SAPIENZA
UNIVERSITÀ DI ROMA

Honeyword Guessing with PassFlow: Using Generative Flows to Attack Honeywords

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Corso di Laurea in Informatica

Candidate

Pietro Cau
ID number 1883900

Thesis Advisor

Prof. Emiliano Casalicchio

Academic Year 2022/2023

Ringraziamenti

Voglio ringraziare i miei genitori per il loro lavoro, il loro sostegno e il loro amore, che insieme mi hanno reso possibile di essere qui oggi.
Voglio ringraziare Michele e Chiara, per essermi da esempio ogni giorno, forse anche di più di quanto io lo sia per loro; siete fortissimi.
Voglio poi ringraziare Valeria e Manuel, per essermi stati sempre accanto tra alti e bassi, per essere stati miei sostenitori, consiglieri e tra i migliori amici che avrei mai potuto chiedere.
Voglio inoltre ringraziare, insieme a loro, anche tutti gli altri colleghi con i quali ho avuto il piacere di condividere sudore, sconfitte e vittorie in questo percorso; Grazie per aver reso superabili i momenti difficili, e per aver fatto di questa esperienza un ricordo da custodire.
Voglio ringraziare gli amici di una vita, abbiamo fatto tanta strada insieme, e da ragazzi che eravamo ora siamo diventati uomini, facendoci da spalla a vicenda. Ovunque il destino ci porterà so che io vi porterò con me.
Voglio infine ringraziare tutte le altre innumerevoli persone che in un modo o nell'altro mi hanno aiutato, sostenuto o accompagnato, che sia stato per un anno o per un giorno, grazie.
A tutti voi dedico questo traguardo.

Honeyword Guessing with PassFlow: Using Generative Flows to Attack Honeywords
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: pietrocau000@gmail.com

Abstract

Honeywords are decoy passwords strategically placed alongside genuine passwords to confuse attackers and detect unauthorized access attempts. Building on of previous work done on the analysis of the security of honeywords, we test a new model for honeyword guessing. PassFlow is a flow-based machine learning model originally developed for password guessing, that in this thesis, is used for attacking honeywords. The work of this thesis is, to the best of our knowledge, the first effort to assess the capabilities and potential of Generative Flows (such as PassFlow) as guessing models used to attack honeywords. Through empirical experiments we show how PassFlow's honeyword-guessing performance, while not always up to the task, under certain settings can outperform models used in previous research.

Contents

Chapter 1

Introduction

In today's era, passwords are by far the most widespread method of online user authentication, vastly used for accessing accounts and safeguarding sensitive data. They act as the initial barrier against unauthorized intrusion, guarding information ranging from personal emails and social media accounts to financial details and healthcare records. While passwords have proved to strike an adequate balance between usability and security for use by the general public, this method remains vulnerable to a number of security threats, of which today, one of the most common is the leakage of password files from online services and platforms.

Password leaks occur when confidential password data is unintentionally exposed or compromised, often through security breaches, vulnerabilities, or human error. These leaks can have serious consequences, enabling unauthorized individuals to access sensitive information, perpetrate identity theft, and possibly compromise organizational or in some cases even national security.

Some notable instances of such incidents include Facebook (2021) [13] with 530M account exposed, MySpace (2013) [18], with 360M accounts exposed and Yahoo (2013-2016) [2] with over 3 billion accounts exposed. These are just a few stand-out examples of this very widespread problem. The ITRC 2022 Data Breach Report [5] counts 175 leaks containing passwords just in 2022. What's worse, is that these leaks go undetected for long periods of time, sometimes ranging from months to years, and most often they are only detected when the attackers publish the data online themselves. One notable example is the 2013, 3 billion records leak of Yahoo, only revealed in 2017.

Honeywords [10] emerge as one of the most promising countermeasures to this problem. First envisioned by Juels & Rivest, the honeyword system makes use of automatically generated decoy passwords, that are strategically inserted among the genuine passwords in such a way that when a leak occurs attackers are unable, or at least significantly impeded, in distinguishing the real passwords from the decoys. The main concept of this approach is to complicate the attackers' task by making it difficult to discern a real passwords from a decoy one, so enabling a system to detect an attack and take protective measure.

In Chapter 2 we will give a detailed explanation of the honeyword system.

Some previous work has been done to critically analyze the security of the honeyword system, in particular, Wang et al., in their work "Security analysis of honeywords" [17], which serves as a base for this thesis, have experimented with different types of attacks towards a honeyword protected system. What Wang et al. were able to show is that while still effective in principle, the honeyword system is substantially less safe than claimed in [10], and even unsophisticated methods can be used to guess the genuine passwords among the decoys with a relatively high

level of accuracy.

This thesis proposes to take as example the work of Wang et al. in [17] and to compare a new, more sophisticated attack method to one of the best performers in [17], in order to test it's efficacy.

The method we are going to use is based on a machine learning model called PassFlow [15], devised by Pagnotta et al.

PassFlow is a type of generative model, more specifically categorized as a generative flow type model.

The model works by learning a probability distribution, in this case passwords, and then sampling from that distribution. We however are not interested in the generative capabilities of the model, what we are going to do is use it "in reverse". Generative Flows, compared to other types of generative models based on machine learning methods like VAEs [12] or GANs [7], have the unique property of being invertible in such a way, that instead of using it to sample new elements from a learned distribution, a sample can also be given to it and in return an exact log-probability of that sample in the learned distribution is obtained. This is the intuition behind the use of PassFlow for this thesis.

We discuss more in depth Generative Flows and PassFlow in Chapter 4.

In this thesis we tested PassFlow's guessing capabilities against all four honeyword generation techniques presented in [10], and then compared PassFlow's performance with a model chosen amongst the ones used in [17].

Our findings show that while the other model (Markov) performs generally better, PassFlow has a distinct and significant advantage in some specific situations.

We will discuss the experiment setup and our results in Chapter 5.

Chapter 2

Honeywords

2.1 Basics of the Honeywords System

Honeywords, as proposed by Juels and Rivest in [10], are decoy passwords associated with each user's account alongside the genuine password. These false passwords are designed to confound adversaries who manage to obtain a file of hashed passwords. When an attacker successfully inverts the hash function, distinguishing between the real password and a honeyword becomes a challenge.

