2019 IEEE Symposium on Security and Privacy

Beyond Credential Stuffing: Password Similarity Models using Neural Networks

Bijeeta Pal*, Tal Daniel†, Rahul Chatterjee*, Thomas Ristenpart*

* Cornell Tech † Technion

***Abstract*—Attackers increasingly use passwords leaked from one website to compromise associated accounts on other websites. Such targeted attacks work because users reuse, or pick similar, passwords for different websites. We recast one of the core technical challenges underlying targeted attacks as the task of modeling similarity of human-chosen passwords. We show how to learn good password similarity models using a compilation of 1.4 billion leaked email, password pairs.**

Using our trained models of password similarity, we exhibit the most damaging targeted attack to date. Simulations indicate that our attack compromises more than 16% of user accounts in less than a thousand guesses, should one of their other passwords be known to the attacker and despite the use of state-of-the-art countermeasures. We show via a case study involving a large university authentication service that the attacks are also effective in practice. We go on to propose the first-ever defense against such targeted attacks, by way of personalized password strength meters (PPSMs). These are password strength meters that can warn users when they are picking passwords that are vulnerable to attacks, including targeted ones that take advantage of the user's previously compromised passwords. We design and build a PPSM that can be compressed to less than 3 MB, making it easy to deploy in order to accurately estimate the strength of a password against all known guessing attacks.

I. Introduction

Despite repeated calls to replace passwords entirely with different authentication mechanisms [1]–[4], human-chosen passwords remain widespread today and will continue for the foreseeable future. This is true despite their notoriety for being easy-to-guess [5], hard-to-remember [6], and difficult-to-type-correctly [7]. The latter two issues tend to encourage reuse of similar passwords across websites: nearly 40% of users reuse their passwords or use slight variations [8].

Password reuse and the rising prevalence of password leaks make targeted guessing attacks an increasingly severe threat. The most prevalent form of targeted attack is so-called credential stuffing, where the attacker simply tries to log into a user's account using password(s) associated to that user found in a leak. The threat is acute: more than five billion leaked accounts were being distributed on the Internet by the end of 2017 [9], [10]; bot-driven credential stuffing attacks account for 90% of the login traffic to some of the world's largest websites [11]; and these attacks represent the largest source of account take over [11].

Website operators, sometimes with the help of third-party services such as HIBP [9], reset user passwords if their usernames or passwords are found in breaches. Such safeguards, which are now actively being recommended by NIST [12], may only prevent credential stuffing — the user can select some small variant of the breached password as their password. A small number of academic works have investigated generalizations of credential stuffing, picking variants of the leaked passwords based on mangling rules [13] or probabilistic context-free grammars (PCFG) [14]. They show such targeted attacks can be damaging, which makes sense given the well-known tendency of users to pick similar passwords [8], even after a password reset [15]. We use *credential tweaking* to refer to attacks that submit variants of a leaked password.

In this work, we investigate credential tweaking attacks from the viewpoint of understanding similarity between human-chosen passwords. We explore data-driven methods for modeling similarity using modern machine learning techniques. This gives rise to a new targeted password guessing attack that outperforms all previous ones, as well as the design of a new kind of password strength meter that includes, in strength estimates, vulnerability to targeted attacks.

Briefly, we treat similarity by learning models that estimate $P(\, w \mid \tilde{w}\,)$, where $\tilde{w}$ is a leaked password from one site and w represents a user's choice of password at another website. We then cast estimating this family of conditional probability distributions (one for each $\tilde{w}$) as a learning task, where we use a compilation of password leaks containing 1.4 billion email, password pairs. We explore various heuristics for identifying passwords used by a single user within the dataset. Ultimately this results in a huge amount of data on password similarity.

We first use this dataset to learn a compact, generative model capturing $P(\, w \mid \tilde{w}\,)$ for all $\tilde{w}$ using sequence-to-sequence (seq2seq) algorithms [16]. These are widely used in the natural language processing literature for language translation and other tasks. Here we treat an input "source" password as $\tilde{w}$ and the model learns how to generate new passwords w in a way that reflects similarity patterns seen in the data. Using seq2seq in this way, however, led to results that do not outperform previous attacks. We therefore took a different approach, training the model to predict the modifications to $\tilde{w}$ needed to transform it into w. While seemingly equivalent, this proved significantly more effective. Intuitively, it focused the model better on learning common transformations found in the data. We call the resulting algorithm *password-to-path* (pass2path), the path denoting the sequence of transformations.

Using the pass2path model, we build a credential tweaking attack that we show via simulation can compromise more than 48% of users' accounts in less than a thousand guesses,

DOI 10.1109/SP.2019.00056

should one of their passwords from another account appear in a breach. The baseline algorithm for credential tweaking attack to guess the leaked passwords only, works about 40% of the time due to password reuse. So, more interesting is how well our attacks work in the case of credential stuffing countermeasures. We perform (separate) simulations for that case, which suggest that 16% of user accounts could be breached with our attack. This is 1.2 times more effective than the previous best targeted attack and 3 times more than the best untargeted attack.

Simulation may not accurately represent efficacy in the real world, and so we evaluate credential tweaking attacks on a real-world system via a collaboration with Cornell University's IT Security Office (ITSO).[1] ITSO deploys credential stuffing countermeasures, as well as other state-of-the-art defenses. Nevertheless, a pass2path-based credential tweaking attack successfully guessed the passwords of over 8.4% of the $15,665$ active Cornell user accounts that appeared in public breaches, in $1,000$ guesses. Our experiments here not only confirm the danger of credential tweaking attacks in practice, but helped us get one step ahead of attackers and identify thousands of potentially vulnerable Cornell accounts for special monitoring. Unfortunately forcing these users to pick new passwords won't necessarily prevent attacks, because they may end up choosing a variant of their previous passwords.

We therefore introduce *personalized password strength meters* (PPSMs). These estimate the strength (non-guessability) of a password considering the user's other (leaked) passwords. We build a PPSM, called vec-ppsm, using neural network-based word embedding techniques [17], [18], which represents another way of modeling password similarity more amenable to deployment as a strength meter than pass2path. Our PPSM can identify passwords unsafe in the face of targeted guessing attacks, and can be used in conjunction with existing password strength meters to give an accurate strength estimate of passwords against all known attacks. In the body we discuss various deployment settings for vec-ppsm.

In summary, our contributions include the following:

- We recast the core technical challenge in targeted guessing attacks as a task of modeling password similarity. This viewpoint allows us to adapt state-of-the-art machine learning tools and apply them to the billions of leaked credentials publicly available. We designed a model pass2path that accurately generates likely user-selected transformations of a given leaked password $\tilde{w}$.
- Using pass2path, we build the most effective targeted password guessing attack to date. It can compromise 16% of user accounts that have been protected against credential stuffing in just $1,000$ guesses.
- We measure targeted attacks in practice for the first time, showing that $1,316$ in-use accounts at Cornell University could have been compromised via our credential tweaking attack, despite credential-stuffing countermeasures.
- We introduce the idea of personalized password strength meters (PPSMs). We build a PPSM using word embedding techniques, and show how it can be used to help prevent credential tweaking attacks.

[1] Our experiment design passed review both by our university IRB as well as by ITSO staff.

II. Background

Password models. Human-chosen passwords have previously been analyzed using tools from natural language processing (NLP). Early examples include using Markov models to help improve dictionary-based cracking tools [19], [20]. Subsequently many data-driven approaches were proposed to learn language models for passwords using password leaks. Weir et al. used probabilistic context-free grammars (PCFGs) [21]. They were later improved by Komanduri et al. in [22] to estimate the distribution of human-chosen passwords. Ma et al. [23] improved upon Markov model-based techniques with some carefully chosen parameters, showing that they outperform PCFG-based models when used to generate a large number of passwords. In 2016, Melicher et al. [24] used recurrent neural networks (RNNs) and Hitaj et al. [25] proposed using deep generative adversarial networks (GAN) to model passwords.

Password guessing attacks. A primary application of password models is to educate brute-force guessing attacks. Such attacks fall into two main categories: offline and online. Offline attacks occur when an attacker obtains cryptographic hashes of some users' passwords and attempts to recover user passwords by guessing-and-checking billions (or even trillions) of passwords. The primary challenge for the attacker is to generate an ordered list of password guesses $w_1, w_2, \ldots$ for which the true user password is likely to appear early. The index of a password w in this list is called the *guess rank* (β) of the password.

An online attack occurs when an attacker uses a login interface or other API to submit password guesses against some account. Because modern authentication systems should lock accounts after a small number of failed attempts (e.g., 10), online attacks are more limited than offline in terms of the number of guesses an attacker can make. The primary challenge, however, is the same. Given a number of guesses or query budget q, the success probability of an attack is what we call the q-success rate, denoted λ_q. For this study, we will focus on the online setting, restricting the query budget to 1,000 or less.

Most password guessing literature focuses on untargeted attacks that generate password guess sequences in a way that is agnostic to the account being attacked. Targeted attacks instead try to take advantage of extra knowledge about the account being attacked. Credential stuffing attacks submit a leaked password for an account to an associated account at another website. These are a growing concern, in large part due to the vast number of password leaks: user accounts, on any given service, are very likely to be associated with at least one account leaked from another source.

Das et al. [13] is the first academic work on targeted attacks exploiting such side information. They showed that around 43% of users reuse the same password across different websites. They also manually developed a rule-based algorithm to guess a user's password with information about one of their other passwords. We refer to this kind of generalization of credential stuffing as credential tweaking because the adversary also submits modifications to the leaked password. Later, Wang et al. [14] constructed a personalized PCFG model to guide credential tweaking based on personal information, including leaked passwords. These targeted attacks outperform untargeted attacks for the small query budgets relevant to online guessing. These existing techniques, however, are not suitable for taking more advantage of the vast amounts of leaked data now available. We will turn to more modern machine learning techniques to do so.

Password strength meters. Password models are also used to develop strength meters [24], [26], which are used most often as a "nudge" to help guide users towards selecting stronger passwords. Password strength estimation was initially done using various statistical methods like Shannon entropy [27]. This approach has various deficiencies, see [28], [29]. More recently, password strength is estimated by calculating a password's guess rank under some password model. Given a password model, guess ranks can be efficiently estimated using the Monte Carlo techniques introduced by Dell and Filippone [30].

III. Preliminaries

Users choose similar and related passwords for different accounts. Therefore, knowledge of one password of a user can be leveraged to guess their other passwords more efficiently. While there might be many latent factors affecting user's choice of passwords, such as their demographics, sensitivity of the website contents, and the website's password policy, previous studies [14] suggest that a user's previous passwords are the most dominant factor in the choice of their other passwords. Therefore to understand the similarity between passwords, we will focus only on a user's passwords, agnostic to the user who is choosing the password and the website for which the password is being chosen for. We consider two passwords to be '*similar*' if they are often chosen together by users.

More formally, let Σ be the set of characters allowed in a password (e.g., all ASCII characters) and ℓ be the maximum allowed length of a password (e.g., 50). Let p denote the probability that a user selects a password $w \in \Sigma^*$ for an account. We denote the support of that distribution by $\mathcal{W}$. We model similarity between two passwords w and $\tilde{w}$ as the conditional probability $P\left(w \,\middle|\, \tilde{w}\right)$ that a user selects the password $w \in \mathcal{W}$ given that another of their password is $\tilde{w} \in \mathcal{W}$. We can extend this definition of similarity to consider multiple of a user's past passwords $\tilde{w}_1, \tilde{w}_2, \ldots$, and compute the probability that w is chosen by the user. In that case, we can model the conditional probability distribution of passwords as $P\left(w \,\middle|\, \tilde{w}_1, \tilde{w}_2, \ldots\right)$.

