FROM ATTACK TO DEFENSE: BUILDING SYSTEMS SECURE AGAINST BREACHED CREDENTIALS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Bijeeta Pal

August 2022

© 2022 Bijeeta Pal

ALL RIGHTS RESERVED

FROM ATTACK TO DEFENSE:

BUILDING SYSTEMS SECURE AGAINST BREACHED CREDENTIALS

Bijeeta Pal, Ph.D.

Cornell University 2022

Targeted attacks using breached credentials exploit the fact that users reuse some semantic or syntactic structure of passwords across websites to make them easy to remember. Adversaries try to log in to a victim's account using the stolen passwords or variants of these passwords. Protecting accounts from these attacks remains challenging. Adversaries have wide-scale access to billions of stolen credentials from breach compilations, while users and identity providers remain in the dark about which accounts require attention. Our contribution is to show that it is possible to build a large-scale system that allows users to check for vulnerabilities against these attacks without sacrificing the functionality, security, and performance properties.

We initiate the work by addressing the core challenge — modeling how humans choose similar passwords. We train models using modern machine learning techniques and exhibit its efficacy by simulating the most damaging attack to date. Then we formalize the security goals for existing breach checking services that warn if the exact credential is publicly exposed. In the process we also propose novel exact-checking protocols with better security guarantees. All this helped educate the design of the second-generation, similarity-aware, and privacy-preserving credential checking service — Might I get Pwned (MIGP). Finally, we collaborate with Cloudflare to deploy MIGP as part of the web application firewall to notify login servers about potential attacks.

BIOGRAPHICAL SKETCH

Bijeeta Pal is a computer scientist specifically interested in security and privacy. She was born in India, where she completed her undergraduate in computer science at the Indian Institue of Technology (IIT), BHU. She always enjoyed the process of delving deep into a problem and trying to come up with solutions, which inherently got her introduced to the initial flavor of academic research. But being born to professors and spending her childhood on college campuses, she wanted to explore other choices before applying for graduate school. She, therefore, worked for a year in the industry before joining Purdue University for MS in computer science. Purdue University sparked her interest in applied cryptography, security, and privacy. She pursued her Ph.D. at Cornell University under the supervision of Prof. Thomas Ristenpart. During her Ph.D., she endeavored to gather and learn tools and techniques for building a safer and private online experience for everyone. She interned twice at Microsoft Research, once at JP Morgan, and was a recipient of the JP Morgan Fellowship.

For my parents, I am because you are

and Monodeep, my constant source of sunshine

ACKNOWLEDGEMENTS

The work in this thesis would not have been possible without the support of so many people. I am privileged to get a chance to work with all my collaborators and coauthors, from whom I learned a lot. But first and foremost, thanks to Thomas Ristenpart for being a truly wonderful mentor and teacher. Tom provided me with the platform to explore and learn. Yet in the moments when I needed help, he would jump in with a beautiful solution and some guidance. His support, feedback, and counsel throughout my Ph.D. have been vital; I am a better researcher and person in the end. Thank you!

I was fortunate to start working with Rahul Chatterjee early in my Ph.D. days. Thanks for being such an amazing collaborator, mentor, and most importantly friend. A conversation with Rahul can range from half-baked research ideas to philosophical questions to the best restaurants to explore in NYC.

I want to extend my gratitude to my colleagues and friends at Cornell University and Cornell Tech for creating such an intellectually stimulating and collaborative environment. I also want to thank the department staff members for promptly taking care of any administrative issues I encountered over the years.

My interest in security and privacy was first sparked while attending a graduate class by Mikhail Atallah during my masters. Mike motivated me to pursue Ph.D. and contributed greatly to my foundation as a researcher.

Last but not least, thanks to my friends and family, in India and USA, for being my support system. I wish to inherit from my parents, Basab Bijoy Pal and Mita Pal, their passion for learning and teaching, coupled with humility and kindness. Also, I am lucky to have my husband, Monodeep Kar, as my companion on this crazy journey.

Thank you all for everything.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Passwords are the most prevalent way to authenticate people on the web, despite being easy-to-guess [95] and hard-to-remember [31]. For convenience, users often pick same or similar passwords across different web services. Nearly 40% of users reuse their passwords or use slight variations [106]. The web service accounts with the same or similar passwords are as secure as the weakest ones among them. A breach at any one of these services also puts the other accounts at risk.

Against this backdrop, the sharing of breach compilations containing aggregated compromised data on underground forums has steadily increased [67]. For example, the compilation known as "Collections 1-5" includes 2.2 billion credentials. The widespread availability of breached credentials and password reuse across accounts makes credential stuffing attacks one of the most prominent security threats online. In credential stuffing attacks, the adversary uses a victim's leaked password from one website to compromise accounts associated with the victim on other websites. An incredible 90% of login traffic now consists of credential stuffing attack attempts [115]; it is the most prevalent form of account compromise [115].

A generalization of credential stuffing arises when the attacker picks common variants of a breached password to guess other active passwords of the user. These attacks can be damaging, which makes sense given the well-known tendency of users to select similar passwords [106], even after a password reset notification to prevent credential stuffing attacks [136]. The emergence of these large breached credential compilations accompanied by the increase in computational power and data analysis tools also lead to new opportunities for modeling how users choose these similar passwords.

Recently, some companies, such as HaveIBeenPwned (HIBP) [122], have started providing web services that actively collect the latest breaches and provide interfaces to query if a credential is leaked as a countermeasure against credential stuffing. HIBP has publicly documented APIs to check if a username or password is present in some breach. Such safeguards, now actively recommended by NIST [66], inform users about the at-risk usernames and passwords. These countermeasures, however, don't address the risk associated with users selecting or resetting the breached password to a similar password [50, 104]. Moreover, a rigorous formal and empirical analysis of how to design and implement such services is critical to prevent misuse.

This dissertation is about designing, implementing, and deploying a robust and efficient end-to-end system for checking and preventing attacks from breached credentials. Our research focuses on both studying the landscape of attacks that exploit breaches and building secure infrastructure to defend against them. We use a combination of cryptographic techniques to build protocols for checking the user's private input (their username and password) against sensitive breached data, and formal and empirical analysis for security evaluations as well as performance analysis to address questions that arise when incorporating these novel techniques into real systems.

The starting point for building such a system is to better understand the attack ecosystem. Users often reuse or pick similar passwords across different websites, making breached credential-based attacks effective. For example, a user selecting "qwerty1" as their password may often make minor modifications to generate a password for another website, e.g., “qwerty12”. We use “credential tweaking” to refer to attacks that submit variants of a leaked password. Credential stuffing is the more specific attack scenario where an adversary tries to compromise accounts by submitting the breached passwords unchanged.

A small number of prior academic works have investigated the efficacy of credential tweaking attacks that choose variants of leaked passwords based on mangling rules [50] or probabilistic context-free grammars (PCFG) [127]. In this work, we build a more damaging credential tweaking attack using state-of-the-art deep learning techniques. We demonstrate the attack’s effectiveness using simulations; 16% of user accounts can be compromised in less than 1000 guesses, despite the use of a credential stuffing countermeasure.

The standard countermeasure is to have users or websites proactively check if user credentials are present in known data breaches. This has given rise to web services, such as HaveIBeenPwned (HIBP) [122] and Google Password Checkup (GPC) [121], that released APIs to check for breached passwords. We refer to such services as compromised credential checking (C3) services. These services have to make various tradeoffs spanning from user privacy, accuracy, and performance. We provide the first formal description of C3 services, detailing different settings, operational requirements for deployment, relevant threat models and rigorous security analysis. We also introduce two new protocols that provide better security guarantees than HIBP and GPC, with comparable performance.

Finally, we initiate work on C3 services that protect users from credential tweaking attacks. This second-generation compromised credential checking system allows private similarity checking of passwords. We are interested in solving the decisional version of the problem: given a breach database D containing a set of username, password pairs (u_i, w_i), and a client's query containing username, password pair (u, w), is there an element u, w' in D, such that $w = w'$ or w is similar to w'. The owner of the C3 database D learns nothing about the user input and final query output. Also, a malicious client can not retrieve additional information about D. Therefore, the core underlying challenges of building such a system that scales with the size of D and satisfies the required privacy guarantees are: 1) Explore ways to define similarity in passwords, 2) Building a similarity-checking cryptographic protocol that preserves honest clients' privacy, 3) Preventing malicious clients from quickly extracting the breach data D, and 4) Ensuring the system is fast for deployment in practice.

We overcome these challenges and design "Might I Get Pwned" (MIGP), a new kind of breach alerting service. Our simulations show that MIGP reduces the efficacy of state-of-the-art 1000-guess credential tweaking attacks by 94%. MIGP preserves user privacy and limits potential exposure of sensitive breach entries. We show that the protocol is fast, with response time close to existing C3 services. We also worked with Cloudflare, a major computer security company, to deploy MIGP in practice.

1.1 Overview

This dissertation is organized into three parts. In the first part (Chapter 2), we study the existing attacks based on leaked credentials and develop the most damaging credential tweaking attack to date. The second part (Chapter 3) focuses on the design and implementation of compromised credential checking (C3) systems that protect users against credential stuffing attacks. We also explore the existing C3 services like HIBP and GPC and analyze the security guarantees of these services. These services, however, only check if the exact password is leaked, and therefore do not mitigate credential tweaking attacks. This motivates the design of a second-generation C3 system, Might I Get Pwned (MIGP), that also warns users about passwords similar to the breached ones. The third part (Chapter 4) presents the detailed cryptographic protocol and system design of MIGP along with security and performance evaluation for deployment in practice. In the remainder of the introduction, I will explain the problem in detail and outline the upcoming chapters in the body of the thesis.

1.1.1 Attacks against breached credentials

Text passwords continue to be used widely for online authentication. Given human constraints on memorizing a large number of passwords, users often adopt various strategies — including reuse and weak passwords — for managing their growing online accounts. Das et al. [50] estimated (based on breached data) the percentage of users that have reused passwords across multiple websites to be in the $43–51\%$ range. The use of password managers for the generation of random passwords is still not widely adopted by users. These factors contribute to the increasing threat of credential tweaking and stuffing attacks; inverting a hash of a single weak password or a breach in a single account makes other accounts vulnerable to these attacks.

Credential stuffing attacks are straightforward — breached passwords from one website are used to make login attempts to another. Credential tweaking attacks, however, require generating similar passwords as login attempts to other websites. A key differentiator between our work and prior password credential tweaking attacks is a novel data-driven, machine-learning approach that learns the similarity between passwords of the same user. The similarity can be interpreted as the conditional probability $P(w'|w)$, where w is a leaked password from one site and w' is a user's chosen password at another website.

An estimate of this conditional probability, for all relevant w, was learned using a compilation of password leaks containing 1.4 billion email-password pairs. We cleaned and joined this dataset using different heuristics to identify passwords corresponding to the same user. A generative model was trained on the dataset, capturing the conditional probability $P(w|w')$ using the sequence-to-sequence (seq2seq) architecture [119]. The model predicts the modifications to w to generate w', which gives better accuracy than trying to predict the full password w' itself. This model-based attack is 1.2 times more effective than the previous best credential tweaking attack and 3 times more effective than the best untargeted attack (attacks that don't take a leaked password into account to tailor guesses).

To evaluate the efficacy of credential tweaking attacks on real-world authentication systems, we performed experiments in collaboration with Cornell University's IT security group. In a first-of-its-kind experiment, we tested the efficacy of remote credential tweaking attacks on real user accounts. The breach compilation dataset mentioned above included $19,868$ Cornell emails. Passwords associated with $1,316$ active Cornell user accounts were guessed by our attack in less than $1,000$ guesses. These accounts were put under a watchlist by the Cornell IT security office; our research directly improved the security of these accounts.

Possible defenses

An obvious solution to stop these attacks is to perform audits based on the attack techniques described in the previous paragraph. But these audits are expensive to run. To solve this problem, we introduce the notion of a personalized password strength meter (PPSM), which also considers the similarity of the selected password to the user's leaked password(s) to estimate the selected password's strength.

We implemented a PPSM using embedding-based neural networks. The embedding maps a password to a d-dimensional vector in R^d. The property of the embedding vector is that vectors corresponding to similar passwords (passwords that are chosen by the same user) are closer in vector space. Therefore, we can measure the similarity score between two passwords: compute the embedding vectors and the distance between them. The similarity score warns users from picking vulnerable passwords and guides them towards selecting passwords that will resist credential tweaking attacks. We also made sure that the neural network is lightweight (3 Mb) and fast (0.3 ms) so that it is deployable in a variety of useful contexts.

There are a few different deployment scenarios where PPSMs will help improve security. A PPSM can be integrated as a part of password managers to flag similar selected passwords. One can also use a PPSM in the password-change workflow to determine whether the new password is a variant of the old one. A PPSM, combined with compulsory password change after a breach notification, should be effective at preventing credential tweaking attacks. A PPSM can also be deployed with login functionality. Whenever the user logins, the server checks whether the input password is unsafe based on leaked passwords associated with that account.

Information on what credentials have appeared in breaches is tedious for websites to maintain and often outsourced to third-party like C3 services. GPC, a C3 service discussed earlier, performs scalable private set intersection [121] with the server containing breached datasets. We provide a more detailed analysis of the C3 services in the following subsection 1.1.2. However, current C3 services don't warn users if the input password is similar to any breached passwords. We explore the question of building an improved C3 service with this ability in the subsection 1.1.3.

1.1.2 Compromised Credential Checking System

We compare the security leakage in various deployed C3 services like Google Password Checker and HaveIBeenPwned by formalizing the security requirements of such systems. The secrecy of the client's credential is paramount in these systems; the privacy of the C3 database is also desired. GPC was released as a chrome-extension, concurrently with our paper, in 2019 and later integrated into Chrome. GPC checks if a username-password pair is present in the leak instead of just the username or password, in the case of HIBP, leading to fewer false positives.

All the deployed protocols split the C3 database into smaller buckets and engage in a private set intersection (PSI) protocol [79] to reduce bandwidth use. The current C3 systems use the prefix of the user's credential as the bucket identifier, but we show that this can make the user password easy to guess for an attacker that observes the client's queries. We evaluate the security of such systems using theoretical and empirical analysis and show that knowing the bucket identifier can lead to a 12x increase in attack success. We reported this to Google, who later switched to our suggested improved approach (discussed later).

We also introduce new protocols that provide better security guarantees for both settings - HIBP, where only passwords are stored at the C3 server and GPC, where both username and passwords are stored. The main idea is that partitioning the leaks more effectively reduces security leakage. In password-only setting, we propose a novel bucketization technique, frequency-based bucketization, where the passwords are assigned to buckets in a way so that the bucket access pattern is flattened.

In the username-password setting, we propose a simple modification to the already deployed GPC protocol. The partitioning of the database should be done using the hash prefix of just the username instead of the username-password pair. This change ensures that no information about passwords is revealed to the C3 server (assuming username and passwords are independent). We refer to this protocol as ID-based bucketization (IDB). This change in protocol gives comparable performance guarantees. We reported this to Google, who later transitioned to our suggested improved approach. As mentioned, Google has transitioned to this scheme.

1.1.3 Might I Get Pwned

Existing C3 services only prevent credential stuffing attacks, and leave the users susceptible to credential tweaking attacks. As discussed earlier, credential tweaking attacks exploit the fact that users select similar passwords across websites [106]. Attackers can therefore guess variations of a leaked password to compromise other vulnerable accounts. Our work, discussed earlier, showed that the state-of-the-art credential tweaking attack techniques can compromise 16% of the user accounts in less than a thousand guesses given access to breached passwords for the user, despite existing C3 services in place that prevents selection of the breached password.

This motivates the design of a second-generation breach alerting system called Might I Get Pwned, MIGP, that checks if a password is vulnerable to credential tweaking attacks without revealing the password to the server. We explore the inherent tension between efficacy, security, and performance and tackle the primary challenges of building such a system.

MIGP builds off the first generation C3 protocol, the IDB protocol, that performs an private set intersection (PSI) to check for equality. The IDB protocol executes the PSI over buckets (subsets) of breach data, partitioned based on the hash prefix of usernames. To extend the IDB protocol with similarity checking feature, we augment the breached dataset with n variants of each leaked password and allow users to generate m variants on client-side. We then perform the IDB protocol between the client's queried data and the server's original breached and respective variants. The desired outcome of the protocol is the user getting informed, in case of a match, if it is a variant or original password.

To concretize this approach, we need to answer the following questions — how to generate the similar variants, and what are the optimal values of m and n are. We empirically evaluate the effectiveness of different similarity measures. We show that with the values of $m = 10$ and $n = 10$, the success rate of state-of-the-art credential tweaking attacker with 1000 guesses reduces by 94% compared to using only exact-checking, even when the attacker adapts its strategy to the use of MIGP.

C3 services can be used by malicious users to attempt to extract sensitive and sometimes confidential breach data. We refer to these attacks as breach extraction attacks. The upgrade to similarity-aware breach alerting raises the natural question: Will the extra information regarding similarity to breached data make breach extraction attacks easier? We formalize this attack setting and analyze it both analytically and empirically. We discuss some mitigation techniques against this leakage — such as blocklisting common passwords and rate-limiting client-side queries using proof-of-work, which can significantly reduce the attack success rate.

Finally, we implement MIGP and show that online computational cost is comparable to the existing C3 services (less than 500 ms), showing its feasibility for deployment in practice.

1.1.4 Impact in practice

We worked with Cloudflare, a major CDN and computer security company [24], to deploy the MIGP protocol — (1) as a public-facing API, and (2) as an internal component of Cloudflare's web application firewall (WAF) product for breach alerting [24].

Cloudflare has released their Exposed Credential Check feature as part of their Web Application Firewall (WAF). Websites can opt-in for this feature, which informs the websites about login attempts to their sites that used compromised credentials. Underlying this feature is the MIGP service deployed on Cloudflare Workers. The MIGP service latency is under 135 ms for over 50% of client requests and under 573ms for 95% of requests. The source code underlying the MIGP implementation at Cloudflare is publicly available [23].

To estimate the effectiveness of the MIGP service, we instrumented a measurement study on the WAF deployment. We concluded that MIGP flags 20% more vulnerable login attempts than exact-checking C3 systems. With the large-scale deployment of MIGP at Cloudflare, we demonstrate the feasibility, practicality, and usefulness of checking for breached credentials in a privacy-preserving way.