Consider a web service with n users, where each user u_i has an associated password p_i stored in a database. In the honeyword system, a Honeyword Generation Technique (HGT) is used to automatically generate, for each user, a number of password-like strings called *honeywords* meant to be decoys for an attacker who gets access to the password database. The honeywords are generated for each user, often on the base of the user's original password p_i, and are then inserted in the password database together with the original password, all linked to u_i's username.

The term "sweetword" refers to a password-like string that can either be the genuine password or a honeyword. Each user will have k distinct associated *sweetwords* (1 genuine password, and $k-1$ honeywords), each of which can be denoted as $sw_{i,j}$.

Def. Sweetword: a *sweetword* is a password-like string that could be either a genuine password or a honeyword.

Def. Sweetword Set: a collection of k password-like strings all associated to the same user, one of which will be the user's genuine password, while the remaining $k-1$ will be honeywords generated by the HGT.

We can denote the set of u_i's sweetwords as SW_i, where

$$SW_i = HW_i \cup \{p_i\}$$

and where HW_i is the set of the $k-1$ honeywords generated for the user u_i.

Once the database is protected by honeywords, an attacker who managed to obtain the password database, and to invert the hash function, will still face the hurdle of having to identify which of the k sweetwords associated with each user is the genuine password, also knowing that any attempt to log in using a honeyword might trigger an alarm.

2.2 The Honeychecker

A fundamental piece of the honeyword system as described in [10] is the "honeychecker". The honeychecker is a separate, hardened auxiliary server that is assumed to be secure. It stores the information necessary to differentiate each user's genuine password from its associated honeywords.

Given a tuple (u, s), containing a username u and a string s, the honeychecker will return "true" if s is the genuine password associated to the username u, and "false" otherwise.

In the background however, the honeychecker will also verify if s is one of the honeywords associated with user u. If it finds that s is indeed a honeyword for that user, then the honeychecker will increment two counters: one keeping track of the number of attempted honeyword logins for the user u, and one keeping track of attempted honeyword logins on the whole system.

If the number of attempted honeyword logins for a given user u exceeds a certain threshold value T_1, the user's account will be suspended until some action to restore the safety of the account is taken (e.g. a password reset). If instead the total number of attempted honeyword logins on the entire system exceeds a certain threshold value T_2 the whole system may be shut down or temporarily disabled. The specific threshold values and the protective measures taken in case of an alarm will vary from system to system.

2.3 Honeyword Generation Techniques

As mentioned, honeywords are generated automatically by the system. This process is of the utmost importance, given that the security of the system will in large part depend on the quality of the honeywords generated. A technique or an algorithm used for generating honeywords is referred to as Honeyword Generation Technique (HGT)[1].

In the original honeyword paper [10], Juels & Rivest propose four different HGTs, the security of whose is also analysed in [17], where Wang et al. assess their effectiveness against honeyword guessing attacks.
Here we describe the four techniques as implemented and discussed in [10] and [17]. These are important because we will later run our experiments against each one of these techniques. Note that while the descriptions here primarily illustrate how these techniques work, the actual code implementation requires significant optimization to ensure reasonable run times on large datasets.

2.3.1 Tweaking By Tail

This technique consists in the random tweaking of the last t characters of the password. Given a parameter t (a value commonly used is $t = 3$), and a genuine password p for user u, we generate $k - 1$ honeywords by substituting randomly each one of the last t characters of p with a random character of the same class. This means that a lowercase letter is substituted by another, randomly chosen, lowercase letter, an uppercase letter by an uppercase letter, a digit by a digit and a special character by a special character.
For example, with $t = 3$, some possible honeywords for the password `Tur1ng@4` could be `Tur1nr!2`, `Tur1no&4` and `Tur1nt?0` .

[1] Also called Honeyword Generation Method (HGM) in the literature

2.3.2 Modeling Syntax

This technique, based on the method described in [6], consists of extracting a syntax from the password, and then creating the $k-1$ honeywords by generating strings with the same syntax. In this context "extracting a syntax" means decomposing the password into tokens representing consecutive sequences of characters of the same class inside the password. Honeywords are then generated by randomly replacing each of those tokens with values having the same length and the same type of the token.
For example the password `james007BOND` has syntax $L_5D_3U_4$ (where L_5 represents a sequence of 5 lowercase letters, D_3 a sequence of 3 digits, and so on), using that syntax as a base, some generated honeywords could for example be `bravo435HOLA`, `nicks120BEAR` or `jones899LAMA`. Note that the values for tokens composed by lowecase or uppercase letters are selected from a dictionary (for this thesis the dictionary used was a list containing the most common 20k English words [11]).

2.3.3 Hybrid

The Hybrid technique aims at combining the strengths of the Tweaking by Tail technique and the Modeling Syntax technique. For a given password p, its syntax is first extracted, then from that syntax, a set of a "seed sweetwords" are created following the same approach we used in the Modeling Syntax technique, (note that this set of seed sweetwords also contains p itself). Then from each of the seed sweetwords, b honeywords are created using the Tweaking by Tail approach. Now we have a selection pool of $a \cdot b$ honeywords from which we randomly select $k-1$ samples.
It's recommended for efficiency that $a \approx b \approx \sqrt{k}$.

2.3.4 Simple Model

The final approach proposed by Juels & Rivest in [10] consists of a relatively straightforward algorithm. Firstly, a list L containing thousands of real passwords and a given number (around 8% of L's total length) of random character sequences of varying length is initialized. Then, in order to generate each honeyword a random password w of length d is selected from L. We then iterate through the characters w_i of w, with i ranging from 1 to d, and we insert them one by one as the characters c_i of the honeyword that is being created, however at each step i, before setting $c_i = w_i$:

- With 10% probability w is replaced with another randomly selected password from L.
- With 40% probability w is replaced with another randomly selected password from L, satisfying the condition that $w_{i-1} = c_{i-1}$.
- With 50% probability, w remains the same.

Note that each time w changes, the value of d must be updated. Once a honeyword is created, it's necessary to check that by pure chance it wasn't generated equal to the original password, and other eligibility checks can be implemented as well based on the system's specifics, some examples are given in [10]. This method differs from all the others we have seen before in the sense that the honeywords generated here do not depend on the user's actual password.

