Group-Intersection: Attacking the Password Recovery Service

Alan Lam
California State University, Los Angeles
Los Angeles, USA
alam73@calstatela.edu

Zilong Ye
California State University, Los Angeles
Los Angeles, USA
zilong.ye@calstatela.edu

ABSTRACT

In traditional web services, the password recovery service is simply done by asking users to reset with a new password. It may not be convenient to users, as they need to keep track of all the passwords that have been used. Sometimes, it is hard to generate an easy-to-remember password that meets all the security requirements. There is a recent research proposal that provides an alternative password recovery service by using HoneyGen to generate a number of high-quality honeywords, which are fake passwords similar to the real password. These honeywords are obfuscated with asterisks to help users remember their original password rather than asking users to reset with a new password. In this work, we propose a group intersection attack to crack the user passwords for web services that apply this alternative password recovery service via HoneyGen. We demonstrate the feasibility of the attack through a prototype implementation, and we observe that the attack success rate increases when there exist multiple websites applying the HoneyGen technique for their password recovery service. From our experiment, we conclude that there is a vulnerability with using HoneyGen for the password recovery service, especially when there are multiple websites are using it.

CCS CONCEPTS

• **Security and privacy → Security services**.

KEYWORDS

intersection attack, honeywords

ACM Reference Format:
Alan Lam and Zilong Ye. 2018. Group-Intersection: Attacking the Password Recovery Service. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Passwords are one of the main forms of authentication for a variety of web services (e.g., email services, online banking systems, online shopping systems, etc.). Usually, users sign into their accounts using a username and a password generated by themselves. Sometimes, users may forget the password they used, which prevents them from logging in to their account. To solve this problem, many websites offer a password recovery service through a "Forgot password?" link, which allows users to retrieve their password or simply change their password if they don't remember it. While this is likely to remain the most prevalent solution for forgotten passwords, there is one interesting alternative that was suggested by Dionysiou et al. [1]. They proposed a Honeywords Generation Technique called HoneyGen, and they mentioned that HoneyGen could be used for password recovery because it can generate words that are similar to a user's password. These generated words could be partially obfuscated with asterisks and given to the user to help them recall their original password.

A real-life scenario involving this type of password recovery service would work as follows. A user clicks on the "Forgot password?" link and enters their email along with the last password they remember. The password is given as input to HoneyGen, which returns k passwords that are most similar to the password that the user just entered. Some characters from each of the k passwords are replaced with asterisks and then shown to the user.

Such a password recovery service is considered to be faster and more convenient than simply changing the password that involves email verification and new password creation. However, there is a risk that the password could be leaked or stolen. If the user's real password is included among the fake passwords to increase the chances of helping the user remember their password, then this service is vulnerable to an intersection attack if there are multiple websites using it. For example, given a particular username, an attacker could obtain the obfuscated passwords from one website and compare them with the results obtained from other websites. The attacker could then guess, or even find out, the password depending on the similarity or the overlapping of the obfuscated passwords from different websites. In this study, we refer to this type of attack as the intersection attack.

In this work, we demonstrate the group intersection attack to the password recovery service that is powered by HoneyGen. We show that such a password recovery service may not be secure, especially when there are multiple web services using it for password recovery. In our experiment, the group intersection attack can achieve a success rate of 28%-44% to accurately reveal the user's password when two websites apply such password recovery service. When the number of websites increases (e.g., three websites apply such password recovery service), the success rate of the attack can increase to 58%-66%.

The rest of the paper is organized as follows. Section 2 reviews related works involving password guessing. Section 3 illustrates the design of HoneyGen and the implementation of our group-intersection attack. Section 4 describes our experiment setup and results. Section 5 discusses factors that can influence the success rate of an attack. Section 6 concludes the paper.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

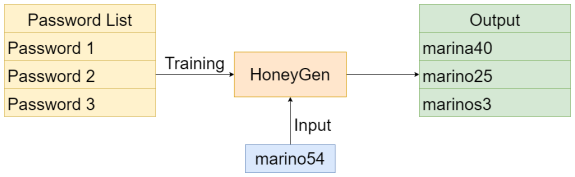

Figure 1: A password dataset is used to train HoneyGen to learn password characteristics and structures. After training, a password can be given as input to output a list of passwords that are similar.