Prior studies have implicitly attempted to understand similarity of human-chosen passwords using manually curated mangling rules [13] or using probabilistic context-free grammars (PCFG) [14]. In recent years neural networks have proved to be very effective for many natural language tasks, such as understanding word similarity or translating natural language texts from one language to another. We adapt neural networks-based NLP tools for modeling password similarity. Using these tools, we build a more efficient attack and an effective defense against targeted attacks.

Applications of password similarity models. A good password similarity model can be used to perform targeted attacks against a user should an attacker have access to user's passwords from other websites. Such model can also be useful to create defenses against state-of-the-art targeted attacks. A client-side application can warn / prevent users when choosing a password w that can be dangerous for them in the face of targeted attacks, by looking at the similarity between w and various other passwords of the user. Another application of password similarity can be in correcting password typos [7], as typos often comprise of similar passwords.

Though all these applications of password similarity requires learning conditional probability distributions, they need different interfaces from the trained model. For example, to construct a targeted attack, one must be able to efficiently enumerate the conditional probability distribution to generate guesses. However, in case of password strength meter, we don't need the capability of efficient enumeration. As such we target two different kinds of models for password similarity.

The first model is a generative model, built using a sequence-to-sequence-style model previously proposed for language translation [16]. Given a password $\tilde{w}$, this model can be used to enumerate similar passwords in decreasing order of their conditional probability $P\left(w \,\middle|\, \tilde{w}\right)$.

The second model we train is based on word embedding techniques, usually used proposed to understand similarity between words [17]. This model is useful to get a similarity score between a pair of passwords (which is representative of the conditional probability), but does not provide an efficient way to enumerate similar passwords given only one input password. While the generative model can be used to obtain similarity scores too, the embedding model is sufficient to build a strength meter. As we show in Section VII, it is easier to train an embedding model, and it is more efficient to compute similarity score between passwords than our generative model.

Password breach dataset. The dataset we used for learning password similarity is a leaked compilation of various password breaches over time. The dataset was first discovered by 4iQ in the Dark Web [31].[2] The dataset consists of 1.4 billion email-password pairs, with 1.1 billion unique emails

[2]While the leak is publicly available on the Internet, we do not want to further publicize it via including its URL here. Researchers can contact the authors for information.

Property	Values	% of PWs
Length	$3-5$	2
	$6-8$	48
	$9-12$	40
	$13-50$	10
Composition	Lower case only	80
	Upper case only	3
	Letters only	38
	Digits only	8
	Special characters only	< 0.1
	Letters & digits only	55
	Containing at least one letter, one digit and one special char	5

Fig. 1: The distribution of password length and composition in the data after cleaning.

		$\mathsf{D}^{\mathsf{E}}_{\mathsf{full}}$	$\mathsf{D}^{\mathsf{U}}_{\mathsf{full}}$	$\mathsf{D}^{\mathsf{M}}_{\mathsf{full}}$
Number of users (millions)		146	195	174
Number of passwords (millions)		183	210	190
Passwords per user	2	77.0	57.1	74.9
	3	15.5	19.1	16.3
	$\geq$4	7.5	23.8	8.8
Password reuse rate		0.0	30.3	39.7
Edit distance	1	9.4	6.8	9.1
	2	5.2	3.9	5.9
	3	3.3	2.4	3.2
	$\geq$4	82.1	86.9	81.8

Fig. 2: Comparison of the datasets under three account joining techniques: email address ($\mathsf{D}^{\mathsf{E}}_{\mathsf{full}}$), username ($\mathsf{D}^{\mathsf{U}}_{\mathsf{full}}$), and a combination of email and username ($\mathsf{D}^{\mathsf{M}}_{\mathsf{full}}$). We only consider users with at least two leaked passwords. The final set of rows give the fraction of distinct passwords from the same user within the indicated edit distance. Except for the first two rows, all values are percentages (%).

and 463 million unique passwords. The (unknown) curator of the dataset removed duplicate email, password pairs.

Although we do not know the exact leaks that were used to compile this dataset, the folder contained a file called "imported.log" that indicates the presence of all major leaks before December $5, 2017$. The listed leaks include Linkedin, Myspace, Badoo, Yahoo, Twitter, Zoosk, Neopet, etc. Although there was no official way to guarantee the authenticity of the leak, a subset of the passwords have been verified as legitimate by various researchers. (Alarmingly, passwords of two authors appear in the leak.)

Dataset cleaning. Several of the passwords in the dataset were uncracked hash values. To clean the dataset, we removed any string that contains a substring of 20 or more characters long containing only hex characters. This removed 1.5 million passwords. We also removed passwords containing non-ASCII characters and passwords that were longer than 30 characters or shorter than 4 characters. Overall we removed 2.6 million passwords (0.6%), reducing the number of valid passwords to 460.4 million. We also found $4,528$ users were associated to thousands of passwords. These are very unlikely to be passwords of a real user, so we removed these accounts.

The most popular password in the clean dataset (123456) is used by 0.9% of all users. Therefore, the min-entropy of the password distribution is 6.68 bits. The q-success rate λ_q is defined as the expected success probability of an attacker who can make q guesses per account. It is upper-bounded by the sum of the probabilities of the q most probable passwords. For our dataset $\lambda_{10^3} = 0.11$. These values are in-line with what prior work has reported for password distributions [14], [32]. Figure 1 shows statistics about composition and lengths of passwords in our cleaned dataset. More than 88% of passwords were within length 6 and 12, and 80% of passwords contain only lower case letters.

Joining accounts. The leak dataset contains account credentials in the form of email-password pairs, with duplicate pairs removed. We want to merge the accounts to find sets of accounts belonging to an individual user. This will give us the list of passwords corresponding to a user. We explored three heuristics to merge accounts as described below.

- **Email based ($\mathsf{D}^{\mathsf{E}}_{\mathsf{full}}$).** In the first (and the most obvious) strategy, we identify users and join accounts based on the email addresses. We can claim that this strategy will only merge accounts that belong to the same user as in most of the cases, an email address belongs to a unique user. However, because duplicate email-passwords were removed from the dataset, we are not able to observe reuse of passwords by a user in this method. A user can also have multiple emails, which this strategy fails to capture.
- **Username based ($\mathsf{D}^{\mathsf{U}}_{\mathsf{full}}$).** We therefore consider another approach, in particular, using the username field of the email address — the string preceding the domain name and '@' symbol (also called local-part [33]) to further join accounts that might belong to the same user. We merge two emails if their usernames are equal. In this process, we found 30% of passwords are reused by users (see Figure 2), which is slightly below what prior works reported (40%) [8]. Also, as we can see in Figure 2, the distribution of number of passwords per user and the distribution of edit distances between password pairs belonging to a user drastically changed from what we get after email based joining. This, we anticipate, is due to incorrect merging of accounts belonging to different users.
- **Mixed method ($\mathsf{D}^{\mathsf{M}}_{\mathsf{full}}$).** To reduce false merges, we finally consider a two-step approach. We first join accounts based on email addresses. Then, two emails are considered "connected" if the username parts of those emails are equal, and the two password sets associated to those emails have at least one password in common. All connected emails are then considered belonging to a single user. Thus, two emails belonging to a user might not have a direct common password but they might share common passwords with another email. This heuristic led to a password reuse rate of 40%, while keeping the distributions of edit distances and number of passwords per user very similar to what we observed from only email based joining.

Another possible heuristic would be to look at more relaxed policies for matching usernames across accounts. For example, attackers may reasonably be able to conclude that "Alice.Chang@service1.com" and "AliceChang@service2.com" are accounts owned by the same person. We did not explore this heuristic in detail.

After joining the accounts, we only consider users who have at least two leaked passwords in the dataset, because training and testing our targeted attacks, as well as personalized strength meters, requires at least two passwords from a user — one password is used as the target account password and another as the one leaked.

As the username-based merging technique was not accurate, we discard this from further discussion. The rates of password reuse (40%) and substring permutations (18.2%) in $\mathsf{D}^{\mathsf{M}}_{\mathsf{full}}$ (see Figure 2) are in line with prior studies [13], [14]. Though we do not have ground truth that the accounts generated by the mixed method are correct, we believe, given the information we have in the dataset, this is the best approximation of the distribution of passwords chosen by a user.

We split the cleaned email-based dataset $\mathsf{D}^{\mathsf{E}}_{\mathsf{full}}$ into two parts: $\mathsf{D}^{\mathsf{E}}_{\mathsf{tr}}$ (80%) and $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$ (20%). Similarly the mixed-dataset into $\mathsf{D}^{\mathsf{M}}_{\mathsf{tr}}$ (80%) and $\mathsf{D}^{\mathsf{M}}_{\mathsf{ts}}$ (20%). Unless otherwise specified, for all training and validation (during training) we use $\mathsf{D}^{\mathsf{E}}_{\mathsf{tr}}$. This is because the distribution of similar (unique) passwords of a user in $\mathsf{D}^{\mathsf{M}}_{\mathsf{tr}}$ and $\mathsf{D}^{\mathsf{E}}_{\mathsf{tr}}$ were almost identical. Since we only consider similar passwords of a user during training, we do not use $\mathsf{D}^{\mathsf{M}}_{\mathsf{tr}}$ for training separately. For testing, we use random samples from both $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$ and $\mathsf{D}^{\mathsf{M}}_{\mathsf{ts}}$.

IV. Generative models of Password Similarity

In this section we describe how to construct a generative model that estimates the conditional probability distribution $p_{\tilde{w}}$ for an input password $\tilde{w}$, where $p_{\tilde{w}}(w) = P\left(w \,\middle|\, \tilde{w}\right)$. A password can be viewed as a sequence of characters $w = c_1, \ldots, c_l$. Therefore we can model the conditional distribution of a sequence of characters given another as follows,

$$\begin{aligned} P\left(w \,\middle|\, \tilde{w}\right) &= P\left(c_1, \ldots, c_l \,\middle|\, \tilde{c}_1, \ldots, \tilde{c}_{l'}\right) \\ &= \prod_{i=1}^{l} P\left(c_i | \tilde{c}_1, \ldots, \tilde{c}_l, c_1, \ldots, c_{i-1}\right) . \end{aligned} \tag{1}$$

This formulation of password similarity matches very closely to the problem of statistical machine translation (SMT), or more generally learning sequence-to-sequence translation. Sutskever et al. [16] provided a very effective generic framework for training sequence-to-sequence (seq2seq) models, without needing to explicitly specify what the sequences represent. Their seq2seq model uses an encoder-decoder-based architecture. The encoder function maps the input sequence onto a real valued vector $v \in \mathbb{R}^d$ for some hyperparameterized dimension d. The vector succinctly "summarizes" the details of the input sequence. The decoder takes the vector v and outputs a conditional probability distribution of tokens of the output sequence space.

A straw proposal for learning password similarity would be to apply the seq2seq approach directly on passwords as character sequences. We call this model password-to-password or *pass2pass*. However, this technique did not results in improved performance compared to prior work. In Appendix A we give the details of how we trained this model. Below we describe an other (more effective) approach to modeling password similarity using an encoder-decoder based architecture.