These four are only a few, fundamental HGTs, and will be used for consistency with previous work ([17],[10]), there are however many more HGTs following different approaches and with different characteristics. A more comprehensive overview of common available techniques can be found in [19].

Chapter 3

Honeyword Guessing

In this chapter, the basic idea of honeyword guessing will be introduced, and an overview of the previous relevant work, as well as previous experiments and methods (namely those in [17]) will be presented.

3.1 Honeyword Guessing

The task of honeyword guessing is a relatively straightforward one:
Given a user u_i and his k associated sweetwords, i.e. the set SW_i containing u_i's genuine password p_i and $k-1$ honeywords, the aim of the honeyword-guessing task is to correctly guess which among the k elements of SW_i is the genuine password for u_i.
To accomplish this, a *guesser model* analyzes each sweetword in SW_i and assigns it a probability, then, the element with the highest assigned probability is picked as the model's guess.
The honeyword method aims to make the genuine password indistinguishable from the honeywords. Therefore, a good Honeyword Generation Technique will generate honeywords that will make it harder for our model to assign a probability score discerning the real password from the honeywords. Given a set of honeywords and a password, a Honeyword Generation Technique (HGT) is said to be ϵ-flat, if the highest probability that a guessing model assigns to the actual genuine password is $\approx \frac{1}{k}$, where k is the number of sweetwords for each user. In an ideal scenario then, with a perfectly flat distribution, every sweetword (whether it's the real password or a honeyword) is given by the attacker an equal probability of being the actual password. A perfect HGT then, would make the guesser model perform no better than a random guesser.

3.2 Findings in Previous Work

One of the primary works tackling the task of analysing the security of the honeyword systems is [17]. In that paper, Wang et al. use simple as well as more advanced trawling-guessing attacks with probabilistic password models (specifically Markov Models and PCFGs), and using such models they manage to empirically prove that the four HGTs proposed by Juels & Rivest are considerably less secure than suggested in [10].

The experiments in Section IV. B of [17] are of particular interest, since they employ Markov Models and PCFGs as guesser models, and use them to attack

the tweaking-by-tail HGT. The results of their experiments show how these two methods perform better than the other, previously tested ones, establishing PCFGs and Markov Models as best performers, at least in the confines of the experiments in [17].

The significance of [17] for this thesis is twofold: it serves both as foundational inspiration and as technical guidance. We will implement similarly (or completely adopt) some of its procedures and evaluation methods, in order to test empirically the performance of a new advanced probabilistic password model: PassFlow [15]. However, it is important to note, that aim of this thesis is not that of comparing results with those obtained by Wang et al. in [17], since some small but significant differences in the implementation make the comparison not entirely equitable. Instead, the primary goal of this thesis is that of assessing the performance of PassFlow as a potential guesser model to be used in honeyword guessing, and to do so by analysing it's performance in relation to that of another probabilistic password model (Markov) used with success in [17].

3.3 Evaluation Metrics

The choice of evaluation methods significantly influences the interpretation of an experiment.

It's important to note that the evaluation resulting from an experiment can be interpreted from two different points of view: it can be interpreted from the guessing perspective, as in "how good is guesser model X at distinguishing the genuine password from the honeywords", or it can be interpreted from the HGT perspective, as in: "how good is HGT Y at generating honeywords indistinguishable from the genuine password".

Every experiment involves both a HGT and a guesser model, so these two aspects are two sides of the same coin: the same data can then be interpreted from the attacker's perspective as well as from the defender's perspective. In [17]Wang et al. compare both HGTs and guessing models by looking at the problem from both perspectives. In this thesis however, we will look at the experiments mostly from the attacker's point of view, comparing the performance of different models across various HGTs.

3.3.1 Flatness

The first evaluation metric for HGTs is the flatness evaluation model introduced in [10], according to which a HGT is said to have flatness ϵ with respect to a certain guesser model if the average success rate for guessing the genuine password on the first attempt is ϵ. The ideal case for a defender, is to have a HGT of flatness $\epsilon = 1/k$, where k is the number of sweetwords for each user. In such a case, where the attacker has a chance not much greater than $1/k$ of guessing the actual password on the first try (essentially random guessing), the HGT is said to be "approximately flat" or ϵ-flat. This method of evaluation however, has the downside of only measuring the average success rate for the first attempt, so it will only be meaningful for a honeychecker where $T_1 = 1$.

3.3.2 Flatness Graph

A more comprehensive evaluation metric proposed in [17], as an expansion of the flatness method, is that of the flatness graph. This is an expansion on the flatness

model as it allows for the measurement of the *average case* even for systems that allow more than one online guess (honeycheckers that allow $T_1 > 1$). This method involves plotting on a graph the average success rate at which the attacker guesses the genuine password in the first x attempts.

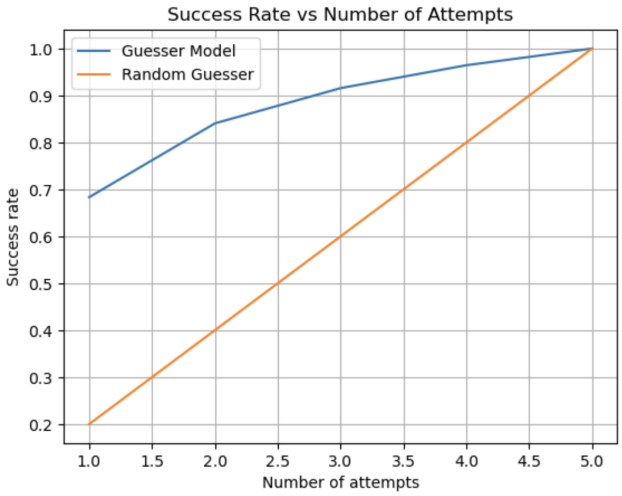


Figure 3.1. Example of flatness graph

Note that in the graph, the value for $x = 1$ (success rate after 1 attempt) is the same measure of flatness proposed by Juels & Rivest.