2 RELATED WORK

In recent years, there has been some research efforts on password guessing attacks. One of the most relevant works is by Wang et al. [4]. The authors developed two types of guessing attacks that could distinguish a real password from its honeywords using just one guess. Here, the honeyword generation techniques they have evaluated are based on the work by Juels and Rivest [3]. Their attacks assign probabilities to passwords based on a known probability distribution (e.g., a leaked password dataset) and select the password with the highest probability to be the assumed real password. They achieved a success rate of 29.29%-32.62% with their basic trawling-guessing attack, 34.21%-49.02% with their advanced trawling-guessing attack, and 56.81%-67.98% with their targeted-guessing attack. The study by Wang et al. [4] disproved the idea that using $k-1$ honeywords will leave attackers with a $1/k$ success rate of guessing the real password as proposed by Juels and Rivest [3].

PassGAN is a password-guessing tool that uses a Generative Adversarial Network (GAN) to generate passwords [2]. It uses a deep neural network trained on a password dataset to learn password characteristics and produce an output of password guesses based on those characteristics. By using a deep neural network, PassGAN is able to learn enough information to compete with traditional password guessing tools, such as John the Ripper and HashCat, without requiring any prior knowledge of password structures and password-selection behaviors.

HoneyGen [1] — specifically, HoneyGen's chaffing-with-a-password-model — uses word representation learning to produce honeywords that look similar to a user's real password. It is implemented using a text classification library called FastText, which is used to generate a word embeddings model. There are two steps. First, the model is trained using a dataset of passwords so that the model can learn the word representations. Then, the model is given a password as input to generate a list of passwords in decreasing order of similarity. Figure 1 shows the architecture and the workflow of HoneyGen.

3 GROUP INTERSECTION ATTACKS

3.1 Overview

HoneyGen can be used to provide a password recovery service. As we discussed in the previous section, HoneyGen can offer a few candidate passwords that are similar to the user's attempts. The generated passwords could be similar to the user's original password as specified during the sign-up process, but with a few bits obfuscated by asterisks. These obfuscated passwords can be considered as hints to help the user remember their original password. If HoneyGen is used in multiple websites for a password recovery service, then there is a high risk of leaking user information when the attacker performs a group intersection attack. For instance, the attacker can contact multiple famous websites to obtain multiple obfuscated passwords (hints) for the known username. With all the obtained obfuscated passwords, the attacker can make a reasonable guess about the real password for the username by cross comparing the characters that are displayed and the characters that are obfuscated by asterisks. We will give more details about the group intersection attack in the following parts.

Our group intersection attack formalizes the process of finding passwords with matching length and characters to increase the chances of determining the real password. The main idea is to consult the known username with multiple websites that apply HoneyGen in their password recovery service. Then, the returning hint passwords (i.e., the obfuscated passwords) can be grouped together. After that, we can intersect the characters of the passwords and perform cross comparing to reveal the password. First, passwords are grouped together based on matching length, which are denoted by "correct group". Secondly, passwords that were not put in any group, i.e., they were the only password that had their length, are all put into one group, denoted as a "wrong group". Additionally, any group created in the first step that has a smaller size than the number of websites we consulted is also put into the "wrong group". For example, if three websites were consulted, and there is a group consisting of two passwords, then this group would be put in the "wrong group". Third, for all groups that are not the "wrong group", subgroups are formed that contain passwords with matching characters. For example, a group may contain the following passwords:

*u**ti*5
ku**k*y*
**u*t*55
*u***i55

The subgroups formed would be:

*u**ti*5
**u*t*55
*u***i55

and

ku**k*y*

For two passwords, we count the matching characters in this way: if the character at the same index in both passwords contain either the same character or an asterisk, then it is a matching character. More formally, passwords p and q contain matching characters if

$$p[i] == q[i] \text{ or } p[i] == * \text{ or } q[i] == *$$
$$\text{for } 0 \leq i < length(p).$$

Fourth, subgroups with the same size as the number of websites used (or greater than) are selected for intersection. All asterisks in a password are replaced by the character in the corresponding position in the other passwords. More formally, to intersect passwords p and q do the following:

$$p[i] = q[i]$$
$$\text{for } 0 \leq i < length(p).$$

p is the result of intersection and will be used for guessing the real password.