1.2 Bibliographical notes

This dissertation is based on the following jointly authored publications:

- Chapters 2: "Beyond Credential Stuffing: Password Similarity Models Using Neural Networks," with Tal Daniel, Rahul Chatterjee and Thomas Ristenpart, published at the IEEE Security and Privacy in 2019 [104],
- Chapter 3: "Protocols for checking compromised credentials," with Lucy Li, Junade Ali, Nick Sullivan, Rahul Chatterjee and Thomas Ristenpart, which appeared at the ACM Computer and Communications Security in 2019 [88], and
- Chapter 4: "Might I Get Pwned: A Second Generation Password Breach Alerting Service," with Mazharul Islam, Marina Sanusi, Nick Sullivan, Luke Valenta,Tara Whalen, Christopher Wood, Thomas Ristenpart and Rahul Chatterjee, which appeared at the USENIX Security Symposium in 2022 [105].

CHAPTER 2

BEYOND CREDENTIAL STUFFING: PASSWORD SIMILARITY MODELS USING NEURAL NETWORKS

2.1 Introduction

Despite repeated calls to replace passwords entirely with different authentication mechanisms [40, 87, 89, 117], human-chosen passwords remain widespread today and will continue for the foreseeable future. This is true despite their notoriety for being easy-to-guess [95], hard-to-remember [31], and difficult-to-type-correctly [44]. The latter two issues tend to encourage reuse of similar passwords across websites: nearly 40% of users reuse their passwords or use slight variations [106].

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 [29, 122]; bot-driven credential stuffing attacks account for 90% of the login traffic to some of the world's largest websites [115]; and these attacks represent the largest source of account take over [115].

Website operators, sometimes with the help of third-party services such as HIBP [122], reset user passwords if their usernames or passwords are found in breaches. Such safeguards, which are now actively being recommended by

NIST [66], 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 [50] or probabilistic context-free grammars (PCFG) [127]. They show such targeted attacks can be damaging, which makes sense given the well-known tendency of users to pick similar passwords [106], even after a password reset [136]. 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 $\Pr[w \mid \tilde{w}]$, 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 $\Pr[w \mid \tilde{w}]$ for all $\tilde{w}$ using sequence-to-sequence (seq2seq) algorithms [120].

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, 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

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

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 [37, 99], 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.

2.2 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 [28,101]. 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) [130]. They were later improved by Komanduri et al. in [82] to estimate the distribution of human-chosen passwords. Ma et al. [91] 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. [97] used recurrent neural networks (RNNs) and Hitaj et al. [70] 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. [50] 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. [127] 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 [97, 131], 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 [41]. This approach has various deficiencies, see [52,55]. 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 [53].

2.3 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 [127] 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 [50] or using probabilistic context-free grammars (PCFG) [127]. 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 [44], 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 [120]. 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 [99]. 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 2.7, 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 [42].[2] The dataset consists of 1.4 billion email-password pairs, with 1.1 billion unique emails 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.)

[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

Figure 2.1: The distribution of password length and composition in the data after cleaning.

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 [39,127]. Figure 2.1 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 [132]) 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 2.2), which is slightly below what prior works reported (40%) [106]. Also, as we can see in Figure 2.2, 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.

		D^E_{full}	D^U_{full}	D^M_{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
	≥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
	≥4	82.1	86.9	81.8

Figure 2.2: Comparison of the datasets under three account joining techniques: email address (D^E_{full}), username (D^U_{full}), and a combination of email and username (D^M_{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 (%).

- **Mixed method (D^M_{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 2.2) are in line with prior studies [50, 127]. 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: $\mathrm{D}^{\mathsf{E}}_{\mathsf{tr}}$ (80%) and $\mathrm{D}^{\mathsf{E}}_{\mathsf{ts}}$ (20%). Similarly the mixed-dataset into $\mathrm{D}^{\mathsf{M}}_{\mathsf{tr}}$ (80%) and $\mathrm{D}^{\mathsf{M}}_{\mathsf{ts}}$ (20%). Unless otherwise specified, for all training and validation (during training) we use $\mathrm{D}^{\mathsf{E}}_{\mathsf{tr}}$. This is because the distribution of similar (unique) passwords of a user in $\mathrm{D}^{\mathsf{M}}_{\mathsf{tr}}$ and $\mathrm{D}^{\mathsf{E}}_{\mathsf{tr}}$ were almost identical. Since we only consider similar passwords of a user during training, we do not use $\mathrm{D}^{\mathsf{M}}_{\mathsf{tr}}$ for training separately. For testing, we use random samples from both $\mathrm{D}^{\mathsf{E}}_{\mathsf{ts}}$ and $\mathrm{D}^{\mathsf{M}}_{\mathsf{ts}}$.

2.4 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,

$$\begin{aligned} P\left(w \,\middle|\, \tilde{w}\right) &= P\left(c_1, \ldots, c_l \,\middle|\, \tilde{c}_1, \ldots, \tilde{c}_{l'}\right) \\ &= \prod_{i=1}^{l} P\left(c_i | \tilde{c}_1, \ldots, \tilde{c}_l, c_1, \ldots, c_{i-1}\right) . \end{aligned} \tag{2.1}$$

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. [120] 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.0.1 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 A.0.3 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.

$$P\left(w \,\middle|\, \tilde{w}\right) = P\left(T_{\tilde{w}\to w} \,\middle|\, \tilde{w}\right) = \prod_{i=1}^{t} P\left(\tau_i \,\middle|\, \tilde{w}, \tau_1, \ldots, \tau_{i-1}\right) \,, \tag{2.2}$$

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

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

The model architecture of pass2path is similar to encoder-decoder based architecture used for seq2seq instantiated using two recurrent neural networks (RNN) [120]. The encoder and decoder RNNs are trained together. The details of the model architecture are given in Appendix A.0.2. 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 2.5.

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 re-ordering 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 'stc$\rangle$passwordsts$\rangle$1', where stc$\rangle$ and sts$\rangle$ represents caps-lock and shift key on the keyboard. Chatterjee et al. [44] 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 [86]. 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 A.0.4.

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 A.0.3.

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.

2.5 Targeted Guessing Attack using Pass2Path

As discussed in Section 4.2, 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 [133]. 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: $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$, generated using only the email to identify users, and $\mathsf{D}_{\mathsf{ts}}^{\mathsf{M}}$, generated using the mixed method (see Section 3.3). 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.

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. [50]	5.8	9.2	11.0
Wang et al. [127]	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. [50]	42.7	44.8	45.9
Wang et al. [127]	43.2	44.3	47.0
Pass2pass	43.7	45.0	45.8
Pass2path	**44.8**	**46.7**	**48.3**

Figure 2.3: Percentage of passwords guessed by various attacks in q guesses on the two test sets generated from (**left**) $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$, the without-repeats setting, and (**right**) $\mathsf{D}^{\mathsf{M}}_{\mathsf{ts}}$, 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.

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. [50] created a manually curated list of transformations to generate similar passwords. Wang et al. [127] 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 [127] 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 A.0.5 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 [97] 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 2.3 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 $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$ 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. [127] 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 faster than Wang et al.

Attack efficacy in the with-repeats setting. We now discuss the results on the test set derived from $\mathsf{D}^{\mathsf{M}}_{\mathsf{ts}}$. 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 $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$ 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 $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$ 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

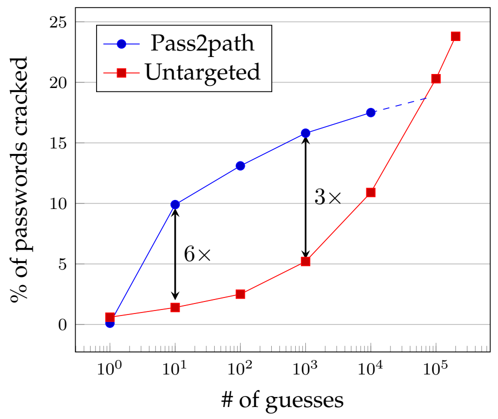

Figure 2.4: The relative advantage of targeted attacks and untargeted attacks for large guessing budgets, and crossover point where untargeted attacks becomes more effective than targeted attacks. Due to computational limits, we did not compute points for greater than 10^4 guesses using pass2path, and so the dotted line reflects the observed trend.

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.

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 2.4 for different number of guesses. To generate this graph, we sampled 10^5 random users from $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$ 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 2.3. 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 2.6, 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 [94,113] 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 2.6.

2.6 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 [95]), 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.

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

Attack	$q = 10$	$q = 10^2$	$q = 10^3$
Untargeted-empirical	0	0	0.1
Wang et al. [127]	0.2	0.6	2.6
Pass2path	**3.3**	**6.0**	**8.4**

Figure 2.5: 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.

generated by the Wang et al. attack algorithm. More details about the experiment setup are given in Appendix A.0.6.

Results. The results of the experiments are summarized in Figure 2.5. 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.

2.7 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 [41], but these measures are wildly inaccurate [52, 54]. State-of-the-art strength meters [97, 131] 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 [131], 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. [97] 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 [131] gives a score in $\{0, \ldots, 5\}$ and nn-pwmeter [124] 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 [37, 99, 107]. 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 [37]. 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 “st” and “$\rangle$” (that are not otherwise in Σ). For example, a password w = `qwerty` with $m_{\min} = 4$ and $m_{\max} = 5$, will have the following n-grams. z_w = {`stqwerty`$\rangle$, `stqwe, qwer, wert, erty, rty`$\rangle$, `stqwer, qwert, werty, erty`$\rangle$}. (Note, the full password is always included in z_w.) Then, the score function is:

$$s(w, \tilde{w}) = \left(\frac{1}{|z_w|} \sum_{g \in z_w} u_g \right)^{\top} \left(\frac{1}{|z_{\tilde{w}}|} \sum_{g \in z_{\tilde{w}}} u_g \right) \tag{2.3}$$

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, $v_w = \frac{1}{|z_w \cap \mathcal{V}|} \sum\limits_{g \in z_w \cap \mathcal{V}} u_g$. 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 2.4. 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 $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$. 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

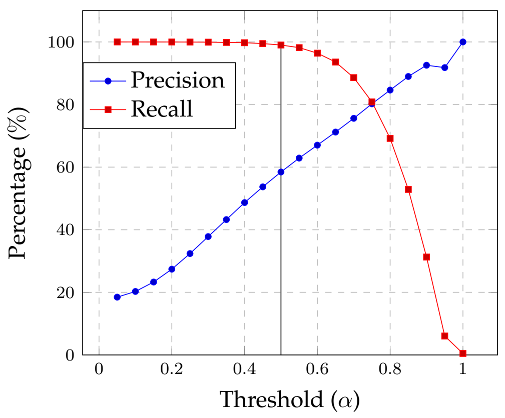

Figure 2.6: Precision and recall of our PPSM classifier for different values of the threshold α computed over a random sample of 10^5 password pairs from $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$.

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 2.6. 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%.

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

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

Figure 2.7: 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.

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 [75] to further compress the vectors, which has been shown to be effective for compressing neural network models [97]. 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, and compute their precision and recall on a sample of 10^5 password pairs chosen from that many random users from $\mathsf{D}_{\mathsf{ts}}^{\mathsf{E}}$. The results are noted in Figure 2.7. 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.

2.8 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 2.7, 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 [131] and nn-pwmeter, a neural network based strength meter proposed in [97,124]. 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 [50, 127] can be used to construct a personalized strength meter, but Das et al. performs poorly as a targeted attack (see Figure 2.3), 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.

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 2.5. We also compute the strength of the target passwords under all the strength meters under consideration.

In Figure 2.8, 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

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

Figure 2.8: 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.

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 2.8.)

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 2.6. 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., [2,122]) 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 [1]. 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 [76]. 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.

2.9 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.

CHAPTER 3

PROTOCOLS FOR CHECKING COMPROMISED CREDENTIALS

3.1 Introduction

Password database breaches have become routine [11]. Such breaches enable credential stuffing attacks, in which attackers try to compromise accounts by submitting one or more passwords that were leaked with that account from another website. To counter credential stuffing, companies and other organizations have begun checking if their users' passwords appear in breaches, and, if so, they deploy further protections (e.g., resetting the user's passwords or otherwise warning the user). Information on what usernames and passwords have appeared in breaches is gathered either from public sources or from a third-party service. The latter democratizes access to leaked credentials, making it easy for others to help their customers gain confidence that they are not using exposed passwords. We refer to such services as *compromised credential checking* services, or C3 services in short.

Two prominent C3 services already operate. HaveIBeenPwned (HIBP) [122] was deployed by Troy Hunt and CloudFlare in 2018 and is used by many web services, including Firefox [16], EVE Online [12], and 1Password [7]. Google released a Chrome extension called Password Checkup (GPC) [111,121] in 2019 that allows users to check if their username-password pairs appear in a compromised dataset. Both services work by having the user share with the C3 server a prefix of the hash of their password or of the hash of their username-password pair. This leaks some information about user passwords, which is problematic should the C3 server be compromised or otherwise malicious. But until now there has been no thorough investigation into the damage from the leakage of current C3 services or suggestions for protocols that provide better privacy.

We provide the first formal treatment of C3 services for different settings, including an exploration of their security guarantees. A C3 service must provide secrecy of client credentials, and ideally, it should also preserve secrecy of the leaked datasets held by the C3 server. The computational and bandwidth overhead for the client and especially the server should also be low. The server might hold billions of leaked records, precluding use of existing cryptographic protocols for private set intersection (PSI) [64,96], which would use a prohibitive amount of bandwidth at this scale.

Current industry-deployed C3 services reduce bandwidth requirements by dividing the leaked dataset into buckets before executing a PSI protocol. The client shares with the C3 server the identifier of the bucket where their credentials would be found, if present in the leak dataset. Then, the client and the server engage in a protocol between the bucket held by the server and the credential held by the client to determine if their credential is indeed in the leak. In current schemes, the prefix of the hash of the user credential is used as the bucket identifier. The client shares the hash prefix (bucket identifier) of their credentials with the C3 server.

Revealing hash prefixes of credentials may be dangerous. We outline an attack scenario against such prefix-revealing C3 services. In particular, we consider a conservative setting where the C3 server attempts to guess the password, while knowing the username and the hash prefix associated with the queried credential. We rigorously evaluate the security of HIBP and GPC under this threat model via a mixture of formal and empirical analysis.

We start by considering users with a password appearing in some leak and show how to adapt a recent state-of-the-art credential tweaking attack [104] to take advantage of the knowledge of hash prefixes. In a credential tweaking attack, one uses the leaked password to determine likely guesses (usually, small tweaks on the leaked password). Via simulation, we show that our variant of credential tweaking successfully compromises 83% of such accounts with 1,000 or fewer attempts, given the transcript of a query made to the HIBP server. Without knowledge of the transcript, only 56% of these accounts can be compromised within 1,000 guesses.

We also consider user accounts not present in a leak. Here we found that the leakage from the hash prefix disproportionately affects security compared to the previous case. For these user accounts, obtaining the query to HIBP enables the attacker to guess 71% of passwords within 1,000 attempts, which is a 12x increase over the success with no hash prefix information. Similarly, for GPC, our simulation shows 33% of user passwords can be guessed in 10 or fewer attempts (and 60% in 1,000 attempts), should the attacker learn the hash prefix shared with the GPC server.

The attack scenarios described are conservative because they assume the attacker can infer which queries to the C3 server are associated to which usernames. This may not be always possible. Nevertheless, caution dictates that we would prefer schemes that leak less. We therefore present two new C3 protocols, one that checks for leaked passwords (like HIBP) and one that checks for leaked username-password pairs (like GPC). Like GPC and HIBP, we *partition* the password space before performing PSI, but we do so in a way that reduces leakage significantly.

Our first scheme works when only passwords are queried to the C3 server. It utilizes a novel approach that we call frequency-smoothing bucketization (FSB). The key idea is to use an estimate of the distribution of human-chosen passwords to assign passwords to buckets in a way that flattens the distribution of accessed buckets. We show how to obtain good estimates (using leaked data), and, via simulation, that FSB reduces leakage significantly (compared to HIBP). In many cases the best attack given the information leaked by the C3 protocol works no better than having no information at all. While the benefits come with some added computational complexity and bandwidth, we show via experimentation that the operational overhead for the FSB C3 server or client is comparable with the overhead from GPC, while also leaking much less information than hash-prefix-based C3 protocols.

We also describe a more secure bucketizing scheme that provides better privacy/bandwidth trade-off for C3 servers that store username-password pairs. This scheme was also (independently) proposed in [121], and Google states that they plan to transition to using it in their Chrome extension. It is a simple modification of their current protocol. We refer to it as IDB, ID-based bucketization, as it uses the hash prefix of only the user identifier for bucketization (instead of the hash prefix of the username-password pair, as currently used by GPC). Not having password information in the bucket identifier hides the user's password perfectly from an attacker who obtains the client queries (assuming that passwords are independent of usernames). We implement IDB and show that the average bucket size in this setting for a hash prefix of 16 bits is similar to that of GPC (average 16,122 entries per bucket, which leads to a bandwidth of 1,066 KB).

Contributions. In summary, the main contributions of this paper are the following:

- We provide a formalization of C3 protocols and detail the security goals for such services.
- We discuss various threat models for C3 services, and analyze the security of two widely deployed C3 protocols. We show that an attacker that learns the queries from a client can severely damage the security of the client's passwords, should they also know the client's username.
- We give a new C3 protocol (FSB) for checking only leaked passwords that utilizes knowledge of the human-chosen password distribution to reduce leakage.
- We give a new C3 protocol for checking leaked username-password pairs (IDB) that bucketizes using only usernames.
- We analyze the performance and security of both new C3 protocols to show feasibility in practice.

We will release as public, open source code our server and client implementations of FSB and IDB.

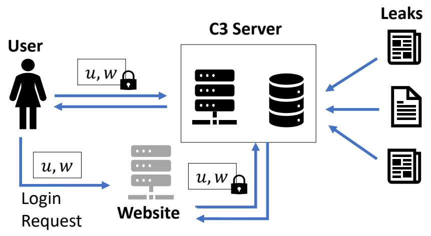

Figure 3.1: A C3 service allows a client to ascertain whether a username and password appear in public breaches known to the service.

3.2 Overview

We investigate approaches to checking credentials present in previous breaches. Several third party services provide credential checking, enabling users and companies to mitigate credential stuffing and credential tweaking attacks [50, 104, 127] , an increasingly daunting problem for account security.

To date, such C3 services have not received in-depth security analyses. We start by describing the architecture of such services, and then we detail relevant threat models.

C3 settings. We provide a diagrammatic summary of the abstract architecture of C3 services in Figure 3.1. A C3 *server* has access to a breach database $\tilde{S}$. We can think of $\tilde{S}$ as a set of size N, which consists of either a set of passwords $\{w_1, \dots, w_N\}$ or username-password pairs $\{(u_1, w_1), \dots, (u_N, w_N)\}$. This corresponds to two types of C3 services — *password-only C3 services* and *username-password C3 services*. For example, HIBP [8] is a password-only C3 service,[1] and Google's service GPC [111] is an example of a username-password C3 service.