3.3.3 Success-Number Graph

Figure 3.2. Example of success-number graph

A second evaluation metric proposed in [17] is the success-number graph. The aim of this method is to measure to which extent a HGT generates what Wang et al. describe as "low-hanging fruits": honeywords that are easily distinguishable from the actual password.

From the attacker's point of view, this same metric is used to evaluate the capability of a guesser model to detect such "low-hanging fruits", generated by a certain HGT, and that's the perspective under which we will use it in this thesis. This evaluation technique involves plotting on a graph the total number of successful guesses made by the attacker before making x incorrect guesses.

It is a metric of interest because from the security of the system not only is it important how many correct guesses an attacker can make, but also how many it can make before making a number of mistakes sufficient to set off the honeychecker's system-wide alarm (at T_2 incorrect attempts). So it's fundamental for an attacker, that the guesser model used is capable of making as many correct guesses as possible as early as possible, before making a high number of mistakes.

3.3.4 Graph Interpretation

These graphs provide a more nuanced insight on the performances of different attackers with respect to the single number provided by the flatness metric. A general rule of thumb to interpret these graphs is that the higher a curve is, or the higher its AUC (Area Under the Curve) is, the better a guesser model is performing. Furthermore, especially with the success-number graph, a curve that rises early and fast indicates a model performing well in the first attempts, which is very important for an attacker, in order to avoid triggering the system-wide honeychecker alarm.

Chapter 4

Using PassFlow for Honeyword Guessing

4.1 Generative Models and Normalizing Flows

Generative models are a class of machine learning algorithms designed to learn to approximate the underlying data distribution derived from a set of training data, in order to then generate previously unseen samples resembling the given training data.

The most commonly seen architectures for generative models are VAEs [12] and GANs [8], and more recently Normalizing Flows [16].

Normalizing Flows, also known as Generative Flows, are a type of generative model. They transform simple probability distributions, usually Gaussian, into more complex distributions by applying a series of invertible and differentiable transformations.

Given a continuous random variable z, sampled from a simple base distribution $p(z)$ the idea is that of transforming this simple distribution into a more complex one using some function f that is created from the composition of a sequence of invertible transformations.

$$z \sim p(z)$$
$$x = f(z) = f_k \circ \cdots \circ f_2 \circ f_1(z)$$

Since each f_i is invertible, f will also be invertible.

What makes normalizing flows unique is that while when using GANs we have no latent variable inference and when using VAEs we can only obtain an approximate inference, Normalizing Flows, due to the invertibility of f, are capable of exact log-likelihood evaluation, which in turn allows for exact latent variable inference. This means that not only can we generate a sample from f, but that additionally, if we have a sample x from the data space, we can feed it to the inverse of f to determine the unique prior z that generated x. We are essentially inverting the direction of the flow and mapping a sample from the data space back to the latent space. When we invert the flow, we can use the change of variables formula after each transformation. This renormalizes the probability distribution, allowing us to calculate the exact log-likelihood of a given sample x.

$$p(x) = p(z)\,\Pi_1^k \left|\det\left(\frac{\partial f_i^{-1}}{\partial z_i}\right)\right| = p(z)\left|\det\left(\frac{\partial f^{-1}}{\partial x}\right)\right| \tag{4.1}$$

where $\left|\det\left(\frac{\partial f^{-1}}{\partial x}\right)\right|$ is the magnitude of the determinant of the Jacobian of f^{-1}.

4.2 PassFlow

PassFlow [15], developed by Pagnotta et al., is a type of generative flow model specifically intended for the task of traditional password guessing. It competes with other ML models for password guessing, like PassGAN [9], or with more traditional tools like John the Ripper [4] or HashCat [3]. PassFlow, when trained on a dataset comprising real-life passwords (for example, sourced from a publicly available password leak such as RockYou [1]), learns an invertible function f. This function is designed to transform a basic Gaussian distribution into one that approximates the distribution of the training data. By utilizing this learned distribution, we can generate a vast number of samples that mimic real-life passwords. These generated samples can subsequently be used in password cracking attempts.

4.3 PassFlow for Honeyword Guessing

The main idea of this thesis is to evaluate PassFlow's performance as a potential guesser model an attacker could use against a honeyword system. The concept, then, is that of using PassFlow for guessing honeywords. In order to do that, we train the model as usual, but then instead of using the learned function f to generate samples from the distribution, we use it "in reverse", feeding samples to f^{-1}, and using Equation 4.1 in order to find the probability that the model gives to the sample, this exact probability assessment is only possible thanks to the invertibility of flow-based models. With this setup, all the k sweetwords for a given user can be fed to the model, which will assign a probability to each of them, then, the one with the highest assigned probability is picked as the model's guess for the user's genuine password among the k sweetwords.

Chapter 5

Experimental Setting and Results

5.1 Dataset

For our experiments we selected the RockYou 32M password dataset [1], but we cut by half its size, by random sampling from it ≈16M passwords. This reduced dataset was created in order to facilitate computation and handling of the data while still being able to give significant results. The resulting list was first filtered to remove all passwords containing whitespace or non-ascii characters, then also all passwords with a length less than 2 characters or more than 10 characters were removed. After this filtering the dataset contained ≈ 14.7M passwords, this reduced and filtered dataset will be the base dataset used for our experiments.

The next step was that of splitting the dataset in a training set and a test set. Note that these two sets only contain real passwords.

The training set was used to train the guesser models, while the test set became the set of passwords from which, for each user, we will later generate honeywords using the various Honeyword Generation Techniques. Those honeywords were then used to create a sweetword database.

Inside the sweetword database every entry consists of a username and a sweetword, that could either be a real password or a honeyword. Each user has k associated sweetwords. In our honeyword system, used in the experiments, we use $k = 5$. Usually the k parameter is higher, and the recommendation given in [10] for a live system is $k = 20$. However, since our system doesn't have real-life security purposes, and we are only comparing the relative performances of two guesser models, a lower k is appropriate.

To create the training and test sets for the experiments, we started from the 14.7M password dataset we described above, and divided it following a 90/10 split.

This resulted in a training set containing ≈ 13.3M passwords, and a test set containing ≈ 1.47M passwords.

The training set was the same for all guesser models.