Password-to-path model. In pass2pass, we tried to learn the conditional probability of a complete password. As that did not perform well, we decided to learn the modifications a user is likely to apply to their previous password. Password policy of a website might impact choices of some of these modifications. Though, our similarity model can be easily extended to consider website password policies, for simplicity, we will ignore the effect of password policy for now and consider all passwords alike.

We treat a modification to a password as a sequence of transformations defined as follows. A unit transformation $\tau \in \mathcal{T}$ specifies what edit to apply and where, in a password. Therefore, τ is denoted by a triplet of the form (e, c', l), where e denotes an edit to apply, $c' \in \Sigma \cup \{\perp\}$ is character or empty string, and $l \in \mathbb{Z}_\ell$ is a location of the edit in a password. We consider three types of edits: substitution (sub), insertion (ins), and deletion (del). For insertion and substitution edits, c' denotes the character to insert or to substitute with; in case of deletion c' is always the empty string $\perp$. For example, applying a transformation (sub, '!', 8) on the string 'password1' implies substituting the 8^{th} (last) character in the password with the character '!', which will result in the string 'password!'.

Given a pair of passwords $(\tilde{w}, w)$ with edit-distance t, we can find a sequence of transformations $\tau_1, \ldots, \tau_t$ that when applied to $\tilde{w}$ in a cumulative manner will produce w. Such transformations are what we call a *path* $T_{\tilde{w}\to w} \in \mathcal{T}^*$. To compute the path between two passwords, we pick the one that is the shortest, where ties are broken by favoring deletion over insertion over substitution. The transformations in the path are ordered by the location of the edit. (See Appendix C for more details.) For example, the path from 'cats' to 'kates' (edit distance of 2) is: {(sub, 'k', 0), (ins, 'e', 3)}.

In pass2path, we define the conditional probability of a password w given another password $\tilde{w}$ as follows.

$$P\left(w \,\middle|\, \tilde{w}\right) = P\left(T_{\tilde{w}\to w} \,\middle|\, \tilde{w}\right) = \prod_{i=1}^{t} P\left(\tau_i \,\middle|\, \tilde{w}, \tau_1, \ldots, \tau_{i-1}\right) ,$$

where t is the minimum edit distance between two passwords w and $\tilde{w}$, and $T_{\tilde{w}\to w} = \tau_1, \ldots, \tau_t$.

We use an encoder-decoder based model where the output of the decoder function is the probability distribution over transformations in $\mathcal{T}$. With this, we can rewrite the above equation as

$$\begin{aligned} P\left(w \,\middle|\, \tilde{w}\right) &= \prod_{i=1}^{t} P\left(\tau_i \,\middle|\, v_0, \tau_1, \ldots, \tau_{i-1}\right) \\ &= \prod_{i=1}^{t} P\left(\tau_i \,\middle|\, v_{i-1}, \tau_{i-1}\right) , \end{aligned}$$

where v_0 is the output of the encoder, v_{i+1} is the output of the decoder on input v_i and τ_i, and τ_0 is a special beginning-of-path symbol. v_{i-1} contains the information from $\tau_1 \ldots \tau_{i-2}$ and thus replaces $\tau_1 \ldots \tau_{i-2}$ in the final equation. We set up the task of learning this probability model as a supervised learning task, with the training objective being to find the parameter θ that maximizes the log probability of the correct edit paths between password pairs chosen by individual users. Let D be the set of such password pairs, then the training objective is

$$\underset{\theta}{\operatorname{argmax}} \frac{1}{|D|} \sum_{(\tilde{w},w)\in D} \log P\left(T_{\tilde{w}\to w} \,\middle|\, \tilde{w} \;;\; \theta\right)$$

The model architecture of pass2path is similar to encoder-decoder based architecture used for seq2seq instantiated using two recurrent neural networks (RNN) [16]. The encoder and decoder RNNs are trained together. The details of the model architecture are given in Appendix B. Below we will describe some further training details for pass2path. Once trained, the model can be used to generate similar passwords given a leaked password $\tilde{w}$. The details on how to generate the q most probable passwords are given in Section V.

Training pass2path. We train our password models using the data created based on the email dataset ($\mathsf{D}^{\mathsf{E}}_{\mathsf{tr}}$). For each user in the dataset, we compute all pairs of passwords including reordering of the pairs, resulting in 823 million password-pairs.

We represent the passwords as a sequence of key-presses (*key-sequence*) on a US keyboard. For example, 'PASSWORD!' is represented as '⟨c⟩password⟨s⟩1', where ⟨c⟩ and ⟨s⟩ represents caps-lock and shift key on the keyboard. Chatterjee et al. [7] showed that key-sequence representation of passwords are effective for improving password typo correction, and we use it here as it captures capitalization-related transforms better than standard edit distance.

For each pair of passwords in the training set, we generated the minimum path between them using a dynamic programming based algorithm. The algorithm is an extension of the seminal algorithm for calculating minimum edit distance between strings [34]. Given a pair of passwords $\tilde{w}$ and w, we first convert the passwords into key-sequences, and then find a path of transitions that can transform $\tilde{w}$ into w. We describe our algorithm in detail in Appendix D.

A manual sample of a small number of password pairs revealed that a large fraction of them were completely different without any apparent semantic or syntactic similarity. Therefore, we decided to filter the passwords before training based on path length (which is also equal to the key-sequence edit distance between the passwords). Given a cutoff δ, we only consider password pairs with path length at most δ. We begin with $\delta = 2$, finish three epochs of training, and then transfer-learn the network incrementally by adding more pairs with $\delta = 3$ and then $\delta = 4$. We found this way the model converges faster, and attains higher accuracy. Overall the model was trained on 144 million password pairs More details on training pass2path is given in Appendix C.

The pass2path model has 2.4 million parameters and takes 60 megabytes of storage space on disk. It took about two days to train the model with batch size 256 on an Nvidia GTX 1080 GPU and Intel Core i9 processor. The training, however, required less than 2 GB of physical memory.

V. Targeted Guessing Attack using Pass2Path

As discussed in Section II, a primary motivating application for learning password similarity is to understand the danger of targeted guessing attacks, where an adversary generates password guesses educated from a user's other password(s). In this section, we will describe how to generate thousands of guesses from our trained pass2path model to build an effective targeted attack. We will show via simulation that our attack outperforms all prior guessing attacks.

Generating similar passwords. To utilize a password similarity model for a targeted attack, we need to be able to generate, given a leaked password $\tilde{w}$, a list of passwords $w_1, w_2, \ldots, w_q$ in decreasing order of likelihood. Namely, $P\left(w_i \,\middle|\, \tilde{w}\right) \geq P\left(w_j \,\middle|\, \tilde{w}\right)$ for $i < j$. Here q is some number of guesses, a parameter we will concretize below. Generating w_1 is pretty straightforward given our pass2path model. First, convert the input password $\tilde{w}$ into a fixed dimension vector $v_{\tilde{w}} \in \mathbb{R}^d$, and feed it to the decoder along with a special beginning-of-sequence symbol. The decoder outputs a probability distribution over the set of transformations $\mathcal{T}$. Pick the most probable output in each iteration and use that as the input to the next invocation of the decoder until the end-of-sequence symbol is reached. The output sequence of transformation is then applied to the input password to generate a new password.

This procedure however only outputs the most probable password. To generate more than one output, we used breadth-first beam search technique [35]. The beam search algorithm uses a set of size q — called the beam — which, at each iteration of decode, stores the q most probable paths (and network states and probabilities) generated so far. We call a path *complete* if it ends with the end-of-sequence transformation, and *incomplete* otherwise. The beam is initialized with the q most probable transformations output by the decoder on the input of the vector $v_{\tilde{w}}$ and the beginning-of-sequence symbol. Next, for each incomplete path $\tau_1, \ldots, \tau_i$ currently stored in the beam, the decoder is called on the last transformation τ_i of the path, and new paths are computed by appending the transformations to the path. Only the q most probable newly constructed paths are kept in the beam for the next iteration. This step is repeated until a predefined maximum iteration count is reached, or all the paths in the beam are complete.

Beam search is a greedy algorithm and not guaranteed to provide the q most probable paths. However, it is a widely used heuristic alternative for finding the top-q guesses given limited memory and time. To find q paths for a input password, beam search will make at most $q \cdot t$ calls to the decode procedure, where t is a parameter denoting the maximum length of the output path allowed in the model.

As there can be multiple paths that when applied to a password $\tilde{w}$ outputs the same password w, the beam search

with beam width q might not generate q unique passwords. We, therefore, generate $q' \geq q$ passwords and then output the first q unique passwords. In our experiment, we found taking $q' = 2 \cdot q$ is sufficient for finding q unique passwords for more than 99.9% of passwords.

Evaluating targeted guessing attacks. We evaluate a targeted attack based on what fraction of user accounts can be compromised with the knowledge of another of their leaked passwords. We focus on the online attack setting, where an account should be blocked (i.e., login will be disallowed without an out-of-band authentication) after too many failed login attempts. The number of attempts, and therefore maximal guesses available to an attacker before an account is blocked is what we call the query budget q. For evaluation of attacks, we will use a guessing budget of $q \in \{10, 100, 1000\}$. These are typical values used by authentication services.

We use both test datasets: $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$, generated using only the email to identify users, and $\mathsf{D}^{\mathsf{M}}_{\mathsf{ts}}$, generated using the mixed method (see Section III). The first simulates attacking a service that has deployed credential-stuffing countermeasures, e.g., by forcing users to select new passwords should their previous password exist in a leak. Because repeat use of passwords across two accounts is disallowed, we refer to this below as the "without-repeats" setting. The second simulates attacking a service that has not deployed such a countermeasure, and we, therefore, refer to it as the "with-repeats" setting.

Either of the datasets contains millions of users. Some of the targeted attacks that we evaluate are computationally very expensive. We need to pick a smaller but representative sample of the test data. We computed the variances of the rates of using the same and similar passwords by a user for different test set sizes. We found the variance is sufficiently low ($< 0.5\%$) for test sets of size $\geq 10^5$. Therefore, we randomly sample 10^5 random users for each dataset to run our evaluation. For each selected user, we pick two passwords at random without replacement from the multiset of passwords associated to the user — one of them (chosen randomly) is considered as the leaked password $\tilde{w}$ and the other as the target password w.

We compare our attack algorithms against the two existing targeted guessing attacks. Das et al. [13] created a manually curated list of transformations to generate similar passwords. Wang et al. [14] provided multiple attacks based on information about a user, including their demographics, their other passwords, and a combination of these. We will focus on the TarGuess-II attack from [14] which operates with the knowledge of prior passwords only. Wang et al. generously provided an implementation of both the Das et al. algorithm and their TarGuess-II algorithm. The latter requires training from a dataset; see Appendix E for details.

We also compare against two attacks based solely on the empirical distributions of passwords in the training set. The first one is an untargeted attack, which simply guesses (for any leaked password) the q most-used password by users in the training dataset $\mathsf{D}^{\mathsf{E}}_{\mathsf{tr}}$. We found this untargeted empirical model outperforms the state-of-the-art untargeted guessing attack [24] for a small number of guesses, such as $q \leq 10^4$.