[1]HIBP also allows checking if a user identifier (email) is leaked with a data breach. We focus on password-only and username-password C3 services.

A *client* has as input a credential $s = (u, w)$ and wants to determine if s is at risk due to exposure. The client and server therefore engage in a set membership protocol to determine if $s \in \tilde{\mathcal{S}}$. Here, clients can be users themselves (querying the C3 service using, say, a browser extension), or other web services can query the C3 service on behalf of their users. Clients may make multiple queries to the C3 service, though the number of queries might be rate limited.

The ubiquity of breaches means that, nowadays, the breach database $\tilde{\mathcal{S}}$ will be quite large. A recently leaked compilation of previous breached data contains 1.4 billion username password pairs [42]. The HIBP database has 501 million unique passwords [8]. Google's blog specifies that there are 4 billion username-password pairs in their database of leaked credentials [111].

C3 protocols should be able to scale to handle set membership requests for these huge datasets for millions of requests a day. HIBP reported serving around 600,000 requests per day on average [9]. The design of a C3 protocol should therefore not be expensive for the server. Some clients may have limited computational power, so the C3 protocol should also not be expensive on the client-side. The number of network round trips required must be low, and we restrict attention to protocols that can be completed with a single HTTPS request. Finally, we will want to minimize bandwidth usage.

Threat model. We consider the security of C3 protocols relative to two distinct threat models: (1) a malicious client that wants to learn a different user's password; and (2) an honest-but-curious C3 server that aims to learn the password corresponding to a C3 query. We discuss each in turn.

A malicious client may want to use the C3 server to discover another user's password. The malicious client may know the target's username and has the ability to query the C3 server. The C3 server's database $\tilde{S}$ should therefore be considered confidential, and our security goal here is that each query to the C3 server can at most reveal whether a particular w or (u, w) is found within the breach database, for password-only and username-password services, respectively. Without some way of authenticating ownership of usernames, this seems the best possible way to limit knowledge gained from queries. We note that most breach data is in fact publicly available, so we should assume that dedicated adversaries in this threat model can find (a substantial fraction of) any C3 service's dataset. For such adversaries, there is little value in attempting to exploit the C3 service via queries. Nevertheless, deployments should rate-limit clients via IP-address-based query throttling as well as via slow-to-compute hash functions such as Argon2 [4].

The trickier threat model to handle is (2), and this will consume most of our attention in this work. Here the C3 server may be compromised or otherwise malicious, and it attempts to exploit a client's queries to help it learn that client's password for some other target website. We assume the adversary can submit password guesses to the target website, and that it knows the client's username. We refer to this setting as a known-username attack (KUA). We conservatively[2] assume the adversary has access to the full breach dataset, and thus can take advantage of both leaked passwords available in the breach dataset and information leaked about the client's password from C3 queries. Looking ahead, for our protocols, the information potentially leaked from C3 queries is the bucket identifier.

[2]This is conservative because the C3 server need not, and should not, store passwords in-the-clear, and it should instead obfuscate them using an oblivious PRF.

It is context-dependent whether a compromised C3 server will be able to mount KUAs. For example, in deployments where a web server issues queries on behalf of their users, queries associated to many usernames may be intermingled. In some cases, however, an adversary may be able to link usernames to queries by observing meta-data corresponding to a query (e.g., IP address of the querying user or the timing of a request). One can imagine cross-site scripting attacks that somehow trigger requests to the C3 service, or the adversary might send tracking emails to leaked email addresses in order to infer an IP address associated to a username [59]. We therefore conservatively assume the malicious server's ability to know the correct username for a query.

In our KUA model, we focus on online attack settings, where the attacker tries to impersonate the target user by making remote login attempts at another web service, using guessed passwords. These are easy to launch and are one of the most prevalent forms of attacks [29, 60]. However, in an online setting, the web service should monitor failed login attempts and lock an account after too many incorrect password submissions. Therefore, the attacker gets only a small number of attempts. We use a variable q, called the guessing budget, to represent the allowed number of attempts.

Should the adversary additionally have access to password hashes stolen from the target web site, they can instead mount an offline cracking attack. Offline cracking could be sped up by knowledge of client C3 queries, and one can extend our results to consider the offline setting by increasing q to reflect computational limits on adversaries (e.g., $q = 10^{10}$) rather than limits on remote login attempts. Roughly speaking, we expect the leakage of HIBP and GPC to be proportionally as damaging here, and that our new protocol FSB will not provide as much benefit for very large q (see discussion in Section 3.6). IDB will provide no benefit to offline cracking attacks (assuming they already know the username).

Finally, we focus in threat model (2) on honest-but-curious adversaries, meaning that the malicious server does not deviate from its protocol. Such actively malicious servers could lie to the client about the contents of $\tilde{\mathcal{S}}$ in order to encourage them to pick a weak password. Monitoring techniques might be useful to catch such misdeeds. For the protocols we consider, we do not know of any other active attacks advantageous to the adversary, and do not consider them further.

Potential approaches. A C3 protocol requires, at core, a secure set membership query. Existing protocols for private set intersection (a generalization of set membership) [47, 81, 109, 110] cannot currently scale to the set sizes required in C3 settings, $N \approx 2^{30}$. For example, the basic PSI protocol that uses an oblivious pseudorandom function (OPRF) [81] computes $y_i = F_\kappa(u_i, w_i)$ for $(u_i, w_i) \in \tilde{\mathcal{S}}$ where F_κ is the secure OPRF with secret key κ (held by the server). It sends all $y_1, \ldots, y_N$ to the client, and the client obtains $y = F_\kappa(u, w)$ for its input (u, w) by obliviously computing it with the server. The client can then check if $y \in \{y_1, \ldots, y_N\}$. But clearly for large N this is prohibitively expensive in terms of bandwidth. One can use Bloom filters to more compactly represent the set $y_1, \ldots, y_N$, but the result is still too large. While more advanced PSI protocols exist that improve on these results asymptotically, they are unfortunately not yet practical for this C3 setting [79, 81].

Practical C3 schemes therefore relax the security requirements, allowing the protocol to leak some information about the client's queried (u, w) but hopefully

Credentials checked	Name	Bucket identifier	B/w (KB)	RTL (ms)	Security loss
Password	HIBP	20-bits of SHA1(w)	32	220	12x
	FSB	Figure 3.6, $\bar{q} = 10^2$	558	527	2x
(Username, password)	GPC	16-bits of Argon2$(u \| w)$	1,066	489	10x
	IDB	16-bits of Argon2(u)	1,066	517	1x

Figure 3.2: **Comparison of different C3 protocols**. HIBP [8] and GPC [111] are two C3 services used in practice. We introduce frequency-smoothing bucketization (FSB) and identifier-based bucketization (IDB). Security loss is computed assuming query budget $q = 10^3$ for users who have not been compromised before.

not too much. To date no one has investigated how damaging the leakage of currently proposed schemes is, so we turn to doing that next. In Figure 3.2, we show all the different settings for C3 we discuss in the paper and compare their security and performance. The security loss in Figure 3.2 is a comparison against an attacker that only has access to the username corresponding to a C3 query (and not a bucket identifier).

3.3 Bucketization Schemes and Security Models

In this section we formalize the security models for a class of C3 schemes that bucketize the breach dataset into smaller sets (buckets). Intuitively, a straightforward approach for checking whether or not a client's credentials are present in a large set of leaked credentials hosted by a server is to divide the leaked data into various buckets. The client and server can then perform a private set intersection between the user's credentials and one of the buckets (potentially) containing that credential. The bucketization makes private set membership tractable, while only leaking to the server that the password may lie in the set associated to a certain bucket.

We give a general framework to understand the security loss and bandwidth overhead of different bucketization schemes, and we will use this framework to evaluate existing C3 services.

Notation. To easily describe our constructions, we fix some notation. Let $\mathcal{W}$ be the set of all passwords, and p_w be the associated probability distribution; let $\mathcal{U}$ be the set of all user identifiers, and p be the joint distribution over $\mathcal{U} \times \mathcal{W}$. We will use $\mathcal{S}$ to denote the domain of credentials being checked, i.e., for password-only C3 service, $\mathcal{S} = \mathcal{W}$, and for username-password C3 service, $\mathcal{S} = \mathcal{U} \times \mathcal{W}$. Below we will use $\mathcal{S}$ to give a generic scheme, and specify the setting only if necessary to distinguish. Similarly, $s \in \mathcal{S}$ denotes a password or a username-password pair, based on the setting. Let $\tilde{\mathcal{S}}$ be the set of leaked credentials, and $|\tilde{\mathcal{S}}| = N$.

Let H be a cryptographic hash function from $\{0,1\}^* \mapsto \{0,1\}^\ell$, where ℓ is a parameter of the system. We use $\mathcal{B}$ to denote the set of buckets, and we let $\beta \colon \mathcal{S} \mapsto \mathcal{P}(\mathcal{B}) \setminus \{\varnothing\}$ be a bucketizing function which maps a credential to a set of buckets. A credential can be mapped to multiple buckets, and every credential is assigned to at least one bucket. An inverse function to β is $\alpha \colon \mathcal{B} \mapsto \mathcal{P}(\mathcal{S})$, which maps a bucket to the set of all credentials it contains; so, $\alpha(b) = \left\{ s \in \mathcal{S} \,\middle|\, b \in \beta(s) \right\}$. Note, $\alpha(b)$ can be very large given it considers all credentials in $\mathcal{S}$. We let $\tilde{\alpha}$ be the function that denotes the credentials in the buckets held by the C3 server, $\tilde{\alpha}(b) = \alpha(b) \cap \tilde{\mathcal{S}}$.

The client sends b to the server, and then the client and the server engage in a set intersection protocol between $\{s\}$ and $\tilde{\alpha}(b)$.

Symbol	Description
u / $\mathcal{U}$	user identifier (e.g., email) / domain of users
w / $\mathcal{W}$	password / domain of passwords
$\mathcal{S}$	domain of credentials
$\tilde{\mathcal{S}}$	set of leaked credentials, $\lvert\tilde{\mathcal{S}}\rvert = N$
p	distribution of username-password pairs over $\mathcal{U} \times \mathcal{W}$
p_w	distribution of passwords over $\mathcal{W}$
$\hat{p}_s$	estimate of p_w used by C3 server
q	query budget of an attacker
$\bar{q}$	parameter to FSB, estimated query budget of an attack
β	function that maps a credential to a set of buckets
α	function that maps a bucket to the set of credentials it contains

Figure 3.3: The notation used in this chapter.

```
$\mathsf{Guess}^{\mathcal{A}}(q)$
$(u, w) \leftarrow_p \mathcal{U} \times \mathcal{W}$
$\{\tilde{w}_1, \ldots, \tilde{w}_q\} \leftarrow \mathcal{A}(u, q)$
return $w \in \{\tilde{w}_1, \ldots, \tilde{w}_q\}$
```

```
$\mathsf{BucketGuess}_{\beta}^{\mathcal{A}'}(q)$
$(u, w) \leftarrow_p \mathcal{U} \times \mathcal{W}$; $s \leftarrow (u, w)$
$B \leftarrow \beta(s)$; $b \leftarrow_\$ B$
$\{\tilde{w}_1, \ldots, \tilde{w}_q\} \leftarrow \mathcal{A}'(u, b, q)$
return $w \in \{\tilde{w}_1, \ldots, \tilde{w}_q\}$
```

Figure 3.4: The guessing games used to evaluate security.

Bucketization schemes. Bucketization divides the credentials held by the server into smaller buckets. The client can use the bucketizing function β to find the set of buckets for a credential, and then pick one randomly to query the server. There are different ways to bucketize the credentials.

In the first method, which we call hash-prefix-based bucketization (HPB), the credentials are partitioned based on the first l bits of a cryptographic hash of the credentials. GPC [111] and HIBP [8] APIs use HPB. The distribution of the credentials is not considered in HPB, which causes it to incur higher security loss, as we show in Section 3.4.

We introduce a new bucketizing method, which we call frequency-smoothing bucketization (FSB), that takes into account the distribution of the credentials and replicates credentials into multiple buckets if necessary. The replication "flattens" the conditional distribution of passwords given a bucket identifier, and therefore vastly reduces the security loss. We discuss FSB in more detail in Section 3.5.

In both HPB and FSB, the bucketization function depends on the user's password. We give another bucketization approach — the most secure one — that bucketizes based only on the hash prefix of the user identifier. We call this identifier-based bucketization (IDB). This approach is only applicable for username-password C3 services. We discuss IDB in Section 3.4.

Security measure. The goal of an attacker is to learn the user's password. We will focus on online-guessing attacks, where an attacker tries to guess a user's password over the login interface provided by a web service. An account might be locked after too many incorrect guesses (e.g., 10), in which case the attack fails. Therefore, we will measure an attacker's success given a certain guessing budget q. We will always assume the attacker has access to the username of the target user.

The security games are given in Figure 3.4. The game Guess models the situation in which no information besides the username is revealed to the adversary about the password. In the game BucketGuess, the adversary also gets access to a bucket that is chosen according to the credentials $s = (u, w)$ and the bucketization function β.

We define the advantage against a game as the maximum probability that the game outputs 1. Therefore, we maximize the probability, over all adversaries, of the adversary winning the game in q guesses.

$$\mathsf{Adv}^{\mathsf{gs}}(q) = \max_{\mathcal{A}}\ P\left(\mathsf{Guess}^{\mathcal{A}}(q) \Rightarrow 1\right),$$

and

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta}(q) = \max_{\mathcal{A}'}\ P\left(\mathsf{BucketGuess}^{\mathcal{A}'}_{\beta}(q) \Rightarrow 1\right).$$

The probabilities are taken over the choices of username-password pairs and the selection of bucket via the bucketizing function β. The security loss $\Delta_{\beta}(q)$ of a bucketizing protocol β is defined as

$$\Delta_{\beta}(q) = \mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta}(q) - \mathsf{Adv}^{\mathsf{gs}}(q).$$

Note,

$$P\left(\mathsf{Guess}^{\mathcal{A}}(q) \Rightarrow 1\right) = \sum_{u} P\left(w \in \mathcal{A}(u,q) \wedge U = u\right).$$

To maximize this probability, the attacker must pick the q most probable passwords for each user. Therefore,

$$\mathsf{Adv}^{\mathsf{gs}}(q) = \sum_{u} \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} P\left(W = w_i \wedge U = u\right). \tag{3.1}$$

In $\mathsf{BucketGuess}_{\beta}$, the attacker has access to the bucket identifier, and therefore the advantage is computed as

$$\begin{aligned}\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta}(q) &= \sum_{u}\sum_{b} \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} P\left(W = w_i \wedge U = u \wedge B = b\right) \\ &= \sum_{u}\sum_{b} \max_{\substack{(u_1,w_1),\ldots,(u_q,w_q)\\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{P\left(W = w_i \wedge U = u\right)}{|\beta((u,w_i))|}\end{aligned} \tag{3.2}$$

The second equation follows because for $b \in \beta((u,w))$, each bucket in $\beta(w)$ is equally likely to be chosen, so

$$\Pr\left[B = b \mid W = w \wedge U = u\right] = \frac{1}{|\beta((u,w))|}.$$

The joint distribution of usernames and passwords is hard to model. To simplify the equations, we divide the users targeted by the attacker into two groups: *compromised* (users whose previously compromised accounts are available to the attacker) and *uncompromised* (users for which the attacker has no information other than their usernames).

We assume there is no direct correlation between the username and password.[3] Therefore, an attacker cannot use the knowledge of only the username to tailor guesses. This means that in the uncompromised setting, we assume $\Pr[W = w \mid U = u] = P(W = w)$. Assuming independence of usernames and passwords, we define in the uncompromised setting

$$\lambda_q = \mathsf{Adv}^{\mathsf{gs}}(q) = \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} P(W = w_i)\,. \tag{3.3}$$

We give analytical (using Equations 3.2 and 3.3) and empirical analysis of security in this setting, and show that the security of uncompromised users is impacted by existing C3 schemes much more than that of compromised users.

In the compromised setting, the attacker can use the username to find other leaked passwords associated with that user, which then can be used to tailor guesses [104, 127]. Analytical bounds on the compromised setting (using Equations 3.1 and 3.2) are less informative, so we evaluate this setting empirically in Section 3.6.

Bandwidth. The bandwidth required for a bucketization scheme is determined by the size of the buckets. The maximum size of the buckets can be determined using a balls-and-bins approach [35], assuming the client picks a bucket randomly from the possible set of buckets $\beta(s)$ for a credential s, and $\beta(s)$ also

[3]Though prior work [90, 127] suggests knowledge of the username can improve efficacy of guessing passwords, the improvement is minimal. See Appendix B.1 for more on this.

maps s to a random set of buckets. In total $m = \sum_{s \in \tilde{\mathcal{S}}} |\beta(s)|$ credentials (balls) are "thrown" into $n = |\mathcal{B}|$ buckets. If $m > |\mathcal{B}| \cdot \log |\mathcal{B}|$, then standard results [35] give that the maximum number of passwords in a bucket is less than $\frac{m}{n} \cdot \left(1 + \sqrt{\frac{n \log n}{m}}\right) \leq 2 \cdot \frac{m}{n}$, with very high probability $1 - o(1)$. We will use this formula to compute an upper bound on the bandwidth requirement for specific bucketization schemes.

For HPB schemes, each credential will be mapped to a random bucket if we assume that the hash function acts as a random oracle. For FSB, since we only randomly choose the first bucket and map a credential to a range of buckets starting with the first one, it is not clear that the set of buckets a credential is mapped to is random. We also show empirically that these bounds hold for the C3 schemes.

3.4 Hash-prefix-based Bucketization

Hash-prefix-based bucketization (HPB) schemes are a simple way to divide the credentials stored by the C3 server. In this type of C3 scheme, a prefix of the hash of the credential is used as the criteria to group the credentials into buckets — all credentials that share the same hash-prefix are assigned to the same bucket. The total number of buckets depends on l, the length of the hash-prefix. The number of credentials in the buckets depends on both l and $|\tilde{\mathcal{S}}|$. We will use $\mathsf{H}^{(l)}(\cdot)$ to denote the function that outputs the l-bit prefix of the hash $\mathsf{H}(\cdot)$. The client shares the hash prefix of the credential they wish to check with the server. While a smaller hash prefix reveals less information to the server about the user's password, it also increases the size of each bucket held by the server, which in turn increases the bandwidth overhead.

Hash-prefix-based bucketization is currently being used for credential checking in industry : HIBP [8] and GPC [111]. We introduce a new HPB protocol called IDB that achieves zero security loss for any query budget. Below we will discuss the design details of these three C3 protocols.

