256-bit hashes, then for each account, the sweetwords take $20 \times 256 = 5120$ bits in the password database and the honeychecker needs $\log_2 20 \approx 5$ bits to store the index of the sugarword. The total amount of storage cost is $5120 + 5 = 5125$ bits. However, in our model, the transformed hash takes 256 bits in the password database and the indexes of the changed bits $m_{i1} \sim m_{i5}$ are stored in the honeychecker which take $5 \times \log_2 256 = 40$ bits, so the storage space needed for each account is just $256 + 40 = 296$ bits. Therefore, comparing with other existing mechanisms, the storage cost of our algorithm is reduced to a large degree.

Usability

Stress on Memorability
This algorithm puts negligible burdens on the memorability of users. Users do not need to memorize a tail or other extra information since the final password is the same as the expected one. Users can choose their passwords freely without being limited or interfered by the system. They can even set a relatively simple password or one that is related to their personal information because even for a simple or person-related password, the transformed password hash is still hard to be decoded. In addition, since the algorithm is resistant to multiple-system attacks, a user can use same passwords for different systems, which brings negligible stresses on the memorability of users when setting passwords for a new account.

Typo-Safety
When typing the password, a user may make mistakes and input a wrong one, and a worse case is that the wrong password happens to hit a honeyword which triggers an alarm. This probably happens especially when honeywords are almost the same as the sugarword. In our algorithm, however, comparing with the sugarword, those passwords which can cause an alarm just have similar hashes, and their original forms can be totally different from the true password, so it is impossible for a user to input a honeyword by error. Thus, this method can be considered as typo-safe.

5 Discussions

In this paper, we present a simple algorithm of the transforming method. The way of transforming hashes can be changed but the values of parameters should be set properly basing on actual situations. For instance, if the password hash is transformed by flipping k bits, then k can affect the number of theoretical honeywords. When k is too small, adversaries can easily find the sugarword by carrying out a dictionary attack; when k is too big, adversaries may find it nearly impossible to find the sugarword and do not try submitting any guesses in the end, and hence the system may lose the ability to detect a potential leakage of the password database.

6 Conclusion

In this paper, we propose HoneyHash, a new direction of generating honeywords which overcomes some inherent defects of existing generating algorithms. It turns out that the proposed methodology meets high standards of flatness, security, performance and usability. We hope that the proposed algorithm can encourage more systems to use the honeyword mechanism.

References

1. Gross, D.: 50 million compromised in Evernote hack. In: CNN (2013)
2. Gaylord, C.: LinkedIn, Last.fm, now Yahoo? Don't ignore news of a password breach. In: Christian Science Monitor (2012)
3. Kontaxis, G., Athanasopoulos, E., Portokalidis, G., Keromytis, A.D.: Sauth: protecting user accounts from password database leaks. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 187–198. ACM (2013)
4. Cappos, J.: PolyPassHash: protecting passwords in the event of a password file disclosure. In: Password Hashing Competition (PHC) (2014)
5. Almeshekah, M.H., Gutierrez, C.N., Atallah, M.J., Spafford, E.H.: ErsatzPasswords: ending password cracking and detecting password leakage. In: Proceedings of ACSAC, pp. 311–320 (2015)
6. Juels, A., Rivest, R. L.: Honeywords: making password-cracking detectable. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 145–160. ACM (2013)
7. Cohen, F.: The use of deception techniques: honeypots and decoys. In: Bidgoli, H. (ed.) Handbook of Information Security, vol. 3, pp. 646–655 (2006)
8. Bojinov, H., Bursztein, E., Boyen, X., Boneh, D.: Kamouflage: loss-resistant password management. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 286–302. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15497-3_18
9. Erguler, I.: Achieving flatness: selecting the honeywords from existing user passwords. IEEE Trans. Depend. Secur. Comput. **13**(2), 284–295 (2016)
10. Chatterjee, R., Bonneau, J., Juels, A., Ristenpart, T.: Cracking-resistant password vaults using natural language encoders. IEEE Secur. Privacy 481–498 (2016)
11. Chakraborty, N., Mondal, S.: Few notes towards making honeyword system more secure and usable. In: Proceedings of 8th International Conference Security and Information Network, pp. 237–245 (2015)

12. Golla, M., Beuscher, B., Dürmuth, M.: On the security of cracking-resistant password vaults. In: Proceedings of ACM CCS, pp. 1230–1241 (2016)
13. Chakraborty, N., Mondal, S.: On designing a modified-UI based honeyword generation approach for overcoming the existing limitations. Comput. Secur. **66**, 155–168 (2017)
14. Pasquini, C., Schöttle, P., Böhme, R.: Decoy password vaults: at least as hard as steganography? In: De Capitani di Vimercati, S., Martinelli, F. (eds.) SEC 2017. IAICT, vol. 502, pp. 356–370. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58469-0_24
15. Genç, Z.A., Kardaş, S., Kiraz, M.S.: Examination of a new defense mechanism: honeywords. In: Hancke, G., Damiani, E. (eds.) Information Security Theory and Practice. WISTP 2017. Lecture Notes in Computer Science, vol. 10741. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-93524-9_8
16. Wang, D., Cheng, H., Wang, P., Yan, J., Huang, X.: A security analysis of honeywords. In: NDSS (2018)
17. Gutierrez, C.N., Almeshekah, M.H., Bagchi, S., Spafford, E.H.: A hypergame analysis for Ersatzpasswords. In: Janczewski, L.J., Kutyłowski, M. (eds.) SEC 2018. IAICT, vol. 529, pp. 47–61. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99828-2_4
18. Akshima, C.D., Goel, A., Mishra, S., Sanadhya, S. K.: Generation of secure and reliable honeywords, preventing false detection. IEEE Trans. Depend. Secure Comput. **16**(5), 757–769, (2019)
19. Wang, D., Cheng, H., Wang, P., Yan, J., Huang, X.: Targeted online password guessing: An underestimated threat. In: Proceedings of ACM SIGSAC Conference on Computing Communication Security, pp. 1242–1254 (2016)
20. Choi, H., Nam, H., Hur, J.: Password typos resilience in honey encryption. In: Proceedings of IEEE 2017 ICOIN, pp. 594–598 (2017)
21. Karuna, P., Purohit, H., Ganesan, R., Jajodia, S.: Generating hard to comprehend fake documents for defensive cyber deception. IEEE Intell. Syst. **33**(5), 16–25 (2018)