The second one is a targeted empirical attack, where for a given leaked password w the attacker outputs the q most popular passwords for the users who also use the password w. While this targeted empirical attack is conceptually straightforward, it would require a prohibitive amount of efficiently accessible memory to implement. We, therefore, simulate the efficacy of this attack by computing the empirical distributions of passwords that occur as leaked in the test data.

We use the leaked password as the first guess for all targeted attacks in all settings. The untargeted empirical attack uses a fixed list of q guesses for all accounts in all settings.

For higher values of the query budget q, some attacks fail to produce q guesses for some leaked passwords. In those cases, we just abort the attack without using up the remaining query budget. In practice, one might try to extend the number of guesses in some ad hoc way, e.g., by adding untargeted guesses. Looking ahead, such an embellishment would not improve the attacks sufficiently to catch up with pass2path.

In Figure 3 we show the different attacks' efficacy in the two settings. We discuss the results for each setting in turn.

Attack efficacy in the without-repeats setting. First we discuss the $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$ results (the left table), where the target password is distinct from the one leaked. Notably, for query budget $q = 10$, the targeted attack based on the empirical distribution performs better than all prior targeted attacks. However, its lack of generalizability hampers its efficacy at higher query budgets. On average only a few associated passwords are in the training dataset for each password, and this attack can only guess passwords observed in the training dataset.

The Das et al. attack doesn't require training data and is the fastest to execute among all targeted attacks we tested. It performed comparatively well, cracking 11% of user accounts in less than a thousand guesses. For many passwords, however, this targeted attack was not able to produce $1,000$ guesses (because it runs out of mangling rules to apply to the leaked password). The Wang et al. [14] algorithm was the state-of-the-art targeted guessing attack before our work. It cracks 13% of user accounts in less than $1,000$ targeted guesses. However, the guess generation is very slow taking more than three days to generate the guesses for all the passwords in just one of our test sets on one thread of a machine with Core i9 CPU and 128 GBs of memory. While in online guessing attacks, computational complexity isn't particularly important (unlike in offline guessing attacks that attempt to crack hashes), we mention it because it proved a significant engineering challenge in our simulations.

Finally, pass2path performed the best among all the attacks, cracking about 13% of passwords in 100 guesses(40% more than what Wang et al. could crack), and 15.8% of passwords in $1,000$ guesses (20% more). The pass2path algorithm is also relatively slow computationally, requiring four hours of computation to evaluate a test set. This was still significantly

Attack Method	$q = 10$	$q = 10^2$	$q = 10^3$
Untargeted-empirical	1.6	2.5	5.2
Targeted-empirical	6.5	7.8	9.0
Das et al. [13]	5.8	9.2	11.0
Wang et al. [14]	6.5	9.3	13.1
Pass2pass	6.9	9.5	10.9
Pass2path	**9.9**	**13.1**	**15.8**

Attack Method	$q = 10$	$q = 10^2$	$q = 10^3$
Untargeted-empirical	0.9	1.9	4.8
Targeted-empirical	42.6	43.4	43.9
Das et al. [13]	42.7	44.8	45.9
Wang et al. [14]	43.2	44.3	47.0
Pass2pass	43.7	45.0	45.8
Pass2path	**44.8**	**46.7**	**48.3**

Fig. 3: Percentage of passwords guessed by various attacks in q guesses on the two test sets generated from (**left**) D^{E}_{ts}, the without-repeats setting, and (**right**) D^{M}_{ts}, the with-repeats setting. In the latter case, a major boost in guessing performance comes from the fact that 40% of target passwords were the same as the one leaked.

faster than Wang et al.

Attack efficacy in the with-repeats setting. We now discuss the results on the test set derived from D^{M}_{ts}. Here about 40% of passwords are reused by users, making them an easy target for credential stuffing. As such, in this case for each attack, we use as the first guess the leaked password. The remaining $q - 1$ guesses are drawn according to the attack technique.

The untargeted empirical attack performs poorly, probably unsurprisingly, as it does not take advantage of the leaked password. The baseline efficacy of other attacks is very high in this context, as 40% accounts are cracked via credential stuffing alone. Our attack pass2path again outperforms all previous algorithms, though here proportionally the improvements are smaller due to high baseline efficacy. For example, the improvement in 1,000 guesses over the best prior attack (Wang et al.) is only a few percentage points. That said, absolutely speaking pass2path compromises nearly half of user accounts appearing in a leak using $1,000$ guesses.

Without credential stuffing defenses, a user's vulnerability to having their account compromised in $1,000$ guesses increases by a factor of ten compared to an untargeted attack, should one of their previous passwords be revealed in a leak.

Attack with multiple leaked passwords. The targeted attack we explained so far assumes access to only one leaked password. But in some cases, attackers will have access to multiple leaked passwords for a target account. In theory, one can train a model similar to pass2path but that uses a sequence of passwords as input instead of only one.

We, however, decided to take a simpler ad hoc approach. We independently generate sorted lists of guesses for each of the input passwords and then merge the lists by picking one from each list in a round robin manner, until the guessing budget is exhausted. To test this attack strategy we picked 10^5 users randomly from without-repeat D^{E}_{ts} dataset, who have at least three or more passwords leaked. For each user, we pick one password randomly as target and the remaining passwords as the leaked passwords. Our heuristic attack approach could compromise 23% of users accounts in 10^3 guesses — a 47% improvement over using just a single leaked password. Future work could explore more advanced models that more carefully utilize multiple leaked passwords.

Attacking any of the accounts. In this experiment, we consider cracking any of a user's accounts given that the attacker knows one of their passwords. In this case, the attacker gets q queries for each account. To test the efficacy of this attack, we sample 10^5 users from D^{E}_{ts} who had more than two leaked passwords and pick one of the passwords as the leaked password and the rest as target passwords. For each account, we generate 10^3 guesses for the leaked password using our pass2path targeted attack and check if any of the target passwords is in the list of guesses. We found 18% of users who lost one of their passwords to an attacker has at least one other account that is susceptible to a targeted attack, even though passwords used in those accounts are different from the one leaked.

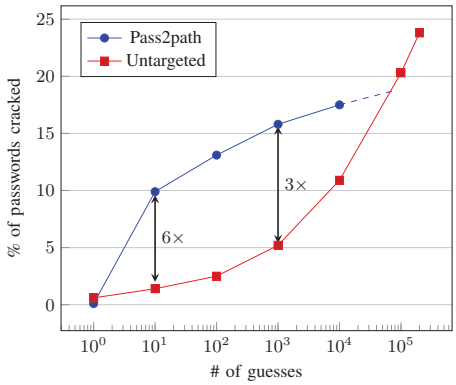

Fig. 4: The relative advantage of targeted attacks and untargeted attacks for large guessing budgets, and crossover point where untargeted attacks becomes more effective than targeted attacks. Due to computational limits, we did not compute points for greater than 10^4 guesses using pass2path, and so the dotted line reflects the observed trend.

Crossover between targeted and untargeted attacks. The targeted attacks are very effective for a small number of guesses ($q \leq 10^3$), compared to an untargeted guessing attack. We observed, however, that as q increases the value of tailoring attacks to the target diminishes. We plot the efficacy of pass2path (targeted) and the untargeted empirical attacks in Figure 4 for different number of guesses. To generate this graph, we sampled 10^5 random users from D^{E}_{ts} and for each user sampled two passwords randomly. Thus we compare the advantage of our targeted attack against the best performing untargeted attack, ignoring the advantage of credential stuffing.

As can be seen, in a guessing budget of $q = 10$, the

pass2path targeted attack can compromise six times more accounts (10% of the test accounts) than what the untargeted attack could (1.6% of the test accounts) . This is also shown by the first column of the left table in Figure 3. This relative advantage however reduces with increased query budget, and if the attacker can make many (say, $q \geq 10^5$) guesses the untargeted attack becomes more advantageous. In an offline attack setting, where an attacker steals the password hash database of a web service and tries to crack the password hashes by making billions of guesses, targeted attacks will be of limited use.

Discussion. In line with prior work, we have used simulations to assess the efficacy of targeted guessing attacks. In practice some additional complications will arise for attackers, such as website-specific rules about password composition. A great advantage of the pass2path model is that it can be adapted to generate passwords matching a website password policy easily. As we show in Section VI, using transfer-learning pass2path model can be retrained only on a subset of the dataset that meets the policy.

Successful password guessing may not alone be sufficient to access modern services that employ two-factor authentication mechanisms. The use of two-factor authentication has increased in recent years, but is still not widespread. Some two-factor authentication systems have vulnerabilities [36], [37] that could be exploited in conjunction with our password guessing attacks.

Finally for the test simulation, we joined accounts using various heuristics but there was no way of determining the number of usernames that were correctly matched. The test dataset also consisted of passwords present in the leaks and thus may be biased towards weaker passwords in general. We wanted to validate the efficiency of the attacks on actual accounts which motivated us to perform real cracking experiments as discussed in the next Section VI.

VI. Targeted Attack Efficacy in Practice

The evaluation of various targeted attacks, in the previous section, was done by comparing their performance against a hold-out test dataset. Here, we turn to evaluating the efficacy of targeted attacks against real accounts, thereby simulating exactly how an attack would proceed in the wild. To do so, we partnered with the IT Security Office of Cornell University (ITSO). We test what fraction of Cornell users' accounts are vulnerable to online guessing attacks. Though untargeted attacks have been analyzed on real-user accounts (e.g., in [5]), to our knowledge, this is the first evaluation of targeted attacks on real user accounts.

In the breach compilation data, we found $19,868$ emails with valid Cornell accounts. From the password change logs that ITSO maintained since 2009, we verified at least $15,776$ accounts definitely have a password selected by the user. Unless otherwise specified, all experiment results below are presented with respect to these $15,776$ accounts. We experimented with three online guessing attacks against these accounts: untargeted empirical, Wang et al., and pass2path.

Attack	$q = 10$	$q = 10^2$	$q = 10^3$
Untargeted-empirical	0	0	0.1
Wang et al. [14]	0.2	0.6	2.6
Pass2path	**3.3**	**6.0**	**8.4**

Fig. 5: The percentage of the $15,776$ active Cornell accounts found in the breach dataset that can be compromised within the indicated number of guesses for three attack approaches.

Cornell uses the **L8C3** password policy, that is, a password must have at least 8 characters from at least three different character classes: upper-case letters, lower-case letters, digits, and symbols. We used transfer learning to retrain pass2path on training data for which the target passwords meet Cornell's password composition requirements. We also adapt untargeted-empirical attacks by considering the most popular passwords that meet the Cornell password composition requirements. However, there is no simple way to tailor the guesses generated by the Wang et al. attack algorithm. More details about the experiment setup are given in Appendix F.

Results. The results of the experiments are summarized in Figure 5. The untargeted empirical attack performed quite poorly: it was able to crack only 0.1% of the target accounts. The Wang et al. attack did a bit better, cracking up to 2.6% of these accounts but as mentioned, its performance is negatively affected by the difficulty of customizing it to Cornell's password requirements.

Pass2path performed the best, cracking over 8.4% of the accounts in less than $1,000$ guesses. Among which, only 22 (0.1%) accounts were cracked using the same password as the one leaked. This is because ITSO uses a third-party service to help prevent credential stuffing attacks.

Recall that our simulations using hold-out data from the breach suggested a success rate of 16%. While it is unclear what explains the gap, we believe it is due to differences in the distribution of passwords at Cornell compared to those found in these breaches. In other words, targeted attacks are slightly overfit to these public data breaches and rates will vary when assessing vulnerability in real systems.