HIBP [8]. HIBP uses HPB bucketization to provide a password-only C3 service. They do not provide compromised username-password checking. HIBP maintains a database of leaked passwords, which contains more than 501 million passwords [8]. They use the SHA1 hash function, with prefix length $l = 20$; the leaked dataset is *partitioned* into 2^{20} buckets. The prefix length is chosen to ensure no bucket is too small or too big. With $l = 20$, the smallest bucket has 381 passwords, and the largest bucket has 584 passwords [34] . This effectively makes the user's password k-anonymous. However, k-anonymity provides limited protection, as shown by numerous prior works [92, 100, 135] and by our security evaluation.

The passwords are hashed using SHA1 and indexed by their hash prefix for fast retrieval. A client computes the SHA1 hash of their password w and queries HIBP with the 20-bit prefix of the hash; the server responds with all the hashes that share the same 20-bit prefix. The client then checks if the full SHA1 hash of w is present among the set of hashes sent by the server. This is a weak form of PSI that does not hide the leaked passwords from the client — the client learns the SHA1 hash of the leaked passwords and can perform brute force cracking to recover those passwords.

HIBP justifies this design choice by observing that passwords in the server side leaked dataset are publicly available for download on the Internet. Therefore, HIBP lets anyone download the hashed passwords and usernames. This can be useful for parties who want to host their own leak checking service without relying on HIBP. However, keeping the leaked dataset up-to-date can be challenging, making a third-party C3 service preferable.

HIBP trades server side privacy for protocol simplicity. The protocol also allows utilization of caching on content delivery networks (CDN), such as Cloudflare.[4] Caching helps HIBP to be able to serve 8 million requests a day with 99% cache hit rate (as of August 2018) [33]. The human-chosen password distribution is "heavy-headed", that is a small number of passwords are chosen by a large number of users. Therefore, a small number of passwords are queried a large number of times, which in turn makes CDN caching much more effective.

GPC [111,121]. Google provides a username-password C3 service, called Password Checkup (GPC). The client — a browser extension — computes the hash of the username and password together using the Argon2 hash function (configured to use a single thread, 256 MB of memory, and a time cost of three) with the first $l = 16$ bits to determine the bucket identifier. After determining the bucket, the client engages in a private set intersection (PSI) protocol with the server. The full algorithm is given in Figure 3.5. GPC uses a computationally expensive hash function to make it more difficult for an adversary to make a large number of queries to the server.

GPC uses an OPRF-based PSI protocol [121]. Let $F_a(x)$ be a function that first calls the hash function H on x, then maps the hash output onto an elliptic curve

[4]`https://www.cloudflare.com/`

Precomputation by C3 Server
Let $\tilde{\mathcal{S}} = \{(u_1, w_1), \ldots, (u_N, w_N)\}$
$\forall j \in [0, \ldots, 2^l - 1]$
$\mathbf{z}_j \leftarrow \{F_\kappa(u_i \| w_i) \mid \mathsf{H}^{(l)}(u \| w) = j)\}$
$\boxed{\mathbf{z}_j \leftarrow \{F_\kappa(u_i \| w_i) \mid \mathsf{H}^{(l)}(u) = j)\}}$

Client		**C3 server**
Input: (u, w)		Input: $\kappa, \mathbf{z}$
$r \leftarrow_{\$} \mathbb{Z}_q$		
$x \leftarrow F_r(u \| w)$		
$b \leftarrow \mathsf{H}^{(l)}(u \| w)$		
$\boxed{b \leftarrow \mathsf{H}^{(l)}(u)}$	$\xrightarrow{x,b}$	
	$\xleftarrow{y,\mathbf{z}_b}$	$y = x^\kappa$
$\tilde{x} \leftarrow y^{\frac{1}{r}}$		
Return $\tilde{x} \in \mathbf{z}_b$		

Figure 3.5: Algorithms for GPC, and the change in IDB given in the box. $F_{(\cdot)}(\cdot)$ is a PRF.

point, and finally, exponentiates the elliptic curve point (using elliptic curve group operations) to the power a. Therefore it holds that $(F_a(x))^b = F_{ab}(x)$.

The server has a secret key κ which it uses to compute the values $y_i = F_\kappa(u_i \| w_i)$ for each (u_i, w_i) pair in the breach dataset. The client shares with the server the bucket identifier b and the PRF output $x = F_r(u \| w)$, for some randomly sampled r. The server returns the bucket $\mathbf{z}_b = \{y_i \mid \mathsf{H}(u_i \| w_i) = b\}$ and $y = x^\kappa$. Finally, the client completes the OPRF computation by computing $\tilde{x} = y^{\frac{1}{r}} = F_\kappa(u \| w)$, and checking if $\tilde{x} \in \mathbf{z}_b$.

The GPC protocol is significantly more complex than HIBP, and it does not allow easy caching by CDNs. However, it provides secrecy of server-side leaked data — the best case attack is to follow the protocol to brute-force check if a password is present in the leak database.

Bandwidth. HPB assigns each credential to only one bucket; therefore, $m = \sum_{w \in \tilde{\mathcal{S}}} |\beta(w)| = |\tilde{\mathcal{S}}| = N$. The total number of buckets is $n = 2^l$. Following the discussion from Section 3.3, the maximum bandwidth for a HPB C3 service should be no more than $2 \cdot \frac{m}{n} = 2 \cdot \frac{N}{2^l}$.

We experimentally verified bandwidth usage, and the sizes of the buckets for HIBP, GPC, and IDB are given in Section 3.7.

Security. HPB schemes like HIBP and GPC expose a prefix of the user's password (or username-password pair) to the server. As discussed earlier, we assume the attacker knows the username of the target user. In the uncompromised setting — where the user identifier does not appear in the leaked data available to the attacker, we show that giving the attacker the hash-prefix with a guessing budget of q queries is equivalent to giving as many as $q \cdot |\mathcal{B}|$ queries (with no hash-prefix) to the attacker. As a reminder, $|\mathcal{B}|$ is the number of buckets. For example, consider a C3 scheme that uses a 5-character hash prefix as a bucket identifier (2^{20} buckets). If an attacker has 10 guesses to figure out a password, then given a bucket identifier, they can eliminate any guesses on their list that don't belong in that bucket. If their original guesses are distributed equally across all buckets, then knowing the 5-character hash prefix can help them get through around $q \cdot 2^{20}$ of those guesses.

Theorem 1. *Let $\beta_{\text{HPB}} : \mathcal{S} \mapsto \mathcal{B}$ be the bucketization scheme that, for a credential $s \in \mathcal{S}$, chooses a bucket that is a function of $\mathsf{H}^{(l)}(s)$, where s contains the user's password. The advantage of an attacker in this setting against previously uncompromised users is*

$$\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{HPB}}}(q) \leq \mathsf{Adv}^{\text{gs}}(q \cdot |\mathcal{B}|) \; .$$

Proof: First, note that $|\beta_{\mathrm{HPB}}(s)| = 1$, for any input s, as every password is assigned to exactly one of the buckets. Following the discussion from Section 3.3, assuming independence of usernames and passwords in the uncompromised setting, we can compute the advantage against game $\mathsf{BucketGuess}$ as

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta_{\mathrm{HPB}}}(q) = \sum_{b \in \mathcal{B}} \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} P\left(W = w_i\right) \leq \mathsf{Adv}^{\mathsf{gs}}(q \cdot |\mathcal{B}|)\,.$$

We relax the $\alpha(b)$ notation to denote set of passwords (instead of username-password pairs) assigned to a bucket b. The inequality follows from the fact that each password is present in only one bucket. If we sum up the probabilities of the top q passwords in each bucket, the result will be at most the sum of the probabilities of the top $q \cdot |\mathcal{B}|$ passwords. Therefore, the maximum advantage achievable is $\mathsf{Adv}^{\mathsf{gs}}(q \cdot |\mathcal{B}|)$. ▮

Theorem 1 only provides an upper bound on the security loss. Moreover, for the compromised setting, the analytical formula in Equation (3.2) is not very informative. So, we use empiricism to find the effective security loss against compromised and uncompromised users. We report all security simulation results in Section 3.6. Notably, with GPC using a hash prefix length $l = 16$, an attacker can guess passwords of 59.7% of (previously uncompromised) user accounts in fewer than 1,000 guesses, over a 10x increase from the percent it can compromise without access to the hash prefix. (See Section 3.6 for more results.)

Identifier-based bucketization (IDB). As our security analysis and simulation show, the security degradation of HPB can be high. The main issue with those protocols is that the bucket identifier is a deterministic function of the user password. We give a new C3 protocol that uses HPB style bucketing, but based only on username. We call this identifier-based bucketization (IDB). IDB is defined for username-password C3 schemes.

IDB is a slight modification of the protocol used by GPC— we use the hash-prefix of the username, $\mathsf{H}^{(l)}(u)$, instead of the hash-prefix of the username-password combination, $\mathsf{H}^{(l)}(u \,\|\, w)$, as a bucket identifier. The scheme is described in Figure 3.5, using the changes in the boxed code. The bucket identifier is computed completely independently of the password (assuming the username is independent of the password). Therefore, the attacker gets no additional advantage by knowing the bucket identifier.

Because IDB uses the hash-prefix of the username as the bucket identifier, two hash computations are required on the client side for each query (as opposed to one for GPC). With most modern devices, this is not a significant computing burden, but the protocol latency may be impacted, since we use a slow hash (Argon2) for hashing both the username and the password. We show experimentally how the extra hash computation affects the latency of IDB in Section 3.7.

Since in IDB, the bucket identifier does not depend on the user's password, the conditional probability of the password given the bucket identifier remains the same as the probability without knowing the bucket identifier. As a result, exposing the bucket identifier does not lead to security loss.

Theorem 2. *With the* IDB *protocol, for all* $q \geq 0$

$$\mathsf{Adv}^{\text{b-gs}}_{\text{IDB}}(q) = \mathsf{Adv}^{\text{gs}}(q).$$

Proof: Because the IDB bucketization scheme does not depend on the password, $\Pr[B = b \mid W = w \wedge U = u] = \Pr[B = b \mid U = u]$.

We can upper bound the success rate of an adversary in the $\mathsf{BucketGuess}_{\mathrm{IDB}}$ game by

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\mathrm{IDB}}(q)\\
&= \sum_{u}\sum_{b} \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} P(W = w_i \wedge U = u) \cdot \Pr[B = b \mid U = u]\\
&= \sum_{u}\left(\sum_{b} \Pr[B = b \mid U = u]\right) \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} P(W = w_i \wedge U = u)\\
&= \mathsf{Adv}^{\mathsf{gs}}(q)
\end{aligned}$$

The first step follows from independence of password and bucket choice, and the third step is true because there is only one bucket for each username. ▮

We would like to note, though IDB reveals nothing about the password, learning the username becomes easier (compared to GPC) — an attacker can narrow down the potential users after seeing the bucket identifier. While this can be concerning for user's privacy, we believe the benefit of not revealing anything about the user's password outweighs the risk.

Unfortunately, IDB does not work for the password-only C3 setting because it requires that the server store username-password pairs. In the next section we introduce a more secure password-only C3 scheme.

3.5 Frequency-Smoothing Bucketization

In the previous section we showed how to build a username-password C3 service that does not degrade security. However, many services, such as HIBP, only provide a password-only C3 service. HIBP does not store username-password pairs so, should the HIBP server ever get compromised, an attacker cannot use their leak database to mount credential stuffing attacks. Unfortunately, IDB cannot be extended in any useful way to protect password-only C3 services.

Therefore, we introduce a new bucketization scheme to build secure password-only C3 services. We call this scheme frequency-smoothing bucketization (FSB). FSB assigns a password to multiple buckets based on its probability — frequent passwords are assigned to many buckets. Replicating a password into multiple buckets effectively reduces the conditional probabilities of that password given a bucket identifier. We do so in a way that makes the conditional probabilities of popular passwords similar to those of unpopular passwords to make it harder for the attacker to guess the correct password. FSB, however, is only effective for non-uniform credential distributions, such as password distributions.[5] Therefore, FSB cannot be used to build a username-password C3 service.

Implementing FSB requires knowledge of the distribution of human-chosen passwords. Of course, obtaining precise knowledge of the password distribution can be difficult; therefore, we will use an estimated password distribution, denoted by $\hat{p}_s$. Another parameter of FSB is $\bar{q}$, which is an estimate of the attacker's query budget. We show that if the actual query budget $q \leq \bar{q}$, FSB has

[5]Usernames (e.g., emails) are unique for each users, so the distribution of usernames and username-password pairs are close to uniform.

```
$\beta_{\text{FSB}}(w)$ :
$\gamma \leftarrow \min\left\{|\mathcal{B}|, \left\lceil \frac{|\mathcal{B}|\cdot\hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil\right\}$
$s \leftarrow f(w)$
If $s + \gamma < |\mathcal{B}|$ then
    $r \leftarrow [s, s + \gamma - 1]$
Else
    $r \leftarrow [0, s+\gamma-1 \mod |\mathcal{B}|]$
    $r \leftarrow r \cup [s, |\mathcal{B}| - 1]$
Return $r$
```

```
$\tilde{\alpha}_{\text{FSB}}(b)$ :
/* returns $\{w \in \tilde{\mathcal{S}} \mid b \in \beta(w)\}$ */
$A \leftarrow \mathcal{W}_{\bar{q}}$
For $w \in \tilde{\mathcal{S}} \setminus \mathcal{W}_{\bar{q}}$ do
    If $b \in \beta_{\text{FSB}}(w)$ then
        $A \leftarrow A \cup \{w\}$
return $A$
```

Figure 3.6: **Bucketizing function** β_{FSB} for assigning passwords to buckets in FSB. Here $\hat{p}_s$ is the distribution of passwords; $\mathcal{W}_{\bar{q}}$ is the set of top-$\bar{q}$ passwords according to $\hat{p}_s$; $\mathcal{B}$ is the set of buckets; f is a hash function $f\colon W \mapsto \mathbb{Z}_{|B|}$; $\tilde{\mathcal{S}}$ is the set of passwords hosted by the server.

zero security loss. Larger $\bar{q}$ will provide better security; however, it also means more replication of the passwords and larger bucket sizes. So, $\bar{q}$ can be tuned to balance between security and bandwidth. Below we will give the two main algorithms of the FSB scheme: β_{FSB} and $\tilde{\alpha}_{\text{FSB}}$, followed by a bandwidth and security analysis.

Bucketizing function (β_{FSB}**).** To map passwords to buckets, we use a hash function $f : \mathcal{W} \mapsto \mathbb{Z}_{|\mathcal{B}|}$. The algorithm for bucketization $\beta_{\text{FSB}}(w)$ is given in Figure 3.6. The parameter $\bar{q}$ is used in the following way: β replicates the most probable $\bar{q}$ passwords, $\mathcal{W}_{\bar{q}}$, across all $|\mathcal{B}|$ buckets. Each of the remaining passwords are replicated proportional to their probability. A password w with probability $\hat{p}_s(w)$ is replicated exactly $\gamma = \left\lceil \frac{|\mathcal{B}|\cdot\hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil$ times, where $w_{\bar{q}}$ is the $\bar{q}^{th}$ most likely password. Exactly which buckets a password is assigned to are determined using the hash function f. Each bucket is assigned an identifier between $[0, |\mathcal{B}| - 1]$. A password w is assigned to the buckets whose identifiers fall in the range $[f(w), f(w) + \gamma - 1]$. The range can wrap around. For example, if $f(w) + \gamma > |\mathcal{B}|$, then the password is assigned to the buckets in the range $[0, f(w) + \gamma - 1 \mod |\mathcal{B}|]$ and $[f(w), |\mathcal{B}| - 1]$.

Bucket retrieving function ($\tilde{\alpha}$). Retrieving passwords assigned to a bucket is challenging in FSB. An inefficient — linear in N — implementation of $\tilde{\alpha}$ is given in Figure 3.6. Storing the contents of each bucket separately is not feasible, since the number of buckets in FSB can be very large, $|\mathcal{B}| \approx N$. To solve the problem, we utilize the structure of the bucketizing procedure where passwords are assigned to buckets in continuous intervals. This allows us to use an interval tree [10] data structure to store the intervals for all of the passwords. Interval trees allow fast queries to retrieve the set of intervals that contain a queried point (or interval) — exactly what is needed to instantiate $\tilde{\alpha}$.

This efficiency comes with increased storage cost: storing N entries in an interval tree requires $\mathcal{O}(N \log N)$ storage. The tree can be built in $\mathcal{O}(N \log N)$ time, and each query takes $\mathcal{O}(\log N + |\tilde{\alpha}(b)|)$ time. The big-O notation only hides small constants.

Estimating password distributions. To construct the bucketization algorithm for FSB, the server needs an estimate of the password distribution p_w. This estimate will be used by both the server and the client to assign passwords to buckets. One possible estimate is the histogram of the passwords in the leaked data $\tilde{\mathcal{S}}$. Histogram estimates are typically accurate for popular passwords, but such estimates are not complete — passwords that are not in the leaked dataset will have zero probability according to this estimate. Moreover, sending the histogram over to the client is expensive in terms of bandwidth, and it may leak too much information about the dataset. We also considered password strength meters, such as zxcvbn [131] as a proxy for a probability estimate. However, this estimate turned out to be too coarse for our purposes. For example, more than 10^5 passwords had a "probability" of greater than 10^{-3}.

We build a 3-gram password model $\hat{p}_n$ using the leaked passwords present in $\tilde{\mathcal{S}}$. Markov models or n-gram models are shown to be effective at estimating human-chosen password distributions [91], and they are very fast to train and run (unlike neural network based password distribution estimators, such as [97]). However, we found the n-gram model assigns very low probabilities to popular passwords. The sum of the probabilities of the top 1,000 passwords as estimated by the 3-gram model is only 0.032, whereas those top 1,000 passwords are chosen by 6.5% of users.

We therefore use a combined approach that uses a histogram model for the popular passwords and the 3-gram model for the rest of the distribution. Such combined techniques are also used in practice for password strength estimation [97, 131]. Let $\hat{p}_s$ be the estimated password distribution used by FSB. Let $\hat{p}_h$ be the distribution of passwords implied by the histogram of passwords present in $\tilde{\mathcal{S}}$. Let $\tilde{\mathcal{S}}_t$ be the set of the t most probable passwords according to $\hat{p}_h$. We used $t = 10^6$. Then, the final estimate is

$$\hat{p}_s(w) = \begin{cases} \hat{p}_h(w) & \text{if } w \in \tilde{\mathcal{S}}_t \text{ ,} \\ \hat{p}_n(w) \cdot \frac{1-\sum_{\tilde{w}\in\tilde{\mathcal{S}}_t} \hat{p}_h(w)}{1-\sum_{\tilde{w}\in\tilde{\mathcal{S}}_t} \hat{p}_n(w)} & \text{otherwise.} \end{cases}$$

Note that instead of using the 3-gram probabilities directly, we multiply them by a normalization factor that allows $\sum_w \hat{p}(w) = 1$, assuming that the same is true for the distributions $\hat{p}_h$ and $\hat{p}_n$.