From the test set we created the sweetword databases that we needed for our experiments. In order to do that we first associated each password in the test set to a generated username, then, using a Honeyword Generation Technique, we created $k - 1 = 4$ honeywords for each user, feeding its associated password to the HGT function. The resulting honeywords are all associated to the user, and added to the sweetword database together with the user's genuine password. Since we selected $k = 5$, once done, each sweetword database contained 1.47M×5 = 7.3M sweetwords

(each password in the test set contributes to 4 honeywords and 1 genuine password, so we multiply the 1.47M entries in the test set by $k = 5$).

We repeated the process described above with each of the four Honeyword Generation Techniques described in Section 2.3.

This left us with four sweetword databases, each containing $\approx$ 7.3M sweetwords, or 1.47M sweetword sets (one for each user).

We already set limits on the length and encoding of valid "password-like" strings for our experiments. Another criteria for filtering out invalid passwords is the choice of allowed characters in the passwords. We choose to include as allowed characters:

1. Lowercase Letters: `abcdefghijklmnopqrstuvwxyz`
2. Uppercase Letters: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
3. Digits: `0123456789`
4. Special Characters minus "`" :
   `!"#$%&'()*+,-./:;<=>?@[]_{|}^~\`

These are all the visible ASCII-7 characters minus the backtick "`" (ASCII code 96), the reason for excluding it is that the backtick character creates some issues with PassFlow, since it's used as a special character in its string encoding algorithm. Due to this, in order to make a fair comparison between the two, we filtered out all passwords containing the "`" character.

5.2 HGT Implementation Details and Settings

In order to conduct our experiments we first implemented the components of the honeyword system, so the honeychecker and the four honeyword generation techniques, as described in Section 2.3.

For implementing the Tweaking by Tail technique we used $t = 3$.

In the implementation of the Modeling Syntax technique, we used as a dictionary a list of the 20K most common English words [11].

For the Hybrid technique we used $a = 3$ and $b = 3$, the settings of the techniques used (Tweaking by Tail and Modeling Syntax), remain the same ($t = 3$ and [11] as a word dictionary).

For implementing the Simple Model technique, the password list from which strings are selected contains all the password from the test set, augmented by 8% random character sequences of various lengths.

It is important to note that the code implementing these techniques needs significant optimization with respect to their description here, which serves more to illustrate their working, in order to ensure reasonable run times.

Specifically, in the Tweaking by Tail technique the creation of dictionaries for faster lookup based on character classes (digits, uppercase, lowercase, special) is important.

In the Simple Model technique, specifically for the "40% probability case" (see Section 2.3.4), a lookup dictionary with an outer and inner key was created beforehand, allowing for fast lookup of lists of passwords having a certain character c in a certain index i.

In the Modeling Syntax technique, regular expressions were used for faster syntax extraction and a token dictionary created beforehand was used for faster generation of tokens of different kinds and lengths.

5.3 Markov Guesser Implementation

Firstly, the Markov Model Guesser was implemented. Markov Models, commonly known as Markov Chains, are statistical models particularly suited for representing stochastic processes that operate over a finite set of states. A Markov Model works by observing sequences generated by a process (in this case the real passwords from the training set) and using the frequency of the transition between each pair of states (one character being followed by another), as a way to infer the probability of the transitions between different states. This data is then used in order to then populate a transition probability matrix. This matrix contains the probabilities $p_{i,j}$ for transitioning between each pair of states (i, j). When working with character sequences, or password-like strings as in our case, the states are all the allowed characters, and the sequences are complete password-like strings. The aim is that of creating a simple model capable of capturing the hidden "rules" and tendencies humans follow when creating real passwords.

After training the Markov Model, by "showing it" all the passwords from the training set, we can present it a new, previously unseen password-like string s. The transition probability matrix derived from the training will then be utilized to determine a probability value that will vary based on how much the structure of the sequence of characters in s resembles that of the real passwords shown during training. The sequence's probability is calculated by iterating over each consecutive pair of states and determining the transition probability from the matrix.

Note that, as best practice, the calculations are done as log-probabilities, in order to avoid potential issues related to floating point underflow.

Furthermore, to ensure stable probabilities and avoid zeroes, an "Add-k" smoothing technique is used.

Finally, the implementation of the Markov Model that was chosen, based on performance in the experiments, does not make use of *start-of-sequence* and *end-of-sequence* states. However, this approach implies that the Markov Model will tend to assign higher probabilities to shorter sequences, and given the necessity to compare sequences of different lengths, the probability needs to be normalized for length: to do this we first exponentiate the log-probability, then we apply the n-th root to the result, where n is the length of the sequence, this effectively corresponds to calculating the geometric mean of the transition probabilities, a common method used to normalize for length.

5.4 PassFlow Implementation

For the implementation of PassFlow, the code of the original implementation used is provided by Pagnotta et al. and it's publicly available online [14]. Only minor adjustments and additions to the original code were necessary to adapt it for the use we made of it, and the inner workings of the model remained unchanged.

For the training of the model, since PassFlow makes use of both a training and validation set, the same training set that we used for training the Markov Model was split in a 80/20 fashion (80% training set and 20% validation set).

One of the main advantages of PassFlow over other machine learning models is that PassFlow can produce very good results with a very small number of samples from the training set. In the original PassFlow paper [15] Pagnotta et al. sample only 300K passwords from the training set, and use those to train the model for 400 epochs. In this thesis we trained the model on 500K samples for 100 epochs. Six different versions of PassFlow were trained varying different parameters including

Table 5.1. PassFlow Training Parameters

Parameter	Value
Max Train Size	500000
Max Test Size	2000000
Epochs	100
Batch Size	128
Learning Rate	5×10^{-4}
Weight Decay	5×10^{-5}

the number of samples used for training, the number of samples used for validation, the number of epochs, the batch size, the learning rate and so on. From these configurations, we selected the best performing one, its parameters are reported in Table 5.1. Further increases in the number of training epochs or the number of training samples didn't seem to affect the performance of the model on guessing tasks in a significative manner. The PassFlow code provided by Pagnotta et al. already has most functions necessary for "reversing" the model. We use PassFlow's functions to pad, encode and preprocess strings, then to calculate the necessary log-probability and log-determinant that we then use to compute the sample's probability.