Nevertheless, this experiment shows the vulnerability of accounts to targeted attacks, with $1,374$ active accounts were vulnerable to at least one of the remote guessing attacks. We notified ITSO about these vulnerable accounts and we are working with ITSO to safeguard them.

VII. Defending Against Targeted Attacks

The previous section highlights the danger of targeted guessing attacks even when using state-of-the-art credential-stuffing countermeasures. Can we protect accounts against these attacks? One approach would be for site operators to simulate targeted attack as we did for ITSO, and reset passwords for those vulnerable within the site's threshold of incorrect login attempts. But even doing this, users might in turn pick variants of their passwords that are themselves vulnerable. We would therefore like to additionally have a

method for gauging password strength in the face of both standard and targeted guessing attacks.

Password strength meters. Password strength meters (PSMs) give real-time feedback to users about the strength of their passwords. Historically, password strength was measured using Shannon entropy or heuristic variants [27], but these measures are wildly inaccurate [28], [38]. State-of-the-art strength meters [24], [26] instead infer the strength of a password by estimating its *guess rank* (β) under the best known guessing attack. The guess rank of a password is the number of guesses an attack makes before reaching the password. But the guessing attacks considered so far are user agnostic and therefore rather inaccurate relative to targeted guessing attacks, as we now explain.

Consider the following example situation. A user goes to register a password "atbaub183417a" at some website under the username "alice@gmail.com". The zxcvbn strength meter [26], which is currently in use on the Internet and considered to be a best-of-breed PSM, suggests that the guess rank of the password is 10^{12}, implying that it is a very strong choice. But should alice@gmail.com exist in an easily accessible leak with password "atbaub183417123", our targeted attack guesses it in less than five guesses. Later we will quantify more broadly how often existing PSMs overestimate the strength of passwords easily cracked by pass2path.

Personalized password strength meters. To deal with the above gap, we propose *personalized password strength meters* (PPSM). PPSMs can be used to give users feedback about their passwords during password selection, either as a nudge or as a strict requirement that passwords be of a requisite strength. A PPSM takes as input a target (potential) password w and a set $\mathcal{P}$ of associated passwords of a user, and returns a guess rank under the best-known attack, including targeted attacks. In the future we might extend PPSMs to take into account additional user- and context-specific information, such as username and site domain name.

One approach for estimating the guess rank of a password w would be to return the guess rank under the best known attack, such as the one using pass2path. However, generating guesses using a neural network based model is both computationally expensive and bandwidth intensive (if needed to be sent to a client over the network).Melicher et al. [24] use various clever optimizations to reduce an RNN model to be more efficient. We could potentially adapt these techniques to our RNN-based encoder-decoder architectures. Instead we will explore a fundamentally different approach that will be more efficient.

Traditionally, password strength meters provide a score (approximately reflecting guess rank) that is easy to compute and easy to interpret. For example, zxcvbn [26] gives a score in $\{0, \ldots, 5\}$ and nn-pwmeter [39] gives a score between $[0, 100]$. Therefore, we observe that for most uses, a PPSM need only output a strength score, and not necessarily output a guess rank. Therefore, underlying our PPSM will be a binary classifier $\mathcal{C}$ that takes as input two passwords w and $\tilde{w}$ and outputs 0 if the target password w is probably easily guessed given another password $\tilde{w}$ using a targeted guessing attack, and outputs 1 otherwise. The reason for building a binary classifier is because passwords susceptible to targeted attacks are passwords that can be guessed in few guesses. We use 1,000 as "few", but our framework can be easily used with other values.

To build such a classifier we will use a password similarity measure based on word embedding techniques. The benefit of all this is that we can get by without a (generally more expensive) generative password model, instead using an embedding-based similarity model that quickly outputs a similarity score between two passwords but does not provide an efficient way to enumerate similar passwords from a leaked one.

Looking ahead, we will then show how one can build our PPSM in a way that combines multiple strength estimates, in particular a conventional untargeted strength meter and our similarity score. This will yield a strength meter that accurately measures the strength of a target password w under both targeted and untargeted attacks.

In the rest of this section, we will discuss how we build the classifier $\mathcal{C}$ using a password similarity measure based on word embedding techniques.

Password similarity via embeddings. Similarity between words has been explored in NLP for decades. Recently neural network-based word embedding techniques have been shown to be very effective [17], [18], [40]. Following word embedding models, we define a *password embedding* as a function that maps a password to a d-dimensional vector in $\mathbb{R}^d$. The dimension d is a parameter, often chosen to be relatively small, such as 100 or 200. The embedding is trained so that vectors of similar passwords have low distance (for some measure of distance). Similarity will be context-dependent. In the case of our personalized strength meter two passwords should be considered similar if they are often chosen by the same user. An embedding gives a way to define a score function $s : \mathcal{W} \times \mathcal{W} \mapsto [-1, 1]$ that measures the similarity of two passwords: apply the embedding and then compute the distance between the resulting vectors.

We build password embeddings using the FastText model described in [18]. The FastText model learns similarities by splitting a large corpus of texts into a set of contexts (short sequences of words). Words that often appear in a context together are considered similar. We apply this to passwords by treating passwords chosen by the same user as being in a context together. FastText takes into account n-grams of words and, as such, can produce an embedding that handles words outside of the training set. This will be important for our application.

For our purposes, a password is represented as a union of its n-grams for $n \in [m_{\min}, m_{\max}]$. Let z_w denote the set of n-grams of password w. The beginning and end of each word is clearly denoted by adding two special symbols "⟨" and "⟩" (that are not otherwise in Σ). For example, a password $w = \texttt{qwerty}$ with $m_{\min} = 4$ and $m_{\max} = 5$, will have the following n-grams.

$z_w = \{\langle$qwerty$\rangle$, $\langle$qwe, qwer, wert, erty, rty$\rangle$, $\langle$qwer, qwert, werty, erty$\rangle\}$. (Note, the full password is always included in z_w.) Then, the score function is:

$$s(w, \tilde{w}) = \left(\frac{1}{|z_w|}\sum_{g \in z_w} u_g\right)^{\top} \left(\frac{1}{|z_{\tilde{w}}|}\sum_{g \in z_{\tilde{w}}} u_g\right) \quad (2)$$

Here u_g is the vector embedding of an element $g \in \mathcal{V}$, and $\mathcal{V}$ is the union of all z_w for w's seen in the training data. We denote the embedding of a password $w \in \mathcal{W}$ as v_w, which is computed as $v_w = u_w$ if $w \in \mathcal{V}$ else, $v_w = \frac{1}{|z_w \cap \mathcal{V}|}\sum_{g \in z_w \cap \mathcal{V}} u_g$. If neither the password w nor any of its n-grams is present in $\mathcal{V}$, the embedding of w is set to a random vector in $\mathbb{R}^d$.

Training a password embedding. To train our password embedding FastText model we used the skip-gram approach with negative sampling. We represent each password as a sequence of key-presses, as we did for training pass2path in Section IV. The model requires choosing various hyperparameters.

We set the *dimension* of the vectors to be $d = 100$. This results in much faster training compared to the normally recommended $d = 300$, as well as better performance of the classifier we build using the embeddings (See below.) We set the *sub-sampling* to 10^{-3}. Sub-sampling smooths out the frequency of updates between frequent and infrequent passwords by randomly ignoring some of the frequent passwords. We also only consider passwords that appeared at least 10 times or more in our training dataset. Finally, we set the minimum size of the n-grams to consider $m_{\min} = 1$ to ensure that we can construct an embedding for any password that is not seen during training. We set $m_{\max} = 4$.

Classifying passwords using similarity scores. We want to use the password similarity function s to build a binary classifier $\mathcal{C}$, which takes a pair of passwords and outputs a binary score. To do so, we determine a threshold α: for any password pair whose similarity score is greater than α we assign them a score of 0, and 1 otherwise. We denote a classifier with threshold α as $\mathcal{C}_\alpha$. We call a password pair *vulnerable* if the target password w can be guessed within 10^3 guesses by any of the three targeted attacks known so far — Das et al., Wang et al., and pass2path — given $\tilde{w}$. We want to choose α so that it correctly identifies vulnerable password pairs (by outputting 0 on them), while otherwise maximizing the number of password pairs for which it outputs 1. The latter competing goal stems from usability of the classifier during password registration, which would be hampered by overzealous marking of password pairs as vulnerable when, in fact, they are not.

Relative to some set of password pairs, the recall of $\mathcal{C}_\alpha$ is the fraction of vulnerable password pairs whose similarity falls above the threshold α. The precision is the fraction of password pairs whose similarity is above the threshold α that are actually vulnerable.

We compute the threshold in the following way. We pick randomly 10^5 users from $\mathrm{D}^{\mathsf{E}}_{\mathsf{ts}}$. For each user, we pick two passwords randomly from the set of passwords associated with them without replacement. One of the passwords (chosen arbitrarily) is considered as the target w, and another as the one leaked $\tilde{w}$. For each pair $(w_i, \tilde{w}_i)$, we flag them as vulnerable or not using the three targeted guessing attack as discussed above. This constitute our ground truth. Now we compute the similarity scores $s(w_i, \tilde{w}_i)$ between each pair. For a sequence of thresholds $\alpha \in [0, 1]$ we compute the precision and recall of $\mathcal{C}_\alpha$. The resulting precisions and recalls are shown in Figure 6. As can be seen from the graph, there exists a trade-off between precision and recall. To ensure a recall of 99% — being able to detect 99% of vulnerable password pairs — we pick a threshold of $\alpha = 0.5$. The precision of $\mathcal{C}_{0.5}$ is 60%.

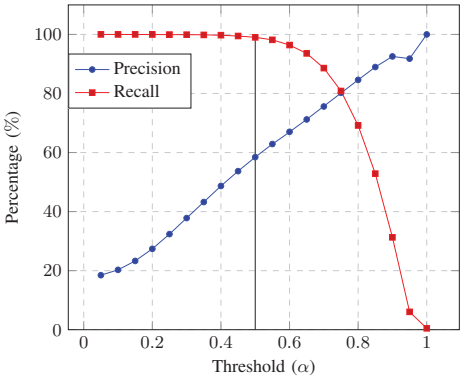

Fig. 6: Precision and recall of our PPSM classifier for different values of the threshold α computed over a random sample of 10^5 password pairs from $\mathrm{D}^{\mathsf{E}}_{\mathsf{ts}}$.

Compressing embedding models. Underlying our password embedding model is a look-up table with keys being a list of frequent passwords and their n-grams, and values being d-dimensional real valued vectors. Therefore, it requires $\mathcal{O}(d \cdot |\mathcal{V}|)$ space to store the embedding. This is more than 1.5 gigabytes for our best performing model. Here we explore two techniques to reduce the size of the model while maintaining good accuracy in identifying weak passwords for targeted guessing.

First, we observed that the quality of the model remains almost the same even after removing all the stored password embedding values $v_w = u_w$ for $w \in \mathcal{V}$. Instead these values can be estimated via $v_w = \frac{1}{|z_w \cap \mathcal{V}|}\sum_{g \in z_w \cap \mathcal{V}} u_g$. Removing the vocabulary of words from the model reduced the size from 1.5 gigabytes to only 195 megabytes, without any noticeable change in the accuracy of the strength estimate.