Bandwidth. We use the formulation provided in Section 3.3 to compute the bandwidth requirement for FSB. In this case, $m = |\mathcal{B}| \cdot \bar{q} + \frac{|\mathcal{B}|}{\hat{p}_s(w_{\bar{q}})} + N$, and

$n = |\mathcal{B}|$. Therefore, the maximum size of a bucket is with high probability less than $2 \cdot \left(\bar{q} + \frac{1}{\hat{p}_s(w_{\bar{q}})} + \frac{N}{|\mathcal{B}|}\right)$. The details of this analysis are given in Appendix B.2.

In practice, we can choose the number of buckets to be such that $|\mathcal{B}| = N$. Then, the number of passwords in a bucket depends primarily on the parameter $\bar{q}$. Note, bucket size increases with $\bar{q}$.

Security analysis. We show that there is no security loss in the uncompromised setting for FSB when the actual number of guesses q is less than the parameter $\bar{q}$ and the estimate $\hat{p}$ is accurate. We also give a bound for the security loss when q exceeds $\bar{q}$.

Theorem 3. *Let FSB be a frequency based bucketization scheme that ensures* $\forall w \in \mathcal{W}$, $|\beta_{\text{FSB}}(w)| = \min\left\{|\mathcal{B}|, \left\lceil \frac{|\mathcal{B}|\cdot\hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil\right\}$. *Assuming that the distribution estimate* $\hat{p}_s = p_w$, *then for the uncompromised users,*

(1) $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) = \mathsf{Adv}^{\text{gs}}(q)$ *for* $q \le \bar{q}$, *and*

(2) *for* $q > \bar{q}$,

$$\frac{\lambda_q - \lambda_{\bar{q}}}{2} \le \Delta_q \le (q - \bar{q}) \cdot \hat{p}_s(w_{\bar{q}}) - (\lambda_q - \lambda_{\bar{q}})$$

Recall that the probabilities λ_q are defined in Equation (3.3). We include the full proof for Theorem 3 in Appendix B.3. Intuitively, since the top q passwords are repeated across all buckets, having a bucket identifier does not allow an attacker to more easily guess these q passwords. Moreover, the conditional probability of these q passwords given the bucket is greater than that of any other password in the bucket. Therefore, the attacker's best choice is to guess the top q passwords, meaning that it does not get any additional advantage when $q \le \bar{q}$, leading to part (1) of the theorem.

The proof of part (2) follows from the upper and lower bounds on the number of buckets each password beyond the top q is placed within. The bounds we prove show that the additional advantage in guessing the password in q queries is less than the number of additional queries times the probability of the $\bar{q}^{th}$ password and at least half the difference in the guessing probabilities λ_q and $\lambda_{\bar{q}}$.

Note that this analysis of security loss is based on the assumption that the FSB scheme has access to the precise password distribution, $\hat{p}_s = p_w$. We empirically analyze the security loss in Section 3.6 for $\hat{p}_s \neq p_w$, in both the compromised and uncompromised settings.

3.6 Empirical Security Evaluation

In this section we empirically evaluate and compare the security loss for different password-only C3 schemes we have discussed so far — hash-prefix-based bucketization (HPB) and frequency-smoothing bucketization (FSB).

We focus on known-username attacks (KUA), since in many deployment settings a curious (or compromised) C3 server can figure out the username of the querying user. We separate our analysis into two settings: previously *compromised* users, where the attacker has access to one or more existing passwords of the target user, and previously *uncompromised* users, where no password corresponding to the user is known to the attacker (or present in the breached data).

We also focus on what the honest-but-curious C3 server can learn from knowing the bucket. In our experiment, we show the success rate of an ad-

	$\tilde{\mathcal{S}}$	T	$T \cap \tilde{\mathcal{S}}$	T_{sp}	$T_{\text{sp}} \cap \tilde{\mathcal{S}}$
# users	901.4	12.9	5.9 (46%)	8.4	3.9 (46%)
# passwords	435.9	8.9	5.7 (64%)	6.7	3.9 (59%)
# user-pw pairs	1,316.6	13.1	3.2 (24%)	8.5	2.0 (23%)

Figure 3.7: Number of entries (in millions) in the breach dataset $\tilde{\mathcal{S}}$, test dataset T, and the site-policy test subset T_{sp}. Also reported are the intersections (of users, passwords, and user-password pairs, separately) between the test dataset entries and the whole breach dataset that the attacker has access to. The percentage values refer to the fraction of the values in each test set that also appear in the intersections.

versary that knows the exact leak dataset used by the server. We expect that an adversary that doesn't know the exact leak dataset will have slightly lower success rates.

First we will look into the unrestricted setting where no password policy is enforced, and the attacker and the C3 server have the same amount of information about the password distribution. In the second experiment, we analyze the effect on security of giving the attacker more information compared to the C3 server (defender) by having a password policy that the attacker is aware of but the C3 server is not.

Password breach dataset. We used the same breach dataset used in [104]. The dataset was derived from a previous breach compilation [42] dataset containing about 1.4 billion username-password pairs. We chose to use this dataset rather than, for example, the password breach dataset from HIBP, because it contains username-password pairs.

We cleaned the data by removing non-ASCII characters and passwords longer than 30 characters. We also combined username-password pairs with the same case-insensitive username, and we removed users with over 1,000 passwords, as they didn't seem to be associated to real accounts. The authors of [104] also joined accounts with similar usernames and passwords using a method they called the *mixed method*. We joined the dataset using the same mixed method, but we also kept the usernames with only one email and password.

The final dataset consists of about 1.32 billion username-password pairs.[6] We remove 1% of username-password pairs to use as test data, denoted as T. The remaining 99% of the data is used to simulate the database of leaked credentials $\tilde{\mathcal{S}}$. For the experiments with an enforced password policy, we took the username-password pairs in T that met the requirements of the password policy to create T_{sp}. We use T_{sp} to simulate queries from a website which only allows passwords that are at least 8 characters long and are not present in Twitter's list of banned passwords [13]. For all attack simulations, the target user-password pairs are sampled from the test dataset T (or T_{sp}).

In Figure 3.7, we report some statistics about T, T_{sp}, and $\tilde{\mathcal{S}}$. Notably, 5.9 million (46%) of the users in T are also present in $\tilde{\mathcal{S}}$. Among the username-password pairs, 3.2 million (24%) of the pairs in T are also present in $\tilde{\mathcal{S}}$. This means an attacker will be able to compromise about half of the previously compromised accounts trivially with credential stuffing. In the site-policy enforced test data T_{sp}, a similar proportion of the users (46%) and username-password pairs (23%) are also present in $\tilde{\mathcal{S}}$.

Experiment setup. We want to understand the impact of revealing a bucket identifier on the security of uncompromised and compromised users separately. As we can see from Figure 3.7, a large proportion of users in T are also present in $\tilde{\mathcal{S}}$. We therefore split T into two parts: one with only username-password

[6]Note, there are duplicate username-password pairs in this dataset.

pairs from compromised users (users with at least one password present in $\tilde{\mathcal{S}}$), T_{comp}, and another with only pairs from uncompromised users (users with no passwords present in $\tilde{\mathcal{S}}$), T_{uncomp}. We generate two sets of random samples of 5,000 username-password pairs, one from T_{comp}, and another from T_{uncomp}. We chose 5,000 because this number of samples led to a low standard deviation (as reported in Figure 3.8). For each pair (u, w), we run the games Guess and BucketGuess as specified in Figure 3.4. We record the results for guessing budgets of $q \in \{1,\, 10,\, 10^2,\, 10^3\}$. We repeat each of the experiments 5 times and report the averages in Figure 3.8.

For HPB, we compared implementations using hash prefixes of lengths $l \in \{12, 16, 20\}$. We use the SHA256 hash function with a salt, though the choice of hash function does not have a noticeable impact on the results.

For FSB, we used interval tree data structures to store the leaked passwords in $\tilde{\mathcal{S}}$ for fast retrieval of $\tilde{\alpha}(b)$. We used $|\mathcal{B}| = 2^{30}$ buckets, and the hash function f is set to $f(x) = \mathsf{H}^{(30)}(x)$, the 30-bit prefix of the (salted) SHA256 hash of the password.

Attack strategy. The attacker's goal is to maximize its success in winning the games Guess and BucketGuess. In Equation (3.1) and Equation (3.2) we outline the advantage of attackers against Guess and BucketGuess, and thereby specify the best strategies for attacks. Guess denotes the baseline attack success rate in a scenario where the attacker does not have access to bucket identifiers corresponding to users' passwords. Therefore the best strategy for the attacker $\mathcal{A}$ is to output the q most probable passwords according to its knowledge of the password distribution.

The optimal attack strategy for $\mathcal{A}'$ in BucketGuess will be to find a list of passwords according to the following equation,

$$\underset{\substack{w_1,\ldots,w_q \\ b\in\beta((u,w_i))}}{\arg\max} \sum_{i=1}^{q} \frac{\Pr\left[W = w_i \mid U = u\right]}{|\beta((u,w_i))|},$$

where the bucket identifier b and user identifier u are provided to the attacker. This is equivalent to taking the top-q passwords in the set $\alpha(b)$ ordered by $\Pr\left[W = w \mid U = u\right]/|\beta((u,w))|$.

We compute the list of guesses outputted by the attacker for a user u and bucket b in the following way. For the compromised users, i.e., if $(u,\cdot) \in \tilde{\mathcal{S}}$, the attacker first considers the passwords known to be associated to that user and the list of 10^4 targeted guesses generated based on the credential tweaking attack introduced in [104]. If any of these passwords belong to $\alpha(b)$ they are guessed first. This step is skipped for uncompromised users.

For the remaining guesses, we first construct a list of candidates L consisting of all 436 million passwords present in the breached database $\tilde{\mathcal{S}}$ sorted by their frequencies, followed by 500×10^6 passwords generated from the 3-gram password distribution model $\hat{p}_n$. Each password w in L is assigned a weight $\hat{p}_s(w)/|\beta((u,w))|$ (See Section 3.5 for details on $\hat{p}_s$ and $\hat{p}_n$). The list L is pruned to only contain unique guesses. Note L is constructed independent of the username or bucket identifier, and it is reordered based on the weight values. Therefore, it is constructed once for each bucketization strategy. Finally, based on the bucket identifier b, the remaining guesses are chosen from $\{\alpha(b)\cap(u,w) \mid w \in L\}$ in descending order of weight.

For the HPB implementation, each password is mapped to one bucket, so $|\beta(w)| = 1$ for all w. For FSB, $|\beta(\cdot)|$ can be calculated using the equation in Theorem 3.

Since we are estimating the values to be used in the equation, the attack is no longer optimal. However, the attack we use still performs quite well against existing C3 protocols, which already shows that they leak too much information. An optimal attack can only perform better.

Protocol	Params	Bucket size		**Uncompromised**			
		Avg	max	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$
Baseline	N/A	N/A	N/A	0.7 (±0.1)	1.5 (±0.1)	2.9 (±0.3)	5.8 (±0.4)
HPB	$l = 20$[‡]	413	491	32.9 (±0.5)	49.5 (±0.3)	62.5 (±0.4)	71.1 (±0.5)
	$l = 16$[†]	6602	6891	17.9 (±0.5)	33.4 (±0.6)	47.3 (±0.3)	59.7 (±0.2)
	$l = 12$	105642	106668	8.2 (±0.4)	17.5 (±0.6)	30.7 (±0.6)	44.4 (±0.4)
FSB	$\bar{q} = 1$	83	122	0.7 (±0.1)	4.7 (±0.4)	69.8 (±0.5)	71.1 (±0.5)
	$\bar{q} = 10$	852	965	0.7 (±0.1)	1.5 (±0.1)	5.3 (±0.3)	70.8 (±0.5)
	$\bar{q} = 10^2$	6299	6602	0.7 (±0.1)	1.5 (±0.1)	2.9 (±0.3)	8.0 (±0.4)
	$\bar{q} = 10^3$	25191	25718	0.7 (±1.0)	1.5 (±0.1)	2.9 (±0.3)	5.8 (±0.4)

Protocol	Params	Bucket size		**Compromised**			
		Avg	max	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$
Baseline	N/A	N/A	N/A	41.1 (±0.4)	51.1 (±0.8)	53.3 (±0.9)	55.7 (±1.0)
HPB	$l = 20$[‡]	413	491	67.3 (±0.8)	74.5 (±0.6)	79.4 (±0.6)	82.9 (±0.4)
	$l = 16$[†]	6602	6891	61.1 (±0.9)	67.4 (±0.8)	73.6 (±0.6)	78.2 (±0.7)
	$l = 12$	105642	106668	56.3 (±1.0)	60.8 (±1.0)	66.5 (±0.8)	72.3 (±0.6)
FSB	$\bar{q} = 1$	83	122	53.7 (±0.9)	55.7 (±0.9)	82.6 (±0.4)	83.0 (±0.4)
	$\bar{q} = 10$	852	965	52.8 (±0.9)	54.2 (±1.0)	56.0 (±0.9)	83.0 (±0.4)
	$\bar{q} = 10^2$	6299	6602	51.9 (±0.8)	53.8 (±0.9)	54.8 (±1.0)	57.1 (±1.0)
	$\bar{q} = 10^3$	25191	25718	51.4 (±0.9)	53.2 (±0.9)	54.7 (±1.0)	55.9 (±0.9)

[‡] HIBP uses $l = 20$ for its password-only C3 service. [†] GPC uses $l = 16$ for username-password C3 service.

Figure 3.8: Comparison of attack success rate given q queries on different password-only C3 settings. All success rates are in percent (%) of the total number of samples (25,000). The standard deviations across the 5 independent experiments of 5,000 samples each are given in the parentheses. Bucket size, the number of passwords associated to a bucket, is measured on a random sample of 10,000 buckets.

Results. We report the success rates of the attack simulations in Figure 3.8. The baseline success rate (first row) is the advantage $\mathsf{Adv}^{\mathsf{gs}}$, computed using the same attack strategy stated above except with no information about the bucket identifier. The following rows record the success rate of the attack for HPB and FSB with different parameter choices. The estimated security loss (Δ_q) can be calculated by subtracting the baseline success rate from the HPB and FSB attack success rates.

The security loss from using HPB is large, especially for previously uncompromised users. Accessibility to the $l = 20$-bit hash prefix, used by HIBP [8], allows an attacker to compromise 32.9% of previously uncompromised users in just one guess. In fewer than 10^3 guesses, that attacker can compromise more than 70% of the accounts (12x more than the baseline success rate with 10^3 guesses). Google Password Checkup (GPC) uses $l = 16$ for its username-password C3 service. Against GPC, an attacker only needs 10 guesses per account to compromise 33% of accounts. Reducing the prefix length l can decrease the attacker's advantage. However, that would also increase the bucket size. As we see for $l = 12$, the average bucket size is 105,642, so the bandwidth required to perform the credential check would be high.

FSB resists guessing attacks much better than HPB does. For $q \leq \bar{q}$ the attacker gets no additional advantage, even with the estimated password distribution $\hat{p}_s$. The security loss for FSB when $q > \bar{q}$ is much smaller than that of HPB, even with smaller bucket sizes. For example, the additional advantage over the baseline against FSB with $q = 100$ and $\bar{q} = 10$ is only 2.4%, despite FSB also having smaller bucket sizes than HPB with $l = 16$. Similarly for $\bar{q} = 100$,

$\Delta_{10^3} = 2.2\%$. This is because the conditional distribution of passwords given an FSB bucket identifier is nearly uniform, making it harder for an attacker to guess the correct password in the bucket $\alpha(b)$ in q guesses.

For previously compromised users — users present in $\tilde{\mathcal{S}}$ — even the baseline success rate is very high: 41% of account passwords can be guessed in 1 guess and 56% can be guessed in fewer than 1,000 guesses. The advantage is supplemented even further with access to the hash prefix. As per the guessing strategy, the attacker first guesses the leaked passwords that are both associated to the user and in $\alpha(b)$. This turns out to be very effective. Due to the high baseline success rate the relative increase is low; nevertheless, in total, an attacker can guess the passwords of 83% of previously compromised users in fewer than 1,000 guesses. For FSB, the security loss for compromised users is comparable to the loss against uncompromised users for $q \leq \bar{q}$. Particularly for $\bar{q} = 10$ and $q = 100$, the attacker's additional success for a previously compromised user is only 2.7% higher than the baseline. Similarly, for $\bar{q} = 100$ an attacker gets at most 1.4% additional advantage for a guessing budget of q=1,000. Interestingly, FSB performs significantly worse for compromised users compared to uncompromised users for $q = 1$. This is because the FSB bucketing strategy does not take into account targeted password distributions, and the first guess in the compromised setting is based on the credential tweaking attack.

In our simulation, previously compromised users made up around 46% of the test set. We could proportionally combine the success rates against uncompromised and compromised users to obtain an overall attack success rate. However, it is unclear what the actual proportion would be in the real world, so we choose not to combine results from the two settings.

Password policy experiment. In the previous set of experiments, we assumed that the C3 server and the attacker use the same estimate of the password distribution. To explore a situation in which the attacker has a better estimate of the password distribution than the C3 server, we simulated a website which enforces a password policy. We assume that the policy is known to the attacker but not to the C3 server.

For our sample password policy, we required that passwords have at least 8 characters and that they must not be on Twitter's banned password list [13]. The test samples are drawn from T_{sp}, username-password pairs from T where passwords follow this policy. The attacker is also given the ability to tailor their guesses to this policy. The server still stores all passwords in $\tilde{S}$, without regard to this policy. Notably, the FSB scheme relies on a good estimate of the password distribution to be effective in distributing passwords evenly across buckets. Its estimate, when compared to the distribution of passwords in T_{sp}, should be less accurate than it was in the regular simulation, when compared to the password distribution from T.

We chose the parameters $k = 16$ for HPB and $\bar{q} = 100$ for FSB, because they were the most representative of how the HPB and FSB bucketization schemes compare to each other. These parameters also lead to similar bucket sizes, with around 6,500 passwords per bucket. Overall, we see that the success rate of an attacker decreases in these simulations compared to the general experiments (without a password policy). This is because after removing popular passwords, the remaining set of passwords that we can choose from has higher entropy, and

Protocol	Uncompromised				Compromised			
	$q = 1$	10	10^2	10^3	$q = 1$	10	10^2	10^3
Baseline	0.1	0.5	1.3	3.4	42.2	49.0	49.8	51.1
HPB ($l = 16$)	12.6	25.9	36.3	48.9	54.6	59.9	65.9	70.3
FSB ($\bar{q} = 10^2$)	0.1	0.5	1.5	13.2	49.2	50.0	50.4	54.9

Figure 3.9: Attack success rate (in %) comparison for HPB with $l = 16$ (effectively GPC) and FSB with $\bar{q} = 10^2$ for password policy simulation. The first row records the baseline success rate $\mathsf{Adv}^{\mathsf{gs}}(q)$. There were 5,000 samples each from the uncompromised and compromised settings.

each password is harder to guess. FSB still defends much better against the attack than HPB does, even though the password distribution estimate used by the FSB implementation is quite inaccurate, especially at the head of the distribution. The inaccuracy stems from FSB assigning larger probability estimates to passwords that are banned according to the password policy.

We also see that due to the inaccurate estimate by the C3 server for FSB, we start to see some security loss for an adversary with guessing budget $q = 100$. In the general simulation, the password estimate $\hat{p}_s$ used by the server was closer to p, so we didn't have any noticeable security loss where $q \leq \bar{q}$.

3.7 Performance Evaluation

We implement the different approaches to checking compromised credentials and evaluate their computational overheads. For fair comparison, in addition to the algorithms we propose, FSB and IDB, we also implement HIBP and GPC with our breach dataset.

Setup. We build C3 services as serverless web applications that provide REST APIs. We used AWS Lambda [3] for the server-side computation and Amazon

DynamoDB [6] to store the data. The benefit of using AWS Lambda is it can be easily deployed as Lambda@Edge and integrated with Amazon's content delivery network (CDN), called CloudFront [5]. (HIBP uses Cloudflare as CDN to serve more than 600,000 requests per day [9].) We used Javascript to implement the server and the client side functionalities. The server is implemented as a Node-JS app. We provisioned the Lambda workers to have a maximum of 3 GB of memory. For cryptographic operations, we used a Node-JS library called Crypto [14].

For pre-processing and pre-computation of the data we used a desktop with an Intel Core i9 processor and 128 GB RAM. Though some of the computation (e.g., hash computations) can be expedited using GPUs, we did not use any for our experiment. We used the same machine to act as the client. The round trip network latency of the Lambda API from the client machine is about 130 milliseconds.

The breach dataset we used is the one described in Figure 3.7. It contains 436 million unique passwords and 1,317 million unique username-password pairs.

To measure the performance of each scheme, we pick 20 random passwords from the test set T and run the full C3 protocol with each one. We report the average time taken for each run in Figure 3.10. In the figure, we also give the breakdown of the time taken by the server and the client for different operations. The network latency had very high standard deviation (25%), though all other measurements had low ($< 1\%$) standard deviations compared to their mean values.

HIBP. The implementation of HIBP is the simplest among the four schemes. The set of passwords in $\tilde{S}$ is hashed using SHA256 and split into 2^{20} buckets based on the first 20 bits of the hash value (we picked SHA256 because we also used the same for FSB). Because the bucket sizes in HIBP are so small (< 500), each bucket is stored as a single value in a DynamoDB cell, where the key is the hash prefix. For larger leaked datasets, each bucket can be split into multiple cells. The client sends the 20 bit prefix of the SHA256 hash of their password, and the server responds with the corresponding bucket.

Among all the protocols HIBP is the fastest (but also weakest in terms of security). It takes only 220 ms on average to complete a query over WAN. Most of the time is spent in round-trip network latency and the query to DynamoDB. The only cryptographic operation on the client side is a SHA256 hash of the password, which takes less than 1 ms.

FSB. The implementation of FSB is more complicated than that of HIBP. Because we have more than 1 billion buckets for FSB and each password is replicated in potentially many buckets, storing all the buckets explicitly would require too much storage overhead. We use interval trees [10] to quickly recover the passwords in a bucket without explicitly storing each bucket. Each password w in the breach database is represented as an interval specified by $\beta_{\text{FSB}}(w)$. We stored each node of the tree as a separate cell in DynamoDB. We retrieved the intervals (passwords) intersecting a particular value (bucket identifier) by querying the nodes stored in DynamoDB. FSB also needs an estimate of the password distribution to get the interval range for a tree. We use $\hat{p}_s$ as described in Section 3.4. The description of $\hat{p}_s$ takes 8.9 MB of space that needs to be included as part of the client side code. This is only a one-time bandwidth cost during client installation. The client would then need to store the description to use.

The depth of the interval tree is $\log N$, where N is the number of intervals (passwords) in the tree. Since each node in the tree is stored as a separate key-value pair in the database, one client query requires $\log N$ queries to DynamoDB. To reduce this cost, we split the interval tree into r trees over different ranges of intervals, such that the i-th tree is over the interval $[(i-1) \cdot \lfloor|\mathcal{B}|/r\rfloor,\ i \cdot \lfloor|\mathcal{B}|/r\rfloor - 1]$. The passwords whose bucket intervals span across multiple ranges are present in all corresponding trees. We used $r = 128$, as it ensures each tree has around 4 million passwords, and the total storage overhead is less than 1% more than if we stored one large tree.

Each interval tree of 4 million passwords was generated in parallel and took 3 hours in our server. Each interval tree takes 400 MB of storage in DynamoDB, and in total 51 GB of space. FSB is the slowest among all the protocols, mainly due to multiple DynamoDB calls, which cumulatively take 273 ms (half of the total time, including network latency). This can be sped up by using a better implementation of interval trees on top of DynamoDB, such as storing a whole subtree in a DynamoDB cell instead of storing each tree node separately. We can also split the range of the range tree into more granular intervals to reduce each tree size. Nevertheless, as the round trip time for FSB is small (527 ms), we leave such optimization for future work. The maximum amount of memory used by the server is less than 91 MB during an API call.

Protocol	Client			Server		Total	Bucket
	Crypto	Server call	Comp	DB call	Crypto	time	size
HIBP	1	217	2	40	–	220	413
FSB	1	524	2	273	–	527	6,602
GPC	47	433	9	72	6	489	16,121
IDB	72	435	10	74	6	517	16,122

Figure 3.10: Time taken in milliseconds to make a C3 API call. The client and server columns contain the time taken to perform client side and server side operations respectively.

On the client side, the computational overhead is minimal. The client performs one SHA256 hash computation. The network bandwidth consumed for sending the bucket of hash values from the server takes on average 558 KB.

IDB and GPC. Implementations of IDB and GPC are very similar. We used the same platforms — AWS Lambda and DynamoDB — to implement these two schemes. All the hash computations used here are Argon2id with default parameters, since GPC in [111] uses Argon2. During precomputation, the server computes the Argon2 hash of each username-password pair and raises it to the power of the server's key κ. These values can be further (fast) hashed to reduce their representation size, which saves disk space and bandwidth. However, hashing would make it difficult to rotate server key. We therefore store the exponentiated Argon2 hash values in the database, and hash them further during the online phase of the protocol. The hash values are indexed and bucketized based on either $\mathsf{H}^{(l)}(u\|w)$ (for GPC) or $\mathsf{H}^{(l)}(u)$ (for IDB). We used $l = 16$ for both GPC and IDB, as proposed in [111].

We used the secp256k1 elliptic curve. The server (for both IDB and GPC) only performs one elliptic curve exponentiation, which on average takes 6 ms. The remaining time incurred is from network latency and calling Amazon DynamoDB.

On the client side, one Argon2 hash has to be computed for GPC and two for IDB. Computing the Argon2 hash of the username-password pairs takes on an average 20 ms on the desktop machine. We also tried the same Argon2 hash computation on a personal laptop (Macbook Pro), and it took 8 ms. In total, hashing and exponentiation takes 47 ms for GPC, and 72 ms (an additional 25 ms) for IDB. The cost of checking the bucket is also higher (compared to HIBP and FSB) due to larger bucket sizes.

IDB takes only 28 ms more time on average than GPC (due to one extra Argon2 hashing), while also leaking no additional information about the user's password. It is the most secure among all the protocols we discussed (should username-password pairs be available in the leak dataset), and runs in a reasonable time.

3.8 Deployment discussion

Here we discuss different ways C3 services can be used and associated threats that need to be considered. A C3 service can be queried while creating a password — during registration or password change — to ensure that the new password is not present in a leak. In this setting C3 is queried from a web server, and the client IP is potentially not revealed to the server. This, we believe, is a safer setting to use than the one we will discuss below.

In another scenario, a user can directly query a C3 service. A user can look for leaked passwords themselves by visiting a web site or using a browser plugin, such as 1Password [7] or Password Checkup [111]. This is the most prevalent use case of C3. For example, the client can regularly check with a C3 service to proactively safeguard user accounts from potential credential stuffing attacks.

However, there are several security concerns with this setting. Primarily, the client's IP is revealed to the C3 server in this setting, making it easier for the attacker to deanonymize the user. Moreover, multiple queries from the same user can lead to a more devastating attack. Below we give two new threat models that need to be considered for secure deployment of C3 services (where bucket identifiers depend on the password).

Regular password checks. A user or web service might want to regularly check their passwords with C3 services. Therefore, a compromised C3 server may learn multiple queries from the same user. For FSB the bucket identifier is chosen randomly, so knowing multiple bucket identifiers for the same password will help an attacker narrow down the password search space by taking an intersection of the buckets, which will significantly improve attack success.

We can mitigate this problem for FSB by derandomizing the client side bucket selection using a client side state (e.g., browser cookie) so the client always selects the same bucket for the same password. We let c be a random number chosen by the client and stored in the browser. To check a password w with the C3 server, the client always picks the j^{th} bucket from the range $\beta(w)$, where $j \leftarrow f(w\|c) \mod |\beta(w)|$.

This derandomization ensures queries from the same device are deterministic (after the c is chosen and stored). However, if the attacker can link queries of the same user from two different devices, the mitigation is ineffective. If the cookie is stolen from the client device, then the security of FSB is effectively reduced to that of HPB with similar bucket sizes.

Similarly, if an attacker can track the interaction history between a user and a C3 service, it can obtain better insight about the user's passwords. For example, if a user who regularly checks with a C3 service stops checking a particular bucket identifier, that could mean the associated password is possibly in the most up-to-date leaked dataset, and the attacker can use that information to guess the user's password(s).

Checking similar passwords. Another important issue is querying the C3 service with multiple correlated passwords. Some web services, like 1Password, use HIBP to check multiple passwords for a user. As shown by prior work, passwords chosen by the same user are often correlated [50, 104, 127]. An attacker who can see bucket identifiers of multiple correlated passwords can mount a stronger attack. Such an attack would require estimating the joint distribution over passwords. We present an initial analysis of this scenario in Appendix B.4.

3.9 Related Work

Private set intersection. The protocol task facing C3 services is private set membership, a special case of private set intersection (PSI) [64, 96]. The latter allows two parties to find the intersection between their private sets without revealing any additional information. Even state-of-the-art PSI protocols do not scale to the sizes needed for our application. For example, Kiss et al. [79] proposed an efficient PSI protocol for unequal set sizes based on oblivious pseudorandom functions (OPRF). It performs well for sets with millions of elements, but the bandwidth usage scales proportionally to the size of the leak dataset and so performance is prohibitive in our setting. Other efficient solutions to PSI [47, 81, 109, 110] have similarly prohibitive bandwidth usage.

Private information retrieval (PIR) [48] is another cryptographic primitive used to retrieve information from a server. Assuming the server's dataset is public, the client can use PIR to privately retrieve the entry corresponding to their password from the server. But in our setting we also want to protect the privacy of the dataset leak. Even if we relaxed that security requirement, the most advanced PIR schemes [32, 102] require exchanging large amounts of information over the network, so they are not useful for checking leaked passwords. PIR with two non-colluding servers can provide better security [57] than the bucketization-based C3 schemes, with communication complexity subpolynomial in the size of the leaked dataset. It requires building a C3 service with two servers guaranteed to not collude, which may be practical if we assume that the breached credentials are public information. However, with a dataset size of at least 1 billion credentials, the cost of one query is likely still too large to be practical.

Compromised credential checking. To the best of our knowledge, HIBP was the first publicly available C3 service. Junade Ali designed the current HIBP protocol which uses bucketization via prefix hashing to limit leakage. Google's Password Checkup extends this idea to use PSI, which minimizes the information about the leak revealed to clients. They also moved to checking username, password pairs.

Google's Password Checkup (GPC) was described in a paper by Thomas et al. [121], which became available to us after we began work on this paper. They introduced the design and implementation of GPC and report on measurements of its initial deployment. They recognized that their first generation protocol leaks some bits of information about passwords, but did not analyze the potential impact on password guessability. They also propose (what we call) the ID-based protocol as a way to avoid this leakage. Our paper provides further motivation for their planned transition to it.

Thomas et al. point out that password-only C3 services are likely to have high false positive rates. Our new protocol FSB, being in the password-only setting, inherits this limitation. That said, should one want to do password-only C3 (e.g., because storing username, password pairs is considered too high a liability given their utility for credential tweaking attacks [104]), FSB represents the best known approach.

Other C3 services include, for example, Vericlouds [17] and GhostProject [15]. They allow users to register with an email address, and regularly keep the user aware of any leaked (sensitive) information associated with that email. Such services send information to the email address, and the user implicitly authenticates (proves ownership of the email) by having access to the email address. These services are not anonymous and must be used by the primary user. Moreover, these services cannot be used for password-only C3.

Distribution-sensitive cryptography. Our FSB protocol uses an estimate of the distribution of human chosen passwords, making it an example of distribution-sensitive cryptography, in which constructions use contextual information about distributions in order to improve security. Previous distribution-sensitive approaches include Woodage et al. [134], who introduced a new type of secure sketch [56] for password typos, and Lacharite et al.'s [84] frequency-smoothing encryption. While similar in that they use distributional knowledge, their constructions do not apply in our setting.

3.10 Conclusion

We explore different settings and threat models associated with checking compromised credentials (C3). The main concern is the secrecy of the user passwords that are being checked. We show, via simulations, that the existing industry deployed C3 services (such as HIBP and GPC) do not provide a satisfying level of security. An attacker who obtains the query to such a C3 service and the username of the querying user can more easily guess the user's password. We give more secure C3 protocols for checking leaked passwords and username-password pairs. We implemented and deployed different C3 protocols on AWS Lambda and evaluated their computational and bandwidth overhead. We finish with several nuanced threat models and deployment discussions that should be considered when deploying C3 services.

CHAPTER 4

MIGHT I GET PWNED: A SECOND GENERATION PASSWORD BREACH ALERTING SERVICE

4.1 Introduction

Users often pick the same or similar passwords across multiple web services [50, 106, 136]. Attackers therefore compromise user accounts using passwords leaked from other websites. This is known as a credential stuffing attack [61]. In response, practitioners have set up third-party services such as Have I Been Pwned (HIBP) [88, 122], Google Password Checkup (GPC) [111, 121], and Microsoft Password Monitor [77] that provide APIs to check if a user's password has been exposed in known breaches. Such breach-alerting services, also called compromised credential checking (C3) services [88], help prevent credential stuffing attacks by alerting users to change their passwords.

Existing C3 services, however, can leave users vulnerable to credential tweaking attacks [50, 104, 127] in which attackers guess variants (tweaks) of a user's leaked password(s). Pal et al. [104] estimate that such a credential tweaking attacker can compromise 16% of user accounts that appear in a breach in less than a thousand guesses, despite the use of a C3 service.

We therefore initiate exploration of C3 services that help warn users about passwords similar to the ones that have appeared in a breach. We design "Might I Get Pwned" (MIGP, the name is a tribute to the first-ever C3 service, HIBP). In MIGP, a server holds a breach dataset $\tilde{S}$ containing a set of username, password pairs $(u_i, \tilde{w}_i)$. A client can query MIGP with a username, password pair (u, w), and learns if there exists $(u, \tilde{w}) \in \tilde{\mathcal{S}}$ such that $w = \tilde{w}$ or w is similar to $\tilde{w}$. To realize such a service, we must (1) determine an effective way of measuring password similarity, that (2) works well with a privacy-preserving cryptographic protocol, and that (3) resists malicious clients that try to extract entries from $\tilde{\mathcal{S}}$.

Ideally, we want our similarity measure to help warn users if their password w is vulnerable to online credential tweaking attacks. These attacks [50,104,127] take as input a breached password $\tilde{w}$ and generate an ordered list of guesses. Therefore, a good starting point for defining similarity is to call w similar to $\tilde{w}$ should w appear early in the guess list generated by a state-of-the-art credential tweaking attack. Such a generative approach also works well with simple extensions to existing cryptographic private membership test (PMT)-based protocol [88,121]. A PMT allows a client to learn if $(u, \tilde{w}) \in \tilde{\mathcal{S}}$ without revealing it to the server. To extend, we can have the server insert n variants of each breached password into $\tilde{\mathcal{S}}$ and we can allow clients to generate m variants and repeat the PMT for each of them. The PMT can be designed to reveal, upon a match, whether a password matches the original password or a variant.

To concretize this approach requires understanding how to efficiently generate effective variants. Existing credential tweaking attack algorithms are computationally expensive to run [104,127], and it is unclear, apriori, what are good values for m and n. We use empiricism to explore different techniques for enumerating variants and show via simulations how these techniques help protect against credential tweaking attacks. We start with the deep learning [104] and mangling rules techniques [50] pioneered in prior works on credential tweaking. We also suggest a new, simple-to-implement generative approach that uses an empirically-derived weighted edit distance to rank mangling rules. We show via simulation that our new approach with $m = 10$ and $n = 10$ reduces credential tweaking attack success rate by 94% compared to using only exact-checking, where the attacker uses a thousand guesses and adapts to the breach alerting service being used.

Another challenge for MIGP services is *breach extraction attacks*. C3 services could contain breach data that is not publicly available. Most C3 services provide public APIs, which malicious clients can abuse to learn a user's breached passwords by querying the service with a sequence of likely passwords. MIGP services may make such extraction attacks faster, because, intuitively, finding one of many variants of the target password would also reduce the search space.

We formalize this new breach extraction attack setting and show that optimal strategies for an attacker are NP-hard to compute. Nevertheless, attackers can use heuristic approximations. We evaluate such heuristics empirically for various values of n and m. Our simulation shows that an attacker can compromise $2.8\times$ more user accounts in 1,000 guesses for server-only variant generation ($n = 100$) than the best attack against a traditional exact-checking service. Allowing a hybrid of client-side ($m > 0$) and server-side variant generation leads to even more effective attacks.

We therefore propose a blocklisting strategy to reduce breach extraction success rates: remove (blocklist) most popular passwords and their variants. Users should be warned to avoid such easy-to-guess passwords whether or not they appear in a breach. Blocklisting the most common 10^4 passwords can reduce the success rate of the best-known breach extraction attack against a MIGP service to below the success rate possible against currently deployed C3 services.

We implement a prototype of MIGP with 1.14 billion breached username, password pairs, and show that online computation work for the server is small, client-side latency is comparable to existing C3 services (500 ms), and certain parameter regimes allow bandwidth required to be less than 1.43 MB. We further empirically explore different trade-offs in performance and security for client-side, server-side variant and hybrid generations for MIGP to help practitioners decide which approach to use. All this helped educate our deployment of MIGP in collaboration with Cloudflare, a major CDN and security service provider [24]. It is now in production use in their web application firewall product to notify login servers about potential attacks.

Contributions. The main contributions of this paper are:

- We initiate exploration of similarity-aware C3 services and present the design of MIGP, which allows checking if a password is vulnerable to credential tweaking attack without revealing it to the MIGP server.
- We empirically evaluate the effectiveness of different similarity measures to mitigate credential tweaking attacks.
- We analytically and empirically analyze the threat of breach extraction attacks, in which malicious clients attempt to extract credentials from a C3 service. We discuss multiple approaches to mitigate this threat, including a new popular-password blocklisting mechanism.
- We report on an initial prototype of MIGP and show its practicality by deploying at Cloudflare.

4.2 Background and Prior Work

Credential stuffing attacks and defenses. Billions of passwords are available online as a result of compromises [61, 122]. As users often choose the same or similar password for different web services [50, 106, 127], attackers use these leaked data for credential stuffing attacks. As a prevention measure, C3 services have been adopted in client browsers [111], in password managers [7], and by login server backends to proactively check user credentials. Existing C3 services include Have I Been Pwned (HIBP) [122], Google Password Checkup [111], Enzoic [19], and the recently introduced Microsoft Password Monitor [77]. HIBP [122] has publicly documented APIs to check if a username or password is in a breach. Several password managers such as 1Password and LastPass and browsers such as Firefox are using HIBP to warn users about their leaked passwords. This may result in false positives since common passwords will always be flagged.[1]

Google Password Checkup (GPC) [121], released as a Chrome-extension in 2019 [111] and later integrated into Chrome, checks if a username, password pair is present in the leak, leading to fewer false positives compared to HIBP. The Chrome password manager uses GPC to check all of a user's website credentials to determine if they are in a known breach, but does not flag passwords that are similar to ones in breaches. Li et al. [88] formalized the security requirements of C3 systems in an honest-but-curious server setting and proposed a protocol that we use to build MIGP in this paper.

[1]We found flagging based on only passwords will raise 29% false alarms, and based on only usernames will raise 36% false alarms to users whose passwords might not be vulnerable to a credential tweaking attack.

The state-of-the-art C3 protocol proposed in [88,121] now deployed by GPC handles a large scale of breach datasets using *bucketization*. To check a username, password pair (u, w), the client sends a bucket identifier j which is the first 16 bits of the cryptographic hash of u (smaller hash prefix helps preserve the privacy of the username). In parallel, the client and server perform a private membership test (PMT) protocol to securely determine if (u, w) is in the bucket containing the set of all $(u_i, \tilde{w}_i)$ with the same username hash prefix. The PMT protocol is built using the efficient oblivious PRF (OPRF) protocol, 2HashDH [74], though a recently proposed partially oblivious PRF 3HashSDHI may be used to slightly improve security [123]. A more recent service, Microsoft Password Monitor [77], uses homomorphic encryption (HE) to compute the PMT, but reveals the username completely to the server.

To prevent users from reusing their password across web services, Wang and Reiter [128,129] proposed protocols to check if a user is using the same password in multiple participating web services. The efficacy of this protocol relies on the coordination of the web services, making it harder to deploy. Moreover, as we show in Section 4.6, the PMT protocols used in their work would not scale to billions of username, password pairs without sacrificing the privacy of the username. Wang and Reiter also mention that their protocol can be extended to check for similar passwords across multiple web services [129], but did not provide details on how to do so.

Credential tweaking attacks and defenses. Currently deployed C3 services cannot warn users about a password unless the exact password is present in the breach. For example, a minor variation, such as adding "7" to the end of the compromised password "yhTgi456", won't be detected by the C3 service.

Users often pick similar passwords while resetting their passwords on a web service [136] or when picking passwords for different web services [50]. These passwords are vulnerable to credential tweaking attacks [50,104,127], where the attacker tries different variations of the leaked password.

Wang et al. [127] and Das et al. [50] used human-curated rules to generate guesses for a credential tweaking attack. Subsequently, Pal et al. [104] took a data-driven, machine-learning approach to build similarity models for passwords from the same user. They trained a sequence-to-sequence [119] style neural network model (pass2path) that outputs similar passwords given an input password. This is now the best-known attack, with simulation showing that a pass2path-based attack can compromise 16% of accounts of users that appeared in a breach using at most 1,000 guesses, despite the use of a C3 service as a credential stuffing countermeasure. Pal et al. also showed in a case study that over a thousand accounts at Cornell University were at the time vulnerable to credential tweaking attacks, showcasing their practical risk.

Pal et al. proposed a potential defense: a personalized password strength meter (PPSM) which considers the strength of a selected password based on its similarity to the user's other passwords. But they do not offer a way to utilize PPSMs in the context of a privacy-preserving C3 service, and left building similarity-checking C3 services as an open question.

4.3 Overview of MIGP

In this paper, we build a similarity-aware C3 service, called Might I Get Pwned (MIGP). MIGP generalizes existing C3 services to add new features that can warn users about passwords that may be vulnerable to credential tweaking attacks.

Service architecture and functionality. The MIGP server will have a breach dataset $\tilde{\mathcal{S}}$, containing a set of N username, password pairs $\{(u_1, w_1), \ldots, (u_N, w_N)\}$ where each $u_i \in \mathcal{U}$ is a username and each $w_i \in \mathcal{W}$ is a password. The sets $\mathcal{U}$ and $\mathcal{W}$ consist of all possible user-chosen usernames and passwords. A MIGP client can query the MIGP server with a username, password pair (u, w) to learn if there exists a $(u, \tilde{w}) \in \tilde{\mathcal{S}}$ such that $w = \tilde{w}$ or w is *similar* to $\tilde{w}$. The MIGP server, therefore, returns "match" if $w = \tilde{w}$, returns "similar" if w is similar to $\tilde{w}$, and returns "none" otherwise.

A MIGP client can be, for example, a user's browser, their password manager, an authentication service, or, as in our Cloudflare deployment, a web application firewall that wants to use breach alerting to help secure user accounts. We will use as a running example the user's browser as client, and discuss other deployment settings in Section 4.7.

Like existing C3 services, MIGP should scale to millions of requests a day with billions of username, password pairs in its database. We propose various techniques to make MIGP fast and practical, like offline processing the breach data to speed up online queries and rate-limiting clients using verifiable delay functions rather than slow hashing (Appendix C.6).

Threat model. In our threat model, we consider two distinct threats: (1) an honest-but-curious server trying to learn about a user's queried password, and (2) a malicious client querying the MIGP server to retrieve other users' breached passwords.

We assume the MIGP server is honest-but-curious: it doesn't deviate from the protocol but observes the protocol in an attempt to glean information from the user queries. Technically, we note that our MIGP protocol is in fact one-sided simulatable [68], a model which allows the server to behave maliciously. But for practical purposes, a malicious server can misguide a user by returning a wrong bucket of passwords and falsely reporting the user's vulnerable password as safe (i.e., an input-switching attack). Regular audits and other monitoring techniques may be useful mitigations. We are not aware of any other active attacks and will focus on the honest-but-curious server setting hereafter.

Ideally, we would like the MIGP server (or any C3 server) to learn nothing about the queried usernames or passwords. However, building a practical solution that achieves this requirement is hard given the huge scale of $\tilde{\mathcal{S}}$ with billions of credentials. The state-of-the-art protocols in existing C3 services reveal some bits of information about the username to allow partitioning $\tilde{\mathcal{S}}$ into smaller buckets on which a private membership test (PMT) protocol can be efficiently executed. Looking ahead, MIGP will extend this approach to perform a private similarity test (PST) over the bucket.

Clients of MIGP can be malicious. In particular, they might mount a guessing attack in an attempt to extract username, password pairs from $\tilde{\mathcal{S}}$. We call this a *breach extraction attack*. These are a concern when the breach database $\tilde{\mathcal{S}}$ contains data from leaks that are not yet publicly available. In turn, learning a user's (leaked) password can help the attacker compromise that user's accounts on other web services through credential stuffing and tweaking attacks. Prior work did not empirically analyze this threat for exact-check C3 services, but they did include anti-abuse countermeasures such as requiring computationally intensive slow hashing to complete a query [121]. This threat is particularly concerning for MIGP as clever attacks may exploit similarity.

Unsatisfactory approaches. The core of MIGP is a password similarity metric. While there are a number of ways to compute password similarity, few can preserve the privacy of the queried password. For example, Pal et al. [104] design password embedding models that map passwords to a vector space; distance in the space captures similarity. Using password embeddings directly (e.g., the client sending a password embedding to the server) is unsafe as it might reveal the underlying passwords.

One can instead build a MIGP service by combining a password embedding with a secure two-party computation (2PC) protocol that privately computes the dot product and threshold comparison. However, even state-of-the-art 2PC protocols for computing dot products [80] are not yet efficient enough to be used in our setting (which will require computing thousands of such dot products per query). We estimate, based on a prototype implementation using a 2PC library named Crypten [80], that a single client query would take 16 seconds (without network latency) to complete private dot product and comparison (Appendix C.1). Other approaches that rely on existing secure two-party computation protocols, such as computing a weighted edit distance between passwords, will similarly fall short of our performance requirements.

Generative models for password similarity. We instead use a generative approach to measure similarity, which will enable more efficient privacy-preserving protocols. We consider generative approaches that either start with a breached password or with a client's password. For the former, let $\tau_n : \mathcal{W} \mapsto \mathcal{W}^n$

be a function that generates n passwords that are likely to be chosen by a user, given one of their other breached passwords. Thus, a client password w and breached password $\tilde{w}$ are similar if $w \in \tau_n(\tilde{w})$. Here, we assumed $w \notin \tau_n(w)$ for all $w \in \mathcal{W}$. For the second approach, an inverse generative model, say $\tilde{\tau}_m$, generates m variants given a client's password; we declare a password similar to a variant if $\tilde{w} \in \tilde{\tau}_m(w)$. Because similarity is not necessarily symmetric, it can be that $\tau_n \neq \tilde{\tau}_m$. Looking ahead, we can use the model τ_n to generate likely variants at the server given a breach dataset, while we can use $\tilde{\tau}_m$ to generate variants at the client. We will also explore a hybrid approach that combines the two, in which case we consider a client's password w similar to a variant $\tilde{w}$, if $\big(\{w\} \cup \tilde{\tau}_m(w)\big) \cap \big(\{\tilde{w}\} \cup \tau_n(\tilde{w})\big) \neq \emptyset$ and $\tilde{w} \neq w$. A big question we will tackle is how to best instantiate τ_n and $\tilde{\tau}_m$.

4.3.1 MIGP protocol

MIGP builds off first-generation C3 designs, specifically, the identity-based bucketization (IDB) protocols due to Li et al. [88] and Thomas et al. [121]. At a high level, the IDB protocol splits the leaked credential database into several buckets based on truncated hashes of usernames. The client reveals the bucket identifier to the service, and then performs an OPRF-based private membership test (PMT) protocol over that bucket to check for equality.

In Figure 4.1, we provide an overview on how to extend IDB to allow the client to check for similar passwords. We augment the server's breach data with variants of each breached password using τ_n. The client queries the server using the IDB protocol with the user password and checks if it succeeds. The client

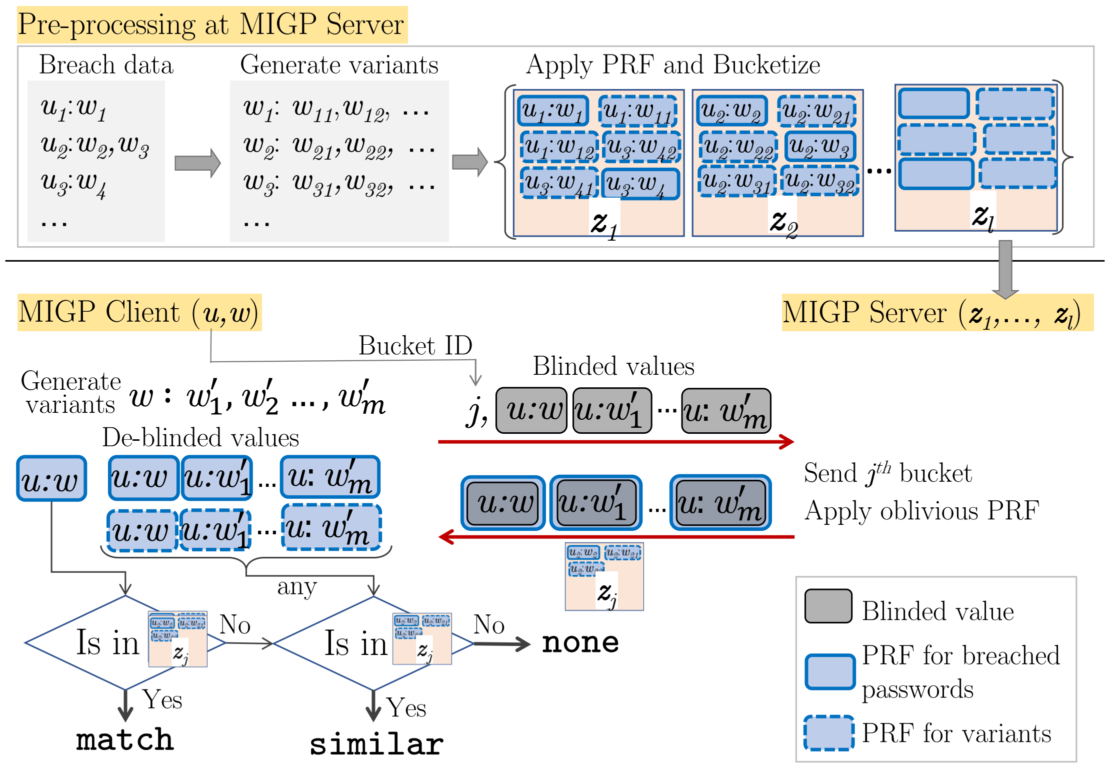

Figure 4.1: MIGP Protocol for checking if a queried password w is similar to a password present in breach data. Cryptographic details of the protocol are given in Figure 4.2.

can also generate variants, via $\tilde{\tau}_m$. There are nuanced security and computation trade-offs for this approach, which we will discuss at the end of this section. For now, we assume the client and the server both generate m and n variants using $\tilde{\tau}_m$ and τ_n functions. Setting $m = 0$ and $n = 0$, reduces MIGP functionality to existing exact-checking C3 services, such as IDB. The cryptographic details of MIGP protocol, which fits our security requirements, is given in Figure 4.2.

Pre-processing. The underlying IDB protocol uses a specialized oblivious PRF construction. Briefly, the PRF takes as input a username u, a password w, and a secret key κ and is defined as $F_\kappa(u\|w) = \mathsf{H}_2(u\|w, \mathsf{H}_1(u\|w)^\kappa)$. This is the same as the 2HashDH construction due to Jarecki et al. [74]. Here H_1 maps onto an elliptic curve group $\mathbb{G}$ (with group operation written multiplicatively) where the decisional Diffie-Hellman (DDH) problem is hard; and $\mathsf{H}_2 : \{0,1\}^* \times \mathbb{G} \mapsto \{0,1\}^\ell$

maps a binary string and a group element to an ℓ-bit string. At least one of the two hash functions used should be computationally expensive (for the client) to ensure rate limiting and abuse prevention on the client side. We explore trade-offs on how to choose the hash functions in Section 4.7 and Appendix C.6.

The server chooses κ and applies F_κ to all the username, password pairs in the breach. These are stored separate "buckets", identified by the l-bit prefix of a cryptographic hash of the username, denoted $\mathsf{H}^{(l)}(u)$. As we want the client to find out if the queried password is similar to one stored by the server, we use two PRF functions: The server stores $F_\kappa(u\|w)$ (shown in thick blue border boxes in Figure 4.1) corresponding to the leaked credential (u, w), and $F'_\kappa(u\|w') = F_\kappa(u\|w') \oplus 1$ for $w' \in \tau_n(w)$ corresponding to the password variants, which is represented by the dashed blue boxes in Figure 4.1. The last bit of the PRF of similar passwords is flipped to differentiate it from the original leaked password.

Online computation. MIGP client, on input a user id u and password w, calculates the ID of the bucket to query based on the username, $j = \mathsf{H}^{(l)}(u)$. Then the user generates m variants of their password w based on $\tilde{\tau}_m$. The client "blinds" the passwords and their variants, sending to the server $\mathsf{H}_1(u\|w)^{r_0}, \mathsf{H}_1(u\|w'_1)^{r_1}, \ldots, \mathsf{H}_1(u\|w'_m)^{r_m}$ for random values $r_0, \ldots, r_m \in \mathbb{Z}$. Blinding ensures that the MIGP server does not learn anything about the query (beyond j).

The server raises each of the blinded values to the secret key κ, and sends these back to the client, along with the bucket $\mathbf{z}_j$. The client can deblind the values to finish computing the PRF on all $m+1$ values. Then it checks if $F_\kappa(u\|w)$ is present in the bucket, and if so, it learns that (u, w) is in the leaked data,

```
Pre-processing at server:
Server's secret key: $\kappa$;   $\tilde{\mathcal{S}} = \{(u_1, \tilde{w}_1), \ldots, (u_N, \tilde{w}_N)\}$
for $(u, \tilde{w}) \in \tilde{\mathcal{S}}$ do:
  $j \leftarrow \mathsf{H}^{(l)}(u)$;   $\mathbf{z}_j \leftarrow \mathbf{z}_j \cup \{F_\kappa(u \| \tilde{w})\}$
  $\mathbf{z}_j \leftarrow \mathbf{z}_j \cup \{F_\kappa(u \| w') \oplus 1 \mid w' \in \tau_n(\tilde{w})\}$
Online phase:
    Client                                                   MIGP server
    Input: $(u, w)$                                          Input: $\kappa, \mathbf{z}$
    $j \leftarrow \mathsf{H}^{(l)}(u)$
    $r_0 \leftarrow_\$ \mathbb{Z}$;  $\mathbf{x}_0 \leftarrow \mathsf{H}_1(u \| w)^{r_0}$
    $(w'_1, \ldots, w'_m) \leftarrow \tilde{\tau}_m(w)$
    for $i \in [1, m]$ do :
      $r_i \leftarrow_\$ \mathbb{Z}$;  $\mathbf{x}_i \leftarrow \mathsf{H}_1(u \| w'_i)^{r_i}$   $\xrightarrow{j, \mathbf{x}}$   for $i \in [1, m]$ do :
                                                               $\mathbf{y}_i \leftarrow \mathbf{x}_i^\kappa$
                                             $\xleftarrow{\mathbf{y}, \mathbf{z}_j}$
    $\tilde{z}_0 \leftarrow \mathsf{H}_2(u \| w, \mathbf{y}_0^{\frac{1}{r_0}})$
    $\tilde{z}_1 \leftarrow \{\tilde{z}_0 \oplus 1\}$
    for $i \in [1, m]$  do :
      $h \leftarrow \mathsf{H}_2(u \| w'_i, \mathbf{y}_i^{\frac{1}{r_i}})$
      $\tilde{z}_1 \leftarrow \tilde{z}_1 \cup \{h, h \oplus 1\}$
    if $\tilde{z}_0 \in \mathbf{z}_j$   return match
    else if $\tilde{z}_1 \cap \mathbf{z}_j \neq \emptyset$   return similar
    else return none
```

Figure 4.2: Protocol for checking if a password similar to the user's password (w) is present in the leaked data ($\tilde{\mathcal{S}}$).

outputting match. If not, the client checks if any other computed PRF values $F_\kappa(u \| w'_i)$, or those values with last bit flipped $F_\kappa(u \| w'_i) \oplus 1$, or $F_\kappa(u \| w) \oplus 1$ is in the bucket. If any are found, then the client learns that (u, w) is similar to a (u, w') found in the password breach, outputting similar. Otherwise, it outputs none.

4.3.2 Server- vs. client-side variant generation

Based on the values of the parameters n and m, MIGP protocol can allow generating variants only on the client-side ($n = 0$), only on the server-side ($m = 0$), or a mix of both. By allowing variants only on the server side, the existing IDB protocol can be easily adopted, making it simpler to implement. However, the server database expands by n times, requiring more disk space and more bandwidth due to larger buckets.

In the case of client-side generation of variants, no change on the server is required. The variants can be batched together in a single API query to the server, saving network round trips and bandwidth. (Note, the client only needs to download the matching bucket from the server once per username.) Moreover, in this approach, the client will have more control over the variations. It can use inputs from the user, such as their other passwords, to generate personalized variants that are likely to be used as passwords by that particular user. Such personalization was shown to be useful for correcting password typos [46] and could be also useful for MIGP.

Although the client-side generation of password variants has some benefits, it also suffers from some key limitations. First, existing C3 services have rate-limiting measures, like slow hashing, to prevent malicious clients from extracting the breach data by repeatedly calling the APIs with different password guesses [111]. This would make checking multiple variations of a password too expensive to be practical. To make things faster, the server could allow batching all queries into one request and reduce the client-side computation. But there is a key security issue with this approach: as the OPRF protocol blinds queried values, the server cannot differentiate if a query contains a set of variants of a password or completely different passwords. This can be exploited by a malicious client to obtain a factor of n improvement in breach extraction attack efficacy (Section 4.5.1). Zero-knowledge proofs [65] could be used, in theory, to prevent a malicious client from checking arbitrary passwords, but it remains an open question whether they can be made practical in this setting. We leave finding an efficient solution to this problem for future work.

In the hybrid approach, the client generates m personalized variations, possibly based on their other passwords or personal information, and the server also stores n variations of each breached password. Such a protocol with appropriate client and server-side generation functions can increase the protection against credential tweaking attacks to the equivalent of generating $n \times m$ variants on the server or the client side (as we show in Section 4.4). The hybrid approach can also reduce the storage cost on the MIGP server, reduce bandwidth cost due to smaller buckets, and lower the advantage gained in breach extraction attack by allowing a smaller number of guesses per malicious query.

In subsequent sections, we explore the performance, security, and efficacy implications of different choices of m, n, along with how to build practical generative models $\tilde{\tau}_m$, τ_n.

4.4 Efficacy of Different Similarity Measures

We explore different measures of password similarity using generative models that enumerate the most likely variants of a given password. Though the client-side and server-side models can be different, we cannot learn two different models due to the limitation of our dataset (as we explain below). Thus we will focus on building a single generative model τ that will be used both on the client and server side; we will show even this simple approach already performs well.

In particular, we compare different similarity measures τ based on efficacy at protecting from credential tweaking attacks, computational performance, and security against breach extraction attacks. We focus on the first two in this section, and discuss the third in Section 4.5.

4.4.1 Similarity measures

As we focus on generative similarity metrics, any credential tweaking attack can be repurposed to be a similarity metric. We, therefore, start with the attack algorithms proposed in Das et al. [50] and Pal et al. [104]. We denote these by Das and P2P, respectively. We also created more efficient and effective variants of these methods, named Das-R and wEdit, as we discussed below. Each method takes as input a password w and outputs an ordered list of similar passwords.

We also compare the generative methods to the embedding-based similarity measure, PPSM, proposed in [104]. Although existing PST protocols suitable for use with PPSM are not fast enough for use in practice (as discussed in Section 4.2), we still discuss them here should PST protocols become more suitable for deployment in the future.

Das. Das et al. [50] were the first to show that users select similar passwords across multiple websites, and that it is easy to guess a user's password given one of their other passwords. They, given a password w, use a set of hand-crafted tweaks to generate similar passwords. We refer to this approach of generating similar passwords as Das.

Das-R. We observed that ordering of the tweaks used in Das is not effective for smaller n. So we reorder the set of tweaks based on the frequency with which these tweaks are used by users in our dataset (Section 4.4.2). We show the reordering significantly improves the efficacy of the rules when considering smaller numbers of variants (≤ 10). We call this similarity measure Das-R. The reordered rules are given in Appendix C.2. Not all tweaks apply to all passwords, in which case we continue applying further tweaks until we obtain n variants.

P2P. While Das et al. used hand-crafted tweaks for generating variations, Pal et al. [104] used a neural network model, called pass2path (P2P), to learn the tweaks a user is likely to make to their passwords. This resulted in the most damaging credential tweaking attack to date, outperforming prior works, such as [50] and [127]. We refer to this approach as P2P. While P2P is quite effective at capturing password similarity, it is slow and expensive (even with GPUs) to compute.

wEdit. Finally, we explore automatically deriving a ranked list of tweaks that can be applied to a password to obtain variants. Although tweaks have long been used in password cracking systems (e.g., [28]), here the goal is different — finding variants likely to be chosen by a user and that are vulnerable to credential tweaking attacks.

Following the definitions in [104], we define a *unit transformation* as a specific edit to be applied to the input password w. A unit transformation is defined by a tuple (e, c, l) where e specifies the edit type as one of insert, delete, or substitute; c denotes the character to be inserted or substituted ($c = \perp$ for deletion); and l is the location for the edit. The location is length-invariant, representing the distance from the first character by positive numbers and from the last character by negative numbers; we use the smaller of the two distances and break ties using the distance from the start of a password. For example, $(\mathsf{insert}, \text{‘0'}, -1)$ specifies adding the character ‘0' to the end of a password, and $(\mathsf{substitute}, \text{‘a'}, 2)$ specifies replacing the second character with a lowercase letter ‘a'.

Given a pair of passwords (w, w'), we can calculate the shortest sequence of unit transformations to generate w' from w. We refer to this as the *transformation path*. The computation can be done using standard edit distance algorithms. We use the keypress representation of the passwords w, w' as defined in [45], which includes special characters such as shift and caps lock.

Given a breach dataset containing multiple passwords associated with the same user accounts, we compute transformation paths for every pair of passwords belonging to the same user. Then we create a ranked list of transformation paths based on how many pairs of passwords it explains. To generate variants of a password w, apply the transformation paths one at a time, in decreasing frequency order, skipping if it is not applicable. We stop if we have generated n variants. Note that wEdit contains a much more exhaustive list of tweaks (transformation paths) compared to Das-R. However, wEdit is not sensitive to the input unlike the handcrafted rules in Das-R, which include rules like insert ‘3' if the last character of the word is ‘2'. (The rules for wEdit and Das-R are given in Appendix C.2.) Nevertheless, we will see below that they have similar efficacy in our context.

#	$\tilde{S}$	S	T	S_1	S_2
Users	908	760	230	380	380
Passwords	438	373	119	210	210
Unique user-pw pairs	1,147	918	229	459	459
Total user-pw pairs	1,317	1,069	248	535	535

Figure 4.3: Number of unique users, passwords, and username, password pairs (in millions) in the entire dataset $\tilde{S}$, breach dataset S, and test dataset T. S_1 and S_2 are two equal partition of S. Total number of username, password pairs with duplicates shown in the last row.

4.4.2 Breach dataset

To drive empirical evaluation of the five similarity approaches, we use a dataset containing a compilation of publicly available breaches on the Internet [42]. This dataset was also used in prior academic research work and industry reports, e.g., [61, 88, 104], and has been confirmed to contain real user accounts. The breach compilation dataset contains nearly 1.4 billion unique email, password pairs. We clean the dataset based on the procedure described in [104], such as removing passwords containing non-ASCII characters or longer than 30 characters (which affects only 0.3% of users). We merged usernames based on the mixed-method from [104] and removed users having more than 1,000 passwords. The resulting dataset $\tilde{S}$ consists of 1.3 billion unique username-password pairs from 908 million unique users (Figure 4.3). More details about the dataset can be found in [104].

For our simulations, we partitioned $\tilde{S}$ into two: a larger split (80%) simulates the leaked dataset S, which we further divide into two equal sets S_1 and S_2 with no common users between them; and the remaining dataset (20%) is used as the testing dataset T. In Figure 4.3, we report some statistics on the dataset splits. S and T consist of 760 and 230 million unique usernames, respectively. About 82 million usernames are present in the intersection of T and S, implying that these users in the test dataset have at least one password in the simulated breach dataset. The number of users, passwords, and user-password pairs are similar for S_1 and S_2, as expected.

For the attack simulations in Section 4.4.3, we conservatively assume the attacker has access to more data than what is known to the MIGP service. That is, we provide the attacker with the entire leaked dataset S but train the similarity mechanism for MIGP only on S_1 (training is needed for Das-R, P2P, and wEdit). The test dataset can, therefore, be considered a list of users' current passwords on some target websites for which the attacker wants to gain illicit account access. The test dataset is neither accessible to the attacker nor to the similarity mechanisms that we train.

4.4.3 Empirical efficacy evaluation

We examine the effectiveness of a similarity measure based on protection from credential tweaking attacks and impact on usability due to false warnings, which can cause user fatigue. To quantify this, we classify each pair of passwords belonging to the same user as *vulnerable* or *safe* based on whether or not they are vulnerable to credential tweaking attacks.

We pick password pairs (w_1, w_2) belonging to the same user, such that w_1 is selected from S_1 and w_2 from T. Hence, both the attacker and service know the breached password w_1 corresponding to the target user and want to attack/protect the user's unknown (test) password w_2. We flag a pair vulnerable if w_2 can be guessed by pass2path [104] given w_1 in a thousand guesses. Otherwise, we flag the pair as safe. From all vulnerable pairs, we randomly sampled

$10,000$ pairs to measure the true positive rate (TPR) of a similarity measure τ as the fraction of vulnerable pairs that are flagged by it. Similarly, we randomly sampled $10,000$ safe pairs and measured the false positive rate (FPR) of τ as the fraction of these pairs that are flagged by τ, which burdens users with spurious warnings.

The efficacy of a generative similarity measure can be different based on whether it is applied to the breached password (on the MIGP server, τ_n) or to the queried password (on the client, $\tilde{\tau}_m$). For a pair of passwords w, w' in the breach data, if we knew w was used before w', then we could train τ_n to generate edits that modify w to w' while $\tilde{\tau}_m$ consider variants of w' leading to w. However, our training data does not contain such temporal ordering information. Therefore, as mentioned above, we use $\tilde{\tau}_m = \tau_n$, i.e., the variants are generated in the same way on the client and the server.

An orthogonal point is that for the hybrid case, in which both $m > 0$ and $n > 0$, better utility may come from considering simultaneously which rules should be used on the client and which ones should be used on the server. But the space of all possible $m \times n$ combined client-server rule sets is large, and we do not know how to search for optimal solutions efficiently. We used a greedy approach to understand the efficacy, but leave to future work developing better search techniques, and evaluating their potential for improving efficacy.

Result. Figure 4.4 shows the performance of the similarity measures. As expected, increasing the number n or m of similar passwords improves the coverage against vulnerable pairs across all methods. However, that also increases the false positive rate, flagging safe passwords.

Parameters	Similarity measures	% True positive	% False positive
$n = 10$ or $m = 10$	Das	33.2	0.6
	Das-R	**52.6**	0.0
	P2P	46.4	0.0
	wEdit	49.6	0.0
$n = 100$ or $m = 100$	Das	46.9	2.2
	Das-R	63.5	0.2
	P2P	**69.0**	0.1
	wEdit	**69.3**	0.1
$n = m = 10$	Das-R	**89.9**	2.9
	wEdit	75.2	2.2
$n = m = 10$ (Greedy)	Das-R	**93.5**	4.5
	wEdit	84.4	3.0
$\theta = 0.83$	PPSM	**67.9**	2.0
$\theta = 0.75$		87.6	4.7
$\theta = 0.5$		99.1	14.0

Figure 4.4: True positive (ones vulnerable to 1,000-guess pass2path attack) and false positive (others) rates for different similarity measures, computed over 10,000 randomly sampled password pairs. The best performing measures are boldfaced.

For MIGP where variants are only generated on the server (or on the client side), Das-R gives the maximum 52.6% coverage for n or $m = 10$ tweaks among all the generative approaches. P2P and wEdit perform the best with n or $m = 100$ with 69% coverage. PPSM gives high coverage against the attacks but also has a higher false-positive rate compared to the generative approaches. As the TPR of PPSM with reasonable FPR $\approx 2\%$ is lower than that of generative approaches (such as wEdit, n or $m = 100$) and anyway does not lead to efficient protocol, we do not consider it further. Between wEdit and P2P, wEdit is drastically simpler to deploy and faster to run, requiring $4.5\times$ less pre-computation time (see Appendix C.1).

The hybrid approach, where $n = 10$ variants are generated by the server and $m = 10$ are generated by the client both using Das-R rules, gives the best coverage to credential tweaking attacks, flagging 90% of vulnerable passwords at considerably low false flagging of safe passwords (2.9%). We also tried a greedy

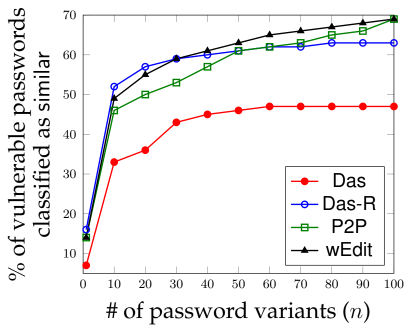

Figure 4.5: Percentage of vulnerable password pairs flagged by different similarity measures for varying n. The slopes of the graphs for all similarity measures decrease rapidly for $n > 10$.

approach where we iteratively pick the tweaks on the client and the server that maximizes coverage of the tweaks until each side has m and n tweaks. This approach performs better at identifying vulnerable passwords, flagging 94% of them, but also has a high false positive rate (4.5%).

Efficacy with increasing variants. Figure 4.5 examines how the efficacy of the four generative models varies by the increasing number of tweaks n in server side variant checking. The results are the same for m in the client-side MIGP. Das-R outperforms other techniques for $n \leq 30$. wEdit outperforms the other measures after that for $30 \leq n \leq 100$. It was surprising to us that the rule-based approaches (Das-R, wEdit) end up matching or exceeding the performance of the much more complex deep learning approach underlying P2P. This is because rule-based approaches can easily capture frequently seen variants, for low values of n. The deep learning approach works better for large n by finding and ordering less frequently seen similarity relationships. For example, for $n = 10^3$, P2P outperforms wEdit by 4%.

Although increasing n increases attack coverage, the slope of the curves decrease rapidly (Figure 4.5). Therefore, the benefit of considering a higher n

value diminishes while increasing storage (only for server-side variant checking), computation, and bandwidth cost, as we see in Section 4.6.

Therefore in the rest of the paper, we use Das-R for n or $m = 10$ or hybrid $n = m = 10$ and wEdit for n or $m = 100$.

4.4.4 Adaptive credential tweaking attackers

We now measure the reduction in a credential tweaking attacker's success in breaking into a user account, should a MIGP service be deployed with one of the similarity measures discussed in Section 4.4.3. We compare against the baseline where an exact-checking C3 service such as [88,121] is used. For the simulation, we adapt the best-known credential tweaking attack — pass2path [104] — to be aware of the MIGP service.

We conservatively assume that the attacker has access to the entire breach dataset S, while the MIGP service has access to the subset S_1. We sample 10,000 users from the test dataset T, who are also present in S_1 and have a password marked safe (not flagged as match or similar) by the service under consideration; this constitutes the target users for the attacker. With this user list, we can simulate the scenario where the service (the exact checking C3 service or the MIGP service) warned the user about their unsafe passwords on a target website and the user subsequently changed their password. Though not all users will abide by warnings, this setup allows us to compare the maximum security benefits of a service using similarity measures.

Breach alerting method	$q = 10$	$q = 100$	$q = 1000$
Exact checking [88,121]	10.1	13.4	16.3
MIGP [Das-R, $n = 10$ or $m = 10$]	2.8	5.0	7.9
MIGP [wEdit, $n = 100$ or $m = 100$]	1.9	3.0	5.2
MIGP [Das-R, $n = 10$ and $m = 10$]	0.6	1.0	1.4

Figure 4.6: Success rate of credential tweaking attacker in $q \in \{10, 100, 1000\}$ guesses, assuming that the attacker is aware of the breach alerting mechanism.

We consider an online attack setting, where too many incorrect password submissions should trigger an account lockout, resulting in attack failure. Thus the attacker has a query budget $q \leq 10^3$. We measure the fraction of user passwords the attacker can guess in q attempts, assuming one of their other passwords is present in S. The attacker enumerates guesses by first generating candidates using pass2path, and skipping any that would be flagged by the service. The attacker can infer this themselves because we assume that the service's breach data and the similarity measure are known to the attacker.

As shown in Figure 4.6, when only credential stuffing countermeasures are in place, such as using [88] or [121], the credential tweaking attacker can guess passwords of 10.1% of accounts using 10 guesses, which matches the performance reported in [104]. The hybrid MIGP reports the highest reduction in attack efficacy, 94% for $q = 10$. For the server or client-side MIGP service, the efficiency decreases to 1.9% when n or $m = 100$ variations based on wEdit are used; a reduction of 81%. The attack accuracy decreases by nearly 78% and 68% for $q = 100$ and $q = 1,000$, respectively. Across the board, larger n or m gives better protection against the credential tweaking attacker