The above mentioned algorithm can be used to accurately predict the real password, given enough number of websites to consult with. In addition, the algorithm is efficient in terms of the time complexity. As we can see in the pseudocode, the time complexity of the group-intersection attack algorithm is $O(n^2)$, where n is the number of passwords that are retrieved from multiple websites.

```
Function GroupIntersectionAttack(P, w):
    potential_groups ← {}
    wrong_group ← [ ]
    guessing_groups ← [ ]
    forall p ∈ P do
        if length(p) ∈ keys(potential_groups) then
            potential_groups[length(p)].append(p)
        else
            potential_groups[length(p)] ← [p]
        end
    end
    to_delete ← [ ]
    forall (k, v) ∈ potential_groups do
        if length(v) < w then
            wrong_group.append(v)
            to_delete.append(k)
        end
    end
    forall d ∈ to_delete do
        potential_groups.remove(d)
    end
    forall (k, v) ∈ potential_groups do
        guessing_groups.append([v[0]])
        forall p ∈ potential_group[1 :] do
            match_found ← false
            forall g ∈ guessing_groups do
                if AllCharactersMatch(g, p) then
                    match_found ← true
                    g.append(p)
                end
            end
            if match_found ≠ true then
                guessing_groups.append([p])
            end
        end
    end
    passwords ← [ ]
    forall g ∈ guessing_groups do
        if length(g) ≥ w then
            passwords.append(IntersectPasswords(g))
        end
    end
    return passwords
```

Algorithm 1: Group Intersection Attack Pseudocode

4 EXPERIMENT

4.1 Datasets

In practice, websites using HoneyGen for password recovery would train their models using their own database that consists of tons of passwords. For our experiment, we simulated three separate databases of passwords by using three different subsets of the leaked RockYou password dataset (preprocessed and provided by Dionysiou et al. [1]). More specifically, we used 50% of the dataset (including 7,167,879 passwords) to train the model (what we call half-model) for the first website, one-third of the list (including 4,778,587 passwords) to train the model (what we call third-model) for the second website, and one-fourth of the list (3,583,940 passwords) to train the last model (what we call quarter-model) for the third website. The half, third, and quarter lists were generated by getting every other, third, and fourth line of the full RockYou list, respectively. The half list contained even-numbered lines and the quarter list contained odd-numbered lines so that the quarter list was not a subset of the half list.

We used a subset of the leaked Zynga and Dropbox password datasets to populate our database with 100,000 fake users so that we can perform our attacks on these users. 50,000 users had passwords from the Zynga list and 50,000 users had passwords from the Dropbox list. These passwords were randomly chosen and provided by Dionysiou et al. [1].

```
Function AllCharactersMatch(G, p):
    forall g ∈ G do
        if length(g) ≠ length(p) then
            return false
        end
        for i ← 0 to length(p) − 1 do
            if g[i] ≠ " ∗ " and p[i] ≠ " ∗ " and g[i] ≠ p[i]
             then
                return false
            end
        end
    end
    return true
Function IntersectPasswords(G):
    character_list ← [ ]
    for i ← 0 to length(G[0]) − 1 do
        character_list.append(G[0][i])
    end
    forall g ∈ G[1 :] do
        for i ← 0 to length(g) − 1 do
            if character_list[i] = " ∗ " then
                character_list[i] ← g[i]
            end
        end
    end
    password ← ""
    for i ← 0 to length(character_list) − 1 do
        password += character_list[i]
    end
    return password
```

Algorithm 2: Auxiliary Functions Pseudocode

4.2 Setup

We used Flask to create three web applications that served a simple login page where we can enter a username and password to login. In addition to the username and password inputs, each page has a "Forgot your password?" link, a "Let me guess a password!" link, an input for the number of passwords to show (denoted in this paper by k), and an input for the number of websites used (denoted in this paper by w). The "Let me guess a password!" link returns a random username from our database of users. For that username, the passwords are shown after the user enters the value of k and clicks on the "Forgot your password?" link. Entering $k = 1$ returns just the actual password without running HoneyGen, while entering $1 < k \leq 3$ runs HoneyGen to generate $k - 1$ similar passwords that will be returned to help the user remember their actual password. For each password returned, half of its characters are randomly chosen and replaced with asterisks (*). The input for the number of websites used was added to help us record our results, which were updated every time we performed a login.

In the work by Dionysiou et al.'s [1], their proposal was for users to enter the last password they remember as input to HoneyGen and for the output of HoneyGen to be shown to the user. Our modifications include (1) using the real password as input to HoneyGen to streamline the process, and (2) combining the real password with the output of HoneyGen to explore the implications of doing so.

The usernames and passwords, as well as the results of our attacks, were stored in a MongoDB cluster. One collection was used to store usernames and passwords and nine other collections were used to store the number of successes and failures using the following parameters: w1_k1, w1_k2, w1_k3, w2_k1, w2_k2, w2_k3, w3_k1, w3_k2, w3_k3, where wi_kj means using i websites that each returned j passwords.