Next, we used the product quantization (PQ) technique [41] to further compress the vectors, which has been shown to be effective for compressing neural network models [24]. PQ takes a parameter η which determines the compression ratio — the lower the value of η the smaller the model size, but also the worse the accuracy of reconstruction of the input vectors after compression. The reconstruction error of the n-gram vectors in turn impact the score function and the classifier $\mathcal{C}_\alpha$.

We construct the classifier $\mathcal{C}_\alpha$ for different values of η,

η	Size (MB)	Precision (%)	Recall (%)
100	50.0	59.1	99.3
10	5.3	48.5	99.0
5	3.0	41.3	98.6

Fig. 7: Effect on the precision and recall of the classifier $\mathcal{C}_{0.5}$ when compressing the underlying password embedding model using product quantization (PQ) for different values of η.

and compute their precision and recall on a sample of 10^5 password pairs chosen from that many random users from $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$. The results are noted in Figure 7. We can see there is little effect on recall even after compressing the model to 3 MB (with $\eta = 5$). The precision reduced from 59% to 41%, which we believe to be acceptable.

VIII. PPSM Evaluation

We build our PPSM, called vec-ppsm, with two components — one responsible for estimating strength against targeted attacks and another for estimating strength against untargeted attacks. For the former we use our classifier $\mathcal{C}_\alpha$ from Section VII, and for the latter we will use zxcvbn due to its accuracy and performance. Vec-ppsm estimates the strength of a password w in the range 0 (least secure) to 4 (secure), given a set of (leaked) passwords $\mathcal{P}$.

Recall $\mathcal{C}_\alpha$ can classify a password given only one other password. To use it in vec-ppsm, when there can be more than one password in the given password set $\mathcal{P}$, we use a min-strength approach also used by zxcvbn. That is to say, we compute the strength score of w given each $\tilde{w} \in \mathcal{P}$, and output the minimum, $\min_{\tilde{w}\in\mathcal{P}} \mathcal{C}_\alpha(w, \tilde{w})$. If $\mathcal{P}$ is empty it outputs 1.

After this, in order to estimate strength against untargeted attack, vec-ppsm works in conjunction with a conventional, untargeted strength meter, such as zxcvbn: if the targeted strength score of w given $\mathcal{P}$ is 0, vec-ppsm outputs 0, otherwise it outputs the score output by zxcvbn.

Other approaches for comparison. We compare the efficacy of vec-ppsm against two state-of-the-art strength meters: zxcvbn [26] and nn-pwmeter, a neural network based strength meter proposed in [24], [39]. The default behavior of these strength meters is to be agnostic to user's other passwords. However, zxcvbn accepts an optional argument to add site-specific password blacklists. We used this option to simulate a targeted strength meter version of zxcvbn, what we will refer to as tar-zx. It applies zxcvbn, setting the optional argument to the set of (leaked) passwords $\mathcal{P}$. The vanilla use of zxcvbn without such modification is called untar-zx in the following.

Both untar-zx and tar-zx gives a score 0 for passwords that could be guessed in less than a thousand guesses. nn-pwmeter returns a percentage value with 0 representing weak passwords and 100 representing very strong. Thus we divided it into 5 parts and assigned a score of 0 to passwords with score less than 20.

In theory, previous targeted attacks [13], [14] can be used to construct a personalized strength meter, but Das et al. performs poorly as a targeted attack (see Figure 3), and Wang et al.'s guess generation procedure is too slow to generate many guesses (e.g., 10^3) in real time. Thus neither are immediately suitable to be used for constructing a strength meter.

Strength Meter	$q \leq 10$ (%)	$q \leq 10^3$ (%)	Uncracked (%)
tar-zx	40	29	8
untar-zx	12	9	8
nn-pwmeter	35	29	23
vec-ppsm	100	96	20
vec-ppsm (compressed)	99	96	31

Fig. 8: Comparing the percentage of vulnerable passwords that are assigned strength zero ("unsafe" to be used) by the considered strength meters. We used untar-zx as the untargeted strength estimate component in vec-ppsm. The last row to vec-ppsm using the compressed embedding model. The rightmost column gives scores for passwords that are not cracked in 10^3 guesses by any of the targeted attacks.

Evaluating vec-ppsm. We sampled 10^5 users randomly from the test dataset $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$, and for each user, we picked two passwords randomly without replacement as the target password w and the user's other password $\tilde{w}$. Then we try to crack the target passwords using the pass2path targeted attack from Section V. We also compute the strength of the target passwords under all the strength meters under consideration.

In Figure 8, we show the percentage of vulnerable passwords — guessable in less than 10 and $1,000$ guesses by pass2path — that are assigned strength 0 (unsafe to be used) by the various strength meters. Unsurprisingly, all the prior strength meters perform poorly: they assign 70–90% of vulnerable passwords a score of 1 or more (meaning that the passwords are safe against online guessing attacks). However, these passwords are guessable in less than $1,000$ guesses by our targeted attack, and therefore dangerous to use.

The scenario is perhaps even more concerning when we only focus on the passwords that can be guessed in $q = 10$ attempts: more than 60% of them are considered safe by prior strength meters. The best-performing strength meter among the three prior meters is tar-zx, which constructs a blacklist by applying a set of mangling rules to the input password, deletes all occurrence of those blacklisted strings from the target password, and then computes its strength. Even then, tar-zx can only detect 40% of passwords that are severely vulnerable to targeted attack in less than 10 guesses.

Finally, vec-ppsm, can detect 96% of all passwords that can be guessed in $1,000$ guesses. The compressed version of vec-ppsm performs similarly, except with increased rate of false positives. (See the last column in Figure 8.)

We also investigated whether or not vec-ppsm flags the Cornell account passwords found to be vulnerable to one of the three online guessing attacks as per Section VI. We found vec-ppsm assigns score 0 (flags unsafe) to 99.1% of the vulnerable passwords, given the associated leaked passwords. The remaining 0.9% are actually passwords that were vulnerable to the untargeted empirical attack, not a targeted attack. In theory the untargeted attack strength meter underlying vec-

ppsm should have flagged these passwords as weak, but it does not take into account the Cornell password policies. This could be addressed by modifying the untargeted attack strength meter to do so.

Deploying vec-ppsm. There are a few different deployment scenarios where PPSMs will help improve security, which we discuss now.

Perhaps the simplest place to immediately deploy a PPSM is during a password change workflow in which the user provides the old password as well as new password. The user's old password can be used as the "leaked" password, and the PPSM can therefore determine if the new password is sufficiently strong even if the previous password has leaked. Coupled with breach notifications that result in a user changing their password, this prevents credential tweaking attacks entirely. The PPSM can be sent as a JavaScript payload, and the strength check performed on the client side, thereby ensuring that candidate passwords need not be sent to the remote server.

We note that in this case the embedding models are sent to a client's machine, and we must consider what risks this may entail. For example, an attacker might try to discover the set of passwords present in the leak dataset used to train the model. But our compressed embedding model does not contain any information about individual passwords, nor the accounts to which they were associated to in the training data. Instead, it contains n-grams of size 1 to 4. It also does not contain any information about their popularity in the training data. It reveals to an attacker some information about password similarity but it does not provide a generative model sufficient for targeted guessing attacks, unlike some other strength meters (e.g., nn-pwmeter).

The second deployment scenario can be using vec-ppsm during login. We assume the service has access to breached password data (possibly via a third party service). Every time a user successfully logs in, the service checks whether or not the entered password is unsafe according to vec-ppsm, given the leaked passwords associated to that account. If so, it takes necessary steps to warn the user or otherwise safeguard the account. This can all be done on the server side.

Another potential place for use of a PPSM is during initial password registration with an authentication service. However a PPSM requires access to a user's other (leaked) passwords to accurately estimate the strength of the password being selected. Without access to the user's other passwords — leaked or not — vec-ppsm will default to an untargeted strength estimate. In typical web registrations, we would want to send the PPSM as a JavaScript payload to the client side, but then it would require sending leaked passwords to the client as well, which is a security risk. Instead, one could perform PPSM checks on the server side, but then this requires revealing candidate passwords to the server.

Finally, one can use vec-ppsm on a client device, in conjunction with a password manager. The password manager, on behalf of the client, could use a third-party leak checking service (e.g., [9], [42]) to check if any of the client's passwords are leaked. Then vec-ppsm can be used to evaluate the strength of the user's other passwords given those leaked passwords (or all other passwords), similar to how they already provide feedback on untargeted attack strength [43]. Of course, modern password managers provide the option of selecting random passwords, a case that obviates the need for vec-ppsm (or any strength meter). However, many users nevertheless use their own choice of password, and simply store them in password managers. Here vec-ppsm will provide benefit.

We have shown that vec-ppsm can warn users about choosing vulnerable, similar passwords. However, we have not yet addressed the user interface questions regarding how to provide constructive feedback and help guide them towards creating strong passwords. For example, users might get confused in case a password is rejected due to being too similar to a leaked password. How to best inform them about this remains an open question.

Proof-of-concept implementation. We implemented vec-ppsm in Python 3.6. For compressing the embedding models, we used product quantization functionality provided by Facebook's Faiss library [44]. We tested our strength meter on a single thread of a Core i9 processor by randomly sampling 100 password pairs and computing the similarity scores. We record the time to load the model from disk, and the average time taken to compute the similarity score for each pair. The average (across 10 runs) time to load and decompress the model with $\eta = 5$ (size on the disk 3.3 MB) is 0.2 seconds. After loading the model, it takes on an average 0.3 millisecond to compute the similarity score for a pair of passwords, with 99 percentile being within 0.1 millisecond.

IX. Conclusion

In this work, we tackled modeling similarity of human-chosen passwords, and showed how this enables building both damaging targeted guessing attacks and new defenses against them. We explored two approaches to learning password similarity: a generative model based on sequence-to-sequence style learning as used previously for language translation, and a discriminative model based on word embedding techniques.

The generative model enables us to construct a new targeted attack, in which the adversary makes tailored guesses against a user account using knowledge of the user's other password(s). We show our best performing attack can, in less than a thousand guesses, compromise 8.4% of active user accounts at Cornell University, for which a previous password was leaked. This attack outperforms the best previous attack by 3.2x.

Though targeted attacks are already a widespread threat, there are few defenses available against them. The only ones we are aware of stop credential stuffing, but do not prevent our credential tweaking attacks. We therefore proposed personalized password strength meters (PPSMs), which can be used to warn against choosing passwords that are easily guessable under different attacks, including targeted attacks. We built a prototype of a PPSM, called vec-ppsm, using word embedding techniques, and showed how it can be used to mitigate attacks.

ACKNOWLEDGMENTS

We thank Tyler Kell, Dan Villanti, and Jerry Shipman for helping us with the experiment with Cornell ITSO. We also thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF grants CNS-1514163 and CNS-1564102, and United States Army Research Office (ARO) grant W911NF-16-1-0145.