5.5 Guessing Tasks

Each Guesser Model is evaluated against two guessing tasks that allow us to gauge the model's performances using the *flatness graph* and the *success-number graph* evaluation metrics described in Section 3.3.

Both these tasks begin with a sweetword database of the kind described above, containing all the k sweetwords for each user. Then, for each user u_i, with set of sweetwords $SW_i = \{sw_{i,1}, sw_{i,2}, \ldots, sw_{i,k}\}$, we feed SW_i to the guesser model which generates a probability value for each sweetword $p_{i,1}, p_{i,2}, \ldots, p_{i,k}$, these probabilities are then normalized for each user as $p_{i,j} = p_{i,j} / \sum_{t=1}^{k} p_{i,t}$.

This normalization in necessary since as pointed out in [17], if we had two sets of sweetwords SW_1, SW_2 for two separate users, and in SW_1 we have a sweetword with $p = 0.4$ and all the others have $p = 0.35$, while in SW_2 we have a sweetword with $p = 0.30$ and all the others have $p = 0.01$, even if the sweetword in SW_1 has higher probability, the one in SW_2 is more likely to be a correct guess, and should therefore be preferred by the guesser model, that should send to the honeychecker the guess for SW_2 first. That's the purpose of the normalization.

As noted in [17], if the honeychecker allows for $T_1 > 1$, then the probabilities must be re-normalized after each failed attempt.

5.5.1 Flatness Graph Guessing Task

This task is used to create the flatness graph.

Starting with the sweetword database and with the normalized probabilities computed for all sweetwords, we want to compute the average success rate for the correct guessing of the actual password (ranging from 0.0 to 1.0) after x attempts sent to the honeychecker (the possible number of attempts ranges from 1, if the correct guess happens on the first try, to k, if the genuine password is left as the last guess). In order to achieve this, we will iterate through each sweetword set SW_i and send guesses to the honeychecker in order of descending probability, only interrupting

after the genuine password is found in x attempts (where $x \in \{1, 2, \ldots, k\}$). We will repeat this procedure for each user's sweetword set, this will allow us to compute the average success rates after x attempts.

Note that the success rate for $x = k$ will always be 1 and the success rate for $x = 1$ will be the flatness as described in Section 3.3.1.

For this task, our implementation of the honeychecker will not set a limit on T_1 or T_2 to ensure the task runs to completion.

5.5.2 Success-Number Graph Guessing Task

The second task will be used to create the success-number graph. As before, we start with the sweetword database complete with the normalized probabilities for all sweetwords, this time however, instead of attempting guesses until each genuine password is found, we will only send to the honeychecker one guess per user, selected as the sweetword with the highest assigned (normalized) probability. The sweetwords are sent to the honechecker in descending order of normalized probability. For each user then, we will have a binary result (successful guess/failed attempt). This will allow us to plot the cumulative number of successful attempts before the x-th failed attempt. As discussed in Section 3.3.3, this is of interest because it will inform us on how many accounts an attacker could get access to before reaching T_2 failed attempts and triggering the system wide alarm.

For this task, our implementation the honeychecker will have $T_1 = 1$, so a maximum of one online guess per user, and we will not set a limit for T_2 (system-wide alarm, see Section 2.2) in order to let the guessing task run to completion.