Three GCP Cloud Functions were created to run HoneyGen. One function used the half-model (trained by half of the RockYou list), another function used the third-model (trained by one-third of the RockYou list), and the last function used the quarter-model (trained by one-fourth of the RockYou list). The functions take a password and k as inputs and outputs a string containing the password and the $k - 1$ generated passwords.

All three FastText models were trained using the following parameters: minCount = 1, minn = 2, epochs = 500, and model = 'skipgram'. These were the same parameters used by Dionysiou et al. [1] to train their FastText models.

4.3 Results

We performed 100 attacks for each of the 9 categories: w1_k1, w1_k2, w1_k3, w2_k1, w2_k2, w2_k3, w3_k1, w3_k2, w3_k3, resulting in a total of 900 attacks. In terms of collecting results for $w = 2$ and $w = 3$, all different combinations of the half-model, third-model and quarter-model were used for the purpose of comprehensive evaluations. For each attack, we define a successful attack to be performing a successful login, i.e., entering a correct password using only one attempt. A failed attack is defined to be performing an unsuccessful login, i.e., entering the incorrect password in the first attempt. Figure 2 provides the comprehensive results of our experiment and Figure 3 shows the number of successful attacks for different values of k and w.

As we can see in those experimental results, there was not a strong correlation between the number of successful attacks and k. As k increased, the number of successes increased for w2 but decreased for w3. The potential effect of k is discussed in the next section, but those cases (both where k could have a positive and negative effect) occurred less than 1% of the time in our experiment. So we attribute the monotonic increase for w2 and monotonic decrease for w3 to random chance.

On the other hand, there was a strong correlation between the number of successful attacks and w. When the number of websites increases, the number of successful attacks increases. The reason behind this is because there is a high chance of revealing more characters in the actual password by performing intersection to decode the asterisks that are marked from different websites. We observe that there was an extremely low success rate (2%-3%) when using only one website because one password with half of the letters hidden had too many asterisks to be easily guessable. But as more websites were used, the success rate increased significantly compared to using only one website. In fact, the attack achieved a success rate of 58%-66% when using three websites. This was the case because using more websites increased the number of real passwords provided, and having more real passwords reduced the number of asterisks after intersection.

If we combine the results of the above two observations, we can conclude that if HoneyGen is only used in one website, even if the number of passwords retrieved from this website increases, the chance of cracking the password is not high. The efficient way to steal the user's password is to involve the results from more websites that uses HoneyGen. The intuition is because different websites have their own databases of passwords for training the model. The more websites that are involved, the more distinctions will be considered in training the model, and thus increasing the chance of revealing the actual password.

5 DISCUSSION

5.1 The Effect of k on Success Rate

There is some potential value in increasing k to achieve a higher success rate. Depending on the size and quality of the password dataset used for training, HoneyGen can produce passwords that are similar enough to the real password, so as to help make a correct guessing of the password easily. For example, consider the following group of passwords:

*usc*t*e**nd*
se****194*
sul***sa*i**
*x*lil***ng
*u**i**bass
*ur**n*b*ss

After performing the group-intersection attack, the result is

*ur*in*bass

which may be difficult to guess. However, by assuming that the other passwords generated are similar to the real password, we can utilize the other passwords to help us achieve a correct guess. In this case, since two of the passwords start with the letter "s", we can assume that the real password starts with an "s", resulting in

sur*in*bass

which then becomes easy to guess.

On the other hand, increasing k may also have a limited value, or sometimes introduce an adverse effect on success rate. Consider the following group of passwords generated by w2_k3:

*a*ahj**n
*e***lan
sa**j**n
*ar*hj***n
*ara***n
sara****n

After performing the group-intersection attack, the result is a subgroup with two passwords:

sarahj**n
saraj**n

In this case, there is more than one password that is potentially correct. This is due to the fact that when the number of k increases, more fake passwords will be generated. As a result, more fake passwords may have the same length as each other. In order to perform the attack, the attacker would first have to choose the correct password, and then guess the remaining asterisks for that password. Incorrectly choosing the correct length of the password would lower the success rate.

5.2 Exploiting Lack of Rate Limitations

In our experiment, we define a successful guess such that only one attempt is allowed for guessing the password correctly. It is expected that the success rate of performing a successful login will increase if more guesses are allowed for each user, because the remaining asterisks can be brute forced after group intersection. For example, if the output is "Buffalo*2", then in the worst case, it will take ten attempts to eventually get the correct password, assuming that the remaining asterisk is a number.