APPENDIX

A. Pass2pass model.

A straw proposal for learning password similarity would be to apply the seq2seq approach directly on passwords as character sequences. We call this model password-to-password or *pass2pass*. The encode function maps the input password $\tilde{w}$ onto a real valued vector $v_0 \in \mathbb{R}^d$. The decoder function takes a vector $v \in \mathbb{R}^d$ and a character $c \in \Sigma \cup \{\text{`}\langle\text{'}, \text{`}\rangle\text{'}\}$ and outputs a probability distribution over the characters in $\Sigma \cup \{\text{`}\langle\text{'}, \text{`}\rangle\text{'}\}$ and another vector $v' \in \mathbb{R}^d$, which is fed to next iteration of the decoder. Every password is enclosed by a special beginning-of-sequence symbol $c_0 =$ '$\langle$' and an end-of-sequence symbol '$\rangle$'. Therefore, in this model, we can rewrite Equation (1) as follows, where v_i is the output of the decoder on input v_{i-1} and c_{i-1}.

$$P\left(w \mid \tilde{w}\right) = P\left(c_1, \ldots, c_l \mid v_0\right) = \prod_{i=1}^{l} P\left(c_i \mid v_{i-1}, c_{i-1}\right)$$

We used the default neural network architecture and the hyperparameters used in seq2seq [16] to train several variants of pass2pass. We evaluated them by testing the trained models' efficacies as targeted guessing attacks on a validation set, distinct from the eventual test set we report on later. (See Section V for details on how to use a seq2seq based model for generating targeted guesses.) Initially we tried training pass2pass with password pairs from $\mathsf{D}^{\mathsf{E}}_{\mathsf{full}}$. This performed horribly. We then restricted attention to password pairs from the same user that were within edit distance two of each other. This sped up training and seemed to help the model focus on easier-to-learn similarities. We also tried edit distance three, but this performed worse than edit distance two. In the end, the efficacy of our best-performing pass2pass model remained underwhelming. The targeted attack based on the best performing pass2pass model was only able to guess 11% of users' passwords in 1,000 guesses, while the state-of-the-art approach from [14] can guess 13.1%. (See Figure 3)

Our intuition for this poor performance is that passwords have a much larger support (we have around 200 million distinct passwords) and does not follow any predefined rules (save those set by password policies) unlike natural languages. Restrictions by edit distance helped learning, but miss many important similarities that ideally an attack would capture. We needed a different approach.

B. Model architecture of pass2path

Pass2path uses two recurrent neural networks (RNN) — one for the encoder function and another for the decoder — which are trained together, similar to what is used for seq2seq learning [16]. RNNs were designed to recognize patterns in sequential data, with varied sequence-lengths. However, vanilla RNN suffer from vanishing and exploding gradient problems. A variant of RNN, called long short-term memory (LSTM) [45] was shown to be effective in avoiding vanishing and exploding gradient problems [46]. We used LSTM blocks wrapped in residual cells. Residual cells, first used for image recognition using deep neural networks [47], "short-circuit" the input of a layer to the output, bypassing internal calculations. (See Figure 9(b).) We found pass2path achieves slightly better accuracy at noticeably lower training time with residual cells than without it. We implemented pass2path in TensorFlow [48] using the building blocks provided by the library. Each LSTM cell in the model has three hidden layers, each layer with 128 hidden units.

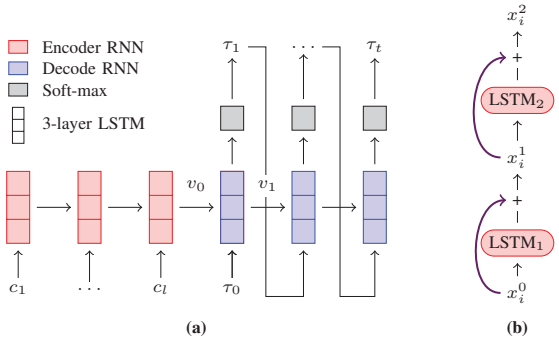

Fig. 9: **(a)** Diagram of encoder-decoder architecture for pass2path learning. **(b)** A 2-layer LSTM cells with residual connection. Here c_i's are characters of input passwords, τ_i's are transitions, and x_i^j's are internal states in neural networks.

A diagram of the neural network architecture of pass2path is given in Figure 9(a). The encoder processes each character in a password sequentially. A character is first represented as a one-hot-vector of dimension $|\Sigma|$, and embedded onto a real-valued vector of dimension 200. The embedded character is then fed to a LSTM cell with three hidden layers, each of dimension 128.A LSTM outputs two vectors, the first one is ignored for the encoder, and the second one, called *state*, is fed to the next LSTM cell, along with the next character of the password. Let the output of the encoder be v_0, obtained after applying it on the whole input character sequence.

The vector v_0 is then fed to the decoder with a special beginning-of-sequence symbol τ_0. The architecture of the decoder is identical to the encoder except that we consider the first output of the LSTM layer, which is projected to a vector of size $|\mathcal{T}|$. The softmax function is applied to the projected vector to convert it into a probability distribution over $\mathcal{T}$. The most probable transformation is considered the output and used as the input to the next iteration of the decoder, except if the output is a special "end-of-sequence" symbol. The sequence of transformation outputs can then be applied to the input password to obtain another password.

C. Training pass2path model

We trained pass2path using an encoder-decoder based neural network architecture. Here we give the details of our training approach, in particular, how we initialize the network prior to training, and the hyperparameters.

We used the initialization techniques proposed in [16]: the embedding layers are initialized with uniform random values from $[-\sqrt{3}, \sqrt{3}]$, while the rest of the network is initialized with uniform values in $[-r, r]$ where $r = \sqrt{6/(n_j + n_{j+1})}$ and n_j is the dimension of the input to the j^{th} layer of the neural network. For training, we used stochastic gradient descent (SGD) using the Adam's optimizer [49] to minimize the cross-entropy loss [50] between the predicted output of the network and the expected output. Minimizing the cross-entropy loss (with softmax) ensures learning the conditional probability of the output given the input.

During initial phase of training, we used *teacher-forcing* to train the model faster, by feeding the expected output transformation as the decoder's input, instead of the predicted character. As the training progresses we start feeding the actual predicted character as input. We did not use *attention* mechanism [51] (a common technique used in seq2seq language translation models) as passwords are relatively small in size compared to sentences in language translation.

We need to pick a number of hyper parameters for our architecture. Excluding those below, we used those suggested in [16]. Below are the ones we set to different values for better performance.

(1) *Learning rate.* The learning rate parameter controls the effect of loss gradient on the change of the model parameters. We used a fixed learning rate of 0.0003 for the training.

(2) *Dropout rate.* The dropout rate controls removal of neural network units (*neurons*) randomly during training, which is useful to prevent overfitting [52]. We tried dropout rates of 0.3 and 0.4, the latter worked best.

(3) *Layers.* Each RNN cell consists of multiple hidden layers. For language model a typical number of hidden layers is $n \in \{3, 4\}$ [46]. We found pass2path with three hidden layers performs better than four layers. Each layer consists of 128 hidden units.

(4) *Epochs.* The number of epochs determines how many times the training procedure iterates over the training dataset. We found three epochs were enough. More significantly increased training time with negligible benefit.

D. Generating paths from password pairs

For every training input pair, we first compute the minimum edit distance using a dynamic programming (DP) approach, and then backtrack the DP solution to find the actual transitions that result in the calculated edit distance. We calculate the distance matrix according to the formula.

$$D(i,j) = \min \begin{cases} D(i-1, j-1) & \text{if } w(i) = \tilde{w}(j) \text{ [copy]} \\ D(i-1, j-1) + 1 & \text{if } w(i) \neq \tilde{w}(j) \text{ [substitute]} \\ D(i-1, j) + 1 & \text{[insert]} \\ D(i, j-1) + 1 & \text{[delete]} \end{cases}$$

GenPath$(w, \tilde{w})$:

$n \leftarrow |w| + 1;\ m \leftarrow |\tilde{w}| + 1\ ;\ D \leftarrow \infty^{n \times m}\ ;\ T \leftarrow \emptyset^{n \times m}$
$D_{0,0} \leftarrow 0$
for $i = 1$ to n do $D_{i,0} \leftarrow i;\ T_{i,0} \leftarrow (i-1, 0)$
for $j = 1$ to m do $D_{0,j} \leftarrow j;\ T_{0,j} \leftarrow (0, j-1)$
for $i = 1$ to n do
  for $j = 1$ to m
    $e \leftarrow [0, 0, 0]$ /* 0 : del, 1 : ins, 2 : sub */
    $e_0 \leftarrow D_{i-1,j} + 1;\ e_1 \leftarrow D_{i,j-1} + 1$
    if $w_{i-1} \neq \tilde{w}_{j-1}$; then $e_2 \leftarrow D_{i-1,j-1} + 1$
    else $e_2 \leftarrow D_{i-1,j-1}$
    $k \leftarrow \text{argmin}\, e$
    $D_{i,j} \leftarrow e_k$
    if $k = 0$ then $T_{i,j} \leftarrow (i-1, j)$ /* del */
    else if $k = 1$ then $T_{i,j} \leftarrow (i, j-1)$ /* ins */
    else $T_{i,j} \leftarrow (i-1, j-1)$ /* sub, copy */
/* Back-trace the dynamic programming solution computed above. */
$i \leftarrow n;\ j \leftarrow m;\ c \leftarrow T_{i,j};\ P \leftarrow \emptyset$
while $c \neq \emptyset$
  parse c as (i_c, j_c)
  if $i_c = i - 1$ and $j_c = j - 1$ then
    if $w_{i_c} \neq \tilde{w}_{j_c}$ then P.append(sub, $\tilde{w}_{j_c}, i_c$)
    $i \leftarrow i - 1;\ j \leftarrow j - 1$ then
  if $i_c = i$ and $j_c = j - 1$ then
    P.append(ins, $\tilde{w}_{j_c}, i_c$)
    $j \leftarrow j - 1$
  else
    P.append(del, $\perp, i_c$)
    $i \leftarrow i - 1$
  $c \leftarrow T_{i_c, j_c}$
return P

Fig. 10: GenPath algorithm for generating sequence of transformations from a pair of passwords.

The pseudocode for generating a path is given in Figure 10.

E. Re-training Wang et al. for our dataset

The Wang et al. algorithm specified in [14] needs to be trained before it is used to generate guesses. The code shared by Wang et al. requires four files for training. All the required files were generated using our training data $D^{\mathsf{E}}_{\mathsf{ts}}$.

1) **PCFG data.** This data file contains three main sections: passwords containing only digits (D), only letters (L), and only special characters(S). Each section had 9 subsections of passwords of length one to nine, namely $D_1, \ldots, D_9$, $L_1, \ldots, L_9$, and $S_1, \ldots, S_9$. Each subsection contains the 10 most popular passwords matching that structure and their probability of occurrence in that subsection. For example, $P(\text{`1234'}) = C(\text{`1234'})/C(D_4)$, where $C(w)$ denotes the probability of w, or the sum of probabilities of passwords in a section, if w denotes a section.

2) **Markov data.** This file contains 4-grams that contain only letters, only digits, and only special characters, along with their probability. For example,

$$P\left(\text{`4'} \,\middle|\, \text{`123'}\right) = C(\text{`1234'})/C(\text{`123'})$$

3) **Reverse Markov data.** This file is similar to the previous Markov data file, except the n-grams are computed after reversing the passwords.

4) **Top password data.** The file contains the top 10^4 passwords and their probabilities from the training data.

F. Targeted password cracking experiment in practice

Cornell University has a large-scale authentication system including nearly half a million accounts. Students, faculty, and staff all receive accounts, and alumni accounts are by policy not deactivated after students graduate. The accounts are enrolled in a single-sign on (SSO) system giving access to email and other systems, and as such are frequently targeted by attackers.