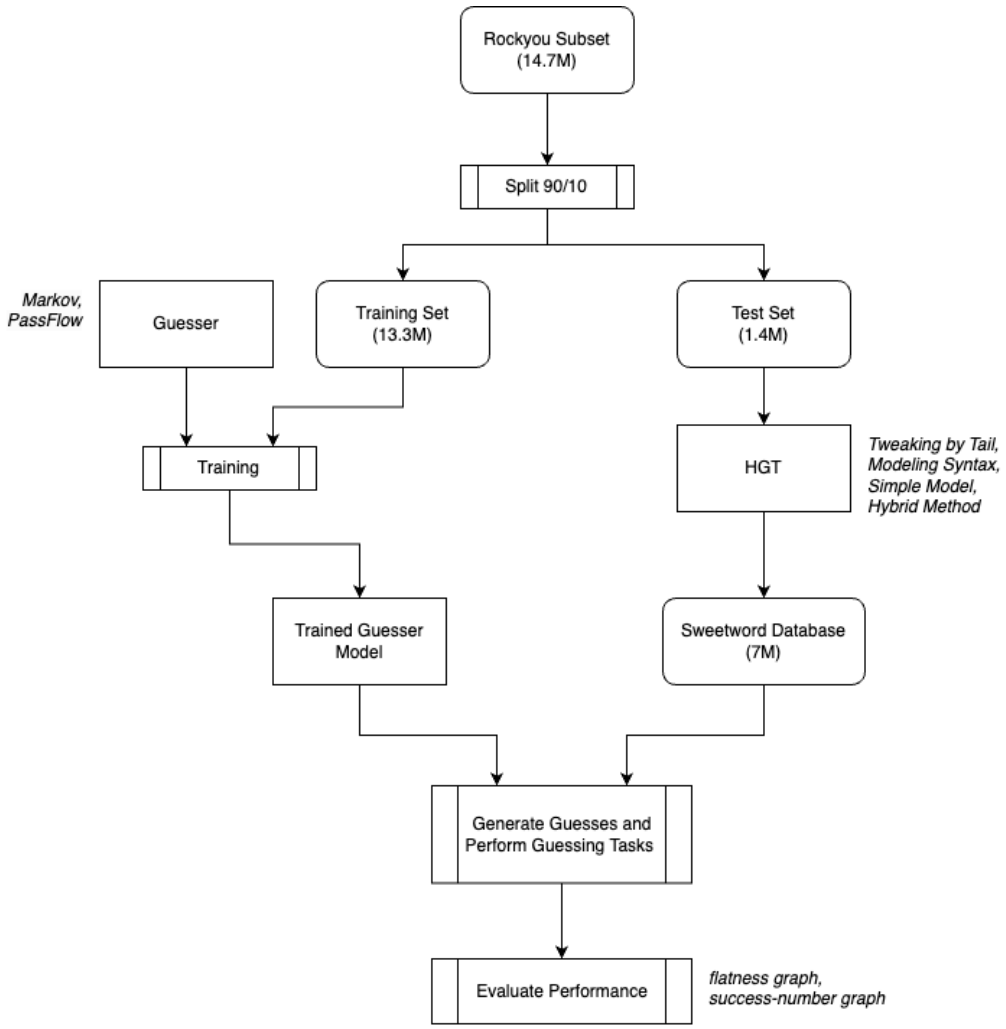


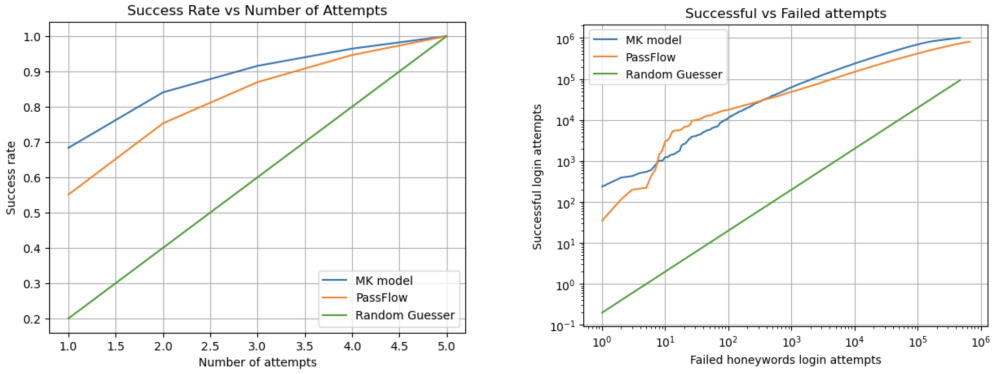
Figure 5.1. Experiment Setup

5.6 Results

In our experiments, we used both guesser models: Markov and PassFlow. Each of them performed both evaluation tasks outlined previously, repeated across the four Honeyword Generation Techniques described in Section 2.3. This approach allowed us to compare their relative performances, strengths and weaknesses under different settings. These are the results from the different experiments.

5.6.1 Tweaking by Tail

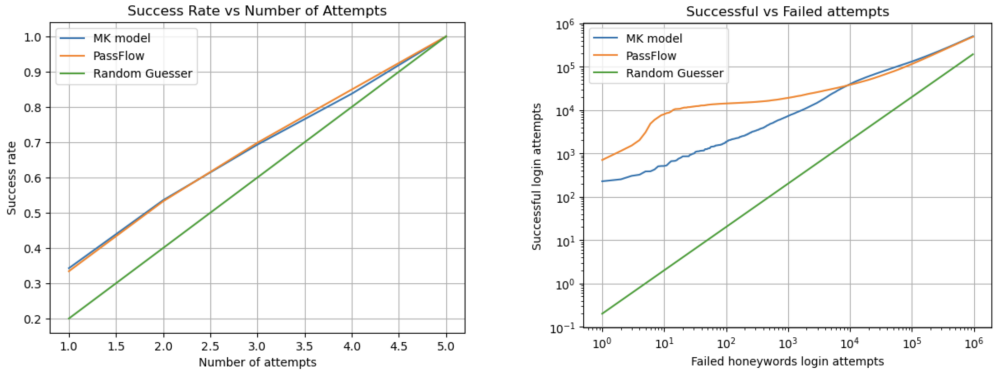
In the Tweaking by Tail experiment the Markov Guesser outperforms the PassFlow Guesser in the success graph. On the success-number graph, their performances are quite similar, with a slight edge for PassFlow in the 10^1 to 10^2 range, and Markov ahead in the rest of the curve.



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

Figure 5.2. Evaluation Graphs for the Tweaking by Tail HGT Experiment

5.6.2 Modeling Syntax



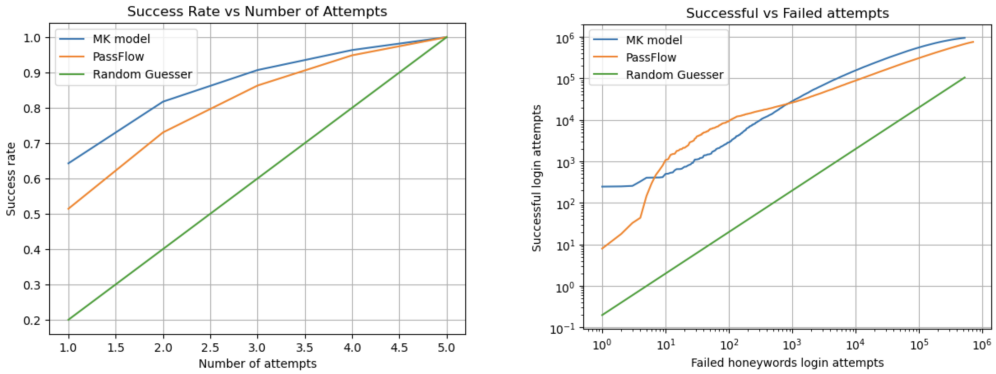
(a) Flatness graph **(b)** Success-number graph

Figure 5.3. Evaluation Graphs for the Modeling Syntax HGT Experiment

In the Modeling Syntax experiment PassFlow displays a strong advantage with respect to Markov. We can see that in the flatness graph, both models perform very similarly, and comparing this graph with the ones from other experiments we can see that they are both having a relatively hard time. In the success-number graph PassFlow starts at an advantage and keeps performing considerably better than Markov until around 10^4 failed attempts, and afterwards they perform similarly. The fact that PassFlow obtains a higher number of successful attempts in that early range is significant, since it would allow an attacker with access to that model to successfully log into a greater number of accounts when attacking a system where the honeychecker has a T_2 value in that range, which is plausible.