The success rate is also expected to increase if multiple "Forgot password" requests for one user are sent to obtain different combinations of obfuscations for the k passwords. The reason is because the asterisks are randomly generated in each attempt, and the user can perform group intersections of these multiple attempts to reduce the number of remaining asterisks.

5.3 Including the Real Password

In our experiment, we include the real password in the list of passwords returned, with it obfuscated. If we had excluded the real password, then we expect our success rate to be as low as virtually 0%. Our group-intersection attack would not have been applicable because the algorithm requires there to be at least one set of passwords that can be grouped and intersected, which may not exist when the real password is excluded. Also, even though the fake passwords may be similar to the real password, they do not show us exactly what the real password looks like. This is important when one wrong character leads to an unsuccessful login attempt.

Excluding the real password may be more secure, but it may also be less efficient for the users to remember their real passwords. If the fake passwords are not similar enough to the real password, then the user would still not be able to remember their password, especially if their real password is not included. For example, consider the following group of passwords generated by our quarter model:

alejandrasofia366292
apancopilla150993
torrance83

These are the passwords generated by HoneyGen for the input "yankee22". While the structure is similar (a group of letters followed by a group of numbers), it is questionable whether the passwords are similar enough to the real password, especially if they are also partly obfuscated with asterisks.

In order to increase the similarity of the fake passwords, HoneyGen would need to be trained on a sufficiently large dataset of passwords, which may not be possible for services with a small user base. Furthermore, the time it takes to generate the passwords increases with the size of the model, which is based on the size of the training dataset. Our smallest model, the quarter model, took about 40 seconds to generate passwords, while our largest model, the half-model, took about 1.5 minutes to generate passwords, which is probably longer than any user would be willing to wait to get their password.