ITSO currently has a number of mechanisms in place to make remote guessing attacks difficult. (1) They require passwords to consist of at least eight characters, and they must cover at least three character classes, namely upper-case letters, lower-case letters, digits, or symbols. Additionally the system will reject passwords containing one or more words from a non-public dictionary of user identifiers, first and last names, common passwords, and common English words. For example, "passw0Rd" is an allowed password, but "password" is not. (2) ITSO subscribes to a service that notifies them of accounts that have appeared in breaches. Such accounts have their password hashes scrambled should the account's email-password pair match that in the breach, and users have to choose a new password to regain access. (3) Users are not allowed to select their old password when choosing a new one. Thus, ITSO uses many state-of-the-art protections, including credential stuffing countermeasures. Of course, the authentication system has been evolving over time, and some accounts had passwords chosen under different policies than the current ones. We will account for this nuance below.

Experimental setup. We worked with ITSO to safely perform an experimental measurement of the vulnerability of Cornell accounts to our targeted attacks. In particular, ITSO uses Kerberos to store authentication information including password hashes. We arranged to receive access, intermediated and overseen by ITSO staff, to a test server that had a mirror of the Kerberos authentication database. This ensured we did not interfere with production authentication pipelines. The research team never received direct access to this server, rather all access was run by ITSO staff who ensured no sensitive information is revealed. We treated password hashes as particularly sensitive, and discuss how we safely handled them more below.

We started by determining which accounts appear in the breach dataset described in Section III. There were $19,868$ accounts found in the breach dataset, meaning that we had at least one previously breached password for them. The age of the last password reset of these potentially vulnerable accounts skew older, with the median age being 5 years, but there were accounts with passwords reset as late as in 2018.

ITSO maintains a log of password change events since 2009 that records whether the password was changed by the user or the password was scrambled. Among the $19,868$ accounts that appeared in the breach, $3,106$ accounts have their last password changed prior to 2009, and therefore, we are unsure what fraction of those accounts contain scrambled passwords. From the remaining $16,762$ accounts with recent password changes (after Jan 1, 2009), 986 accounts definitely have their password scrambled at the time of the study; $15,776$ accounts had passwords chosen by the user. We will call these as *active accounts*.

To perform the experiment more quickly than simulating online attacks directly, we carefully exported salted hashes of $19,868$ account passwords to a secured research machine and run an offline hash cracking with a limited number guesses per account. The machine is only accessible from the Cornell network, has no listening services beyond SSH, requires second factor authentication to login, and disk volumes are encrypted. We also further protect the Kerberos hashes, by rehashing them using $4,096$ iterations of SHA-256 with a 128-bit per hash salt and a strong pepper (acting as a secret key). Once disclosure proceedings are finished with ITSO, we will delete both the pepper (cryptographically erasing these hashes) and the entire hash database.

Cracking was performed on this secure server, a Core i5 machine with 8GB of RAM. It required five days to test all the 45 million passwords against all $15,776$ accounts.

We compared three guessing procedures — the baseline untargeted empirical attack, the Wang et al., and a variant of pass2path. The untargeted-empirical attack first guesses the leaked passwords, followed by the most probable passwords in our breach dataset that meet Cornell password policy. For each of the $15,776$ accounts, we generated $1,000$ guesses based on Wang et al. attack algorithm. Unfortunately, we could not tailor them to the Cornell password policy because the obvious approach — rejection sampling — was prohibitively slow.

We used transfer learning to build a variant of the pass2path model customized to the Cornell character class requirements for passwords. We did not attempt to customize based on the common-words dictionary check, as this dictionary is not publicly available. We took the trained pass2path model and retrained three more epochs on a smaller dataset containing only those leaked password pairs where the target password satisfied Cornell's character class requirements.

For the Wang et al. and customized pass2path, we generated $1,000$ guesses, with the leaked password being the first guess, for each of the target accounts. For accounts that had more than one leaked password, we used the round-robin method as described in Section V. On average, over the targeted accounts, 77% of the Wang et al. and 15% of the customized pass2path guesses do not meet Cornell's requirements.

Results. Overall, the three targeted attacks cumulatively could compromise $1,688$ accounts (including 314 inactive accounts that had their most recent password change prior to 2009). We notified ITSO about these vulnerable accounts. ITSO is

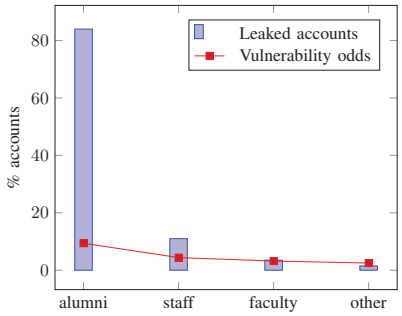

Fig. 11: Distribution of different types of accounts that are found in the breach dataset, and their odds to be vulnerable to one of the three online guessing attacks. The "other" category includes current students, contract workers, and affiliates.

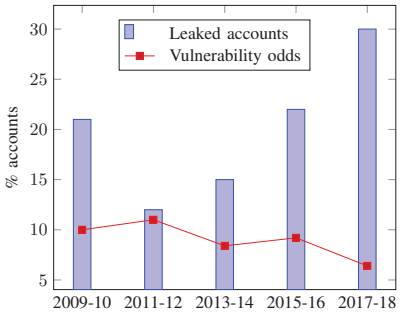

Fig. 12: Distribution of active accounts in the leaked dataset based on the year of their last password reset. The red line shows their odds of being vulnerable to targeted online guessing attacks.

working on a multi-pronged approach to safeguard Cornell users, including scrambling the user passwords, using extra monitoring for those accounts, and using vec-ppsm on the server side.

Among the $1,688$ vulnerable accounts, 93% of accounts belong to alumni of Cornell. Our experiment also found many accounts, belonging to current faculty, staff, and students, are vulnerable to targeted attacks and helped ITSO safeguard those accounts better.

In Figure 11 we show the distribution of accounts found in the leak and the accounts that are vulnerable to pass2path. We also note the odds of being vulnerable to targeted online guessing attacks for each type of accounts as the fraction of accounts in each category that are vulnerable to any of the simulated targeted online attacks. As we can see alumni accounts have higher odds of being vulnerable to such attacks compared to other accounts.

Interestingly, we also found that the passwords that are reset most recently are less likely to be vulnerable. In Figure 12, we show the relation between the last password reset year and the odds of those account being vulnerable to online guessing attacks. We only consider the $15,776$ active accounts (that have changed their password at least once since 2009) for this graph. The passwords created before 2013 is 60% more likely to be vulnerable to targeted online guessing attacks compared to the passwords created after 2013.

REFERENCES

[1] F. Stajano, "Pico: No more passwords!" in *International Workshop on Security Protocols*. Springer, 2011, pp. 49–81.

[2] C.-T. Li and M.-S. Hwang, "An efficient biometrics-based remote user authentication scheme using smart cards," *Journal of Network and computer applications*, vol. 33, no. 1, pp. 1–5, 2010.

[3] X. Li, J. Niu, M. K. Khan, and J. Liao, "An enhanced smart card based remote user password authentication scheme," *Journal of Network and Computer Applications*, vol. 36, no. 5, pp. 1365–1371, 2013.

[4] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *2012 IEEE Symposium on Security and Privacy*, May 2012.

[5] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 173–186.

[6] A. Adams and M. A. Sasse, "Users are not the enemy," *Communications of the ACM*, vol. 42, no. 12, pp. 40–46, 1999.

[7] R. Chatterjee, A. Athalye, D. Akhawe, A. Juels, and T. Ristenpart, "password typos and how to correct them securely," *IEEE Symposium on Security and Privacy*, may 2016, full version of the paper can be found at the authors' website.

[8] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 295–310.

[9] Troy Hunt, "Have I Been Pwned?" https://haveibeenpwned.com/Passwords/, 2018.

[10] 4iQ, "Identities in the Wild: The Tsunami of Breached Identities Continues," https://4iq.com/wp-content/uploads/2018/05/2018_IdentityBreachReport_4iQ.pdf/, 2018.

[11] Shape Security, "2017 Credential spill report," http://info.shapesecurity.com/rs/935-ZAM-778/images/Shape-2017-Credential-Spill-Report.pdf/, 2018.

[12] P. A. Grassi, J. Fenton, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y. Choong *et al.*, "Nist special publication 800-63b. digital identity guidelines: Authentication and lifecycle management," *Bericht, NIST*, 2017.

[13] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse." in *NDSS*, vol. 14, 2014, pp. 23–26.

[14] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 1242–1254.

[15] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: an algorithmic framework and empirical analysis," in *ACM Conference on Computer and Communications Security (ACM CCS)*, 2010, pp. 176–186. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866328

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[18] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[19] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 364–372.

[20] "John the Ripper password cracker," http://www.openwall.com/john/, Referenced March 2014.

[21] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password

cracking using probabilistic context-free grammars," in *IEEE Symposium on Security and Privacy (SP)*, 2009, pp. 162–175.
[22] S. Komanduri, "Modeling the Adversary to Evaluate Password Strength with Limited Samples," 2016.
[23] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2014, pp. 689–704.
[24] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, "Fast, lean and accurate: Modeling password guessability using neural networks."
[25] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "PassGAN: A deep learning approach for password guessing," *arXiv preprint arXiv:1709.00440*, 2017.
[26] D. L. Wheeler, "zxcvbn: Low-budget password strength estimation," in *Proc. USENIX Security*, 2016.
[27] W. E. Burr, D. F. Dodson, and W. T. Polk, *Electronic authentication guideline*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2004.
[28] X. D. C. De Carnavalet, M. Mannan *et al.*, "From very weak to very strong: Analyzing password-strength meters." in *NDSS*, vol. 14, 2014, pp. 23–26.
[29] M. M. Devillers, "Analyzing password strength," *Radboud University Nijmegen, Tech. Rep*, 2010.
[30] M. Dell'Amico and M. Filippone, "Monte carlo strength evaluation: Fast and reliable password checking," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 158–169.
[31] J. Casal, "1.4 Billion Clear Text Credentials Discovered in a Single Database," https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14, Dec, 2017.
[32] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 538–552.
[33] Wikipedia, "Email address," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Email_address#Local-part
[34] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.
[35] C. M. Wilt, J. T. Thayer, and W. Ruml, "A comparison of greedy search algorithms," in *Third Annual Symposium on Combinatorial Search*, 2010.
[36] J. R. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely, "Partitioning attacks: or how to rapidly clone some gsm cards," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 2002, pp. 31–41.
[37] S. Mavoungou, G. Kaddoum, M. Taha, and G. Matar, "Survey on threats and attacks on mobile networks," *IEEE Access*, vol. 4, pp. 4543–4572, 2016.
[38] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password strength: An empirical analysis," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.
[39] B. Ur, F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L. F. Cranor, H. Dixon, P. Emami Naeini, H. Habib *et al.*, "Design and evaluation of a data-driven password meter," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 3775–3786.
[40] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
[41] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.
[42] "4iQ," https://4iq.com/, 2018.
[43] "Lastpass," https://lastpass.com.
[44] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017. [Online]. Available: https://github.com/facebookresearch/faiss
[45] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[46] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.
[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[48] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.
[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[50] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba, "Sequence level training with recurrent neural networks," *arXiv preprint arXiv:1511.06732*, 2015.
[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.
[52] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.