5.6.3 Hybrid Technique

In the Hybrid Technique experiment we can see that the Markov Model Guesser performs better than the PassFlow Guesser in the flatness graph, however in the success-number graph PassFlow overtakes Markov from 10^1 up to almost 10^3 failed login attempts. Like in the previous case, the fact that PassFlow obtains a higher

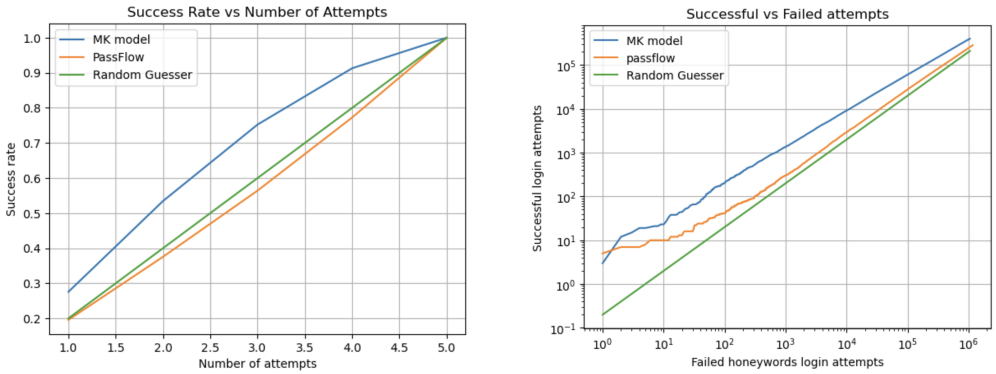


(a) Flatness graph **(b)** Success-number graph

Figure 5.4. Evaluation Graphs for the Hybrid Technique HGT Experiment

number of successful attempts in that range is meaningful, since it would allow an attacker to successfully login into more accounts when attacking a system with a T_2 value in that range.

5.6.4 Simple Model



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

Figure 5.5. Evaluation Graphs for the Simple Model HGT Experiment

The Simple Model technique seems to be the most secure of the four, giving a hard time to both models and especially to the PassFlow Guesser, which in this case performs worse than the random guesser. A possible intuition on why it could be that the two models struggle with this HGT, is that in our implementation of the Simple Model technique, the list L from which we randomly select passwords (see Section 2.3.4), is largely composed by the test set itself, and this results in honeywords that are aggregates of other passwords in the test set, this in turn allows the honeywords to more seamlessly blend in with the genuine passwords, giving more of a challenge to the models.

5.7 Reflection on Results

Our experimental results show that PassFlow can be effectively repurposed as a Honeyword Guessing model, the performance of which varies widely based on the specific HGT that it is working against, and ranges from poor to performing the same as or even significantly better than other models such as Markov's, which qualified themselves as top performers in previous research ([17]).

5.8 Tools

This project was implemented in python, using libraries such as pandas, numpy and matplotlib. Most of the project was run locally on jupyter notebooks. To make use of additional computational power and take advantage of GPU access, the training of the various PassFlow model versions was conducted on Google Colab.

Chapter 6

Conclusions

Honeywords are a promising technology that is very well capable of increasing the security of our password-protected lives. Despite this, they do have some inherent flaws that make them vulnerable to guessing attacks.

PassFlow, and normalizing flows in general, are an extremely capable kind of generative model, that thanks to their unique property of invertibility, and the ability to produce great results from comparatively very small training sets (orders of magnitude less than other kinds of generative models) reach an impressive level of versatility.

In this thesis, we explored these two seemingly disconnected worlds: that of normalizing flows, and that of honeywords, and delved into their technicalities, merging them in a single project that made use of both of these elegant techniques. The work of this thesis is, to the best of our knowledge, the first effort to assess the capabilities and potential of normalizing flows (PassFlow [15]) as attackers in the field of honeyword guessing. This thesis manages to show PassFlow's limitations, potential and applicability in said field, through a series of empirical experiments conducted following the steps of previous research conducted on the same topic ([17]). It's imperative to note that while this study identified particular strengths and vulnerabilities, the rapid evolution of the technology around machine learning and artificial intelligence requires continuous research and adaptation. For instance, recent advancements such as [20] explore the use of Large Language Models for HGTs.

6.1 Future Work

More research is needed to truly test the potential and limitations of PassFlow as a honeyword guesser model and to understand the limitations of the honeyword system. It would be worthwhile to conduct more comprehensive evaluations of such models and seek ways to enhance their performance in this field. A potentially interesting research question is exploring the efficacy of these models from a defensive point of view, essentially investigating their possible utility as HGTs. Overall, the research towards the development of improved and more robust HGTs remains itself an area of interest.

Bibliography

[1] Rockyou. http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2, 2010.

[2] Yahoo raises breach estimate to full 3 billion accounts, by far biggest known. https://fortune.com/2017/10/03/yahoo-breach-mail/, 2016.

[3] Hashcat. https://hashcat.net, 2021.

[4] John the ripper. http://www.openwall.com/john/, 2021.

[5] 2022 data breach report. *Identity Theft Resource Center*, page 20, 2022.

[6] 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 15*, pages 286–302. Springer, 2010.

[7] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1):53–65, 2018.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Commun. ACM*, 63(11):139–144, oct 2020.

[9] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. 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 17*, pages 217–237. Springer, 2019.

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

[11] Josh Kaufman. google-10000-english. https://github.com/first20hours/google-10000-english/blob/master/20k.txt, 2012.

[12] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[13] Hannah Murphy. Facebook faces investigation over leak of data of 530m users. https://www.ft.com/content/0bd7fc8f-cd62-4d57-96c4-bbee8d3deb71, 2021.

[14] Giulio Pagnotta. passflow. https://github.com/pagiux/passflow, 2022.

[15] Giulio Pagnotta, Dorjan Hitaj, Fabio De Gaspari, and Luigi V Mancini. Passflow: guessing passwords with generative flows. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 251–262. IEEE, 2022.

[16] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

[17] Ding Wang, Haibo Cheng, Ping Wang, Jeff Yan, and Xinyi Huang. A security analysis of honeywords. 10 2017.

[18] Elizabeth Weize. 360 million myspace accounts breached. https://www.ft.com/content/0bd7fc8f-cd62-4d57-96c4-bbee8d3deb71, 2013.

[19] Yasser A. Yasser, Ahmed T. Sadiq, and Wasim AlHamdani. A scrutiny of honeyword generation methods: Remarks on strengths and weaknesses points. *Cybernetics and Information Technologies*, 22(2):3–25, 2022.

[20] Fangyi Yu and Miguel Vargas Martin. Targeted honeyword generation with language models, 08 2022.