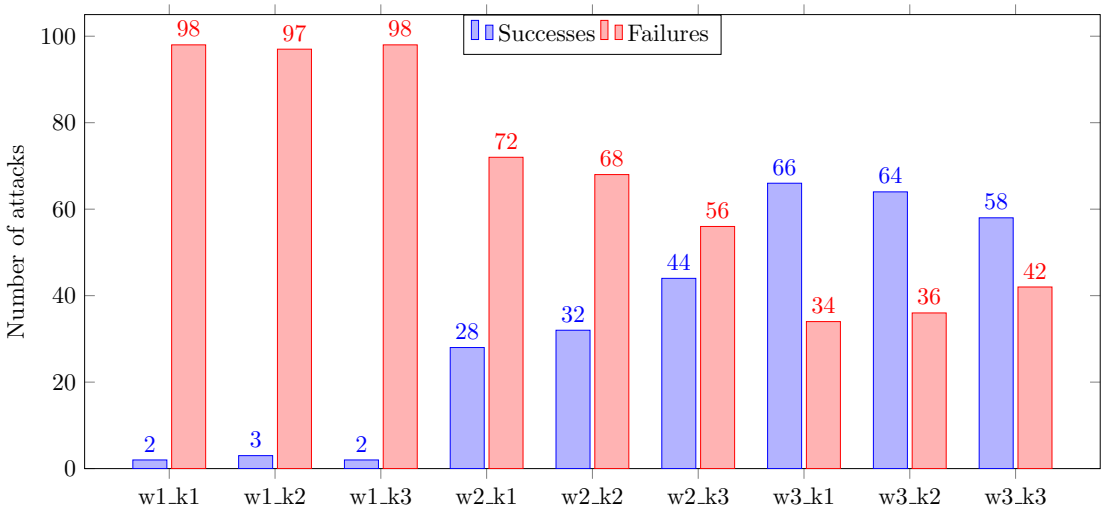

Figure 2: Comprehensive Results of Group-Intersection Attack Using Only 1 Guess

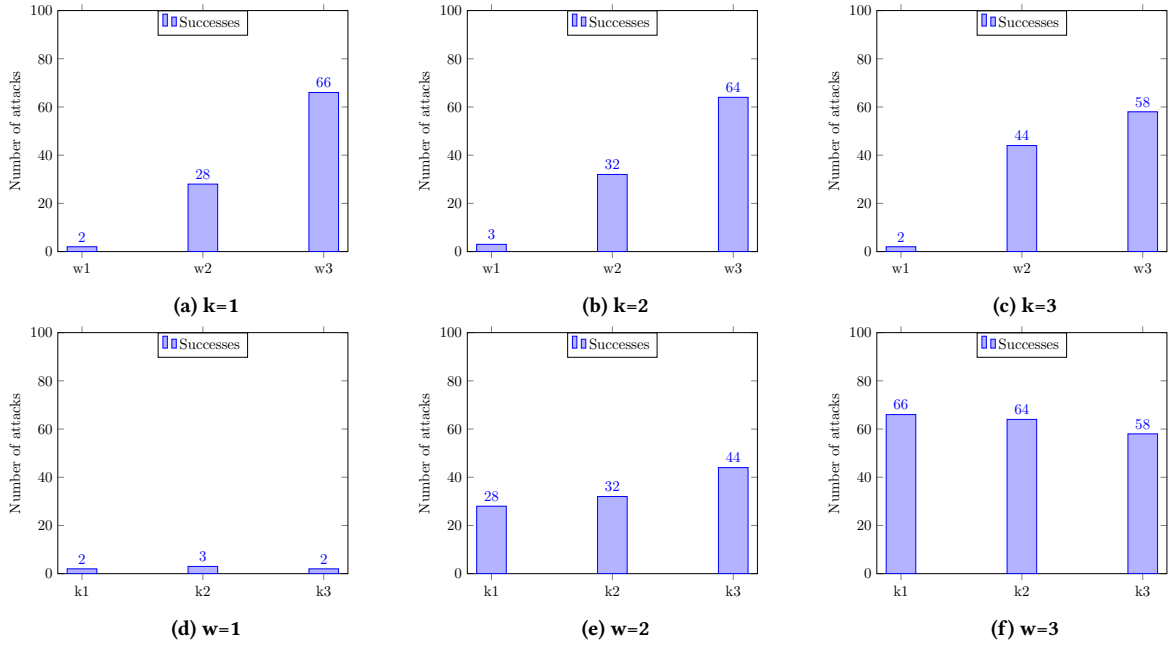

Figure 3: Results of Successful Attacks Using Only 1 Guess

5.4 Random Asterisk Placement

The proposed group-intersection attack algorithm is effective in cracking the passwords if there are multiple websites applying HoneyGen in their password recovery service. However, there are still a few limitations of the proposed attacking technique, which can be further improved. Firstly, under the assumption that the attacker only has one attempt to guess a user's password, there are a few cases where this algorithm will still lead to an unsuccessful attack.

Due to the random placement of asterisks, the result of intersection may be hard to guess. For example, consider the following subgroup:

B*f*alo**
B*f*al**2
Bu***lo*2

The intersection results in "Buffalo*2", which is difficult to know with certainty what the remaining hidden character is.

Secondly, due to the random placement of asterisks, the result of intersection may not always be correct. For example, consider the following passwords:

*a***123
*and*do***
ca*d**2*
dr*0**
n123
ca*de***

After performing all the grouping steps, the result is a subgroup that looks like:

*a***123
ca*d**2*
n123
ca*de***

The intersection step gives us "cande123", but the real password is "candy123" in our dataset. If the "y" had been revealed in any of the first three passwords and the "e" had been hidden in the last password, the correct password would have been produced.

6 CONCLUSION

We implemented a novel alternative idea for password recovery proposed by Dionysiou et al. [1], using their proposed HoneyGen algorithm. On top of that, we developed a novel group intersection attack to show that this alternative password recovery service may not be sufficiently secure if the real password was included in the returned passwords, especially when there were multiple websites applying HoneyGen. Through comprehensive experiment results, we observe that the more websites that used HoneyGen, the more successful our attack was. To the best of our knowledge, this is the first implementation of password recovery service using HoneyGen, and this is also the first implementation of performing password attacking using group intersection. These can be considered as useful guidelines for the industry when deploying the password recovery service in their web service.

REFERENCES

[1] Antreas Dionysiou, Vassilis Vassiliades, and Elias Athanasopoulos. 2021. HoneyGen: Generating Honeywords Using Representation Learning. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ACM, New York, NY, 265–279. https://doi.org/10.1145/3433210.3453092

[2] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Pérez-Cruz. 2019. PassGAN: A Deep Learning Approach for Password Guessing. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings*. 217–237. https://doi.org/10.1007/978-3-030-21568-2_11

[3] Ari Juels and Ronald L. Rivest. 2013. Honeywords: Making Password-Cracking Detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, New York, NY, 145–160. https://doi.org/10.1145/2508859.2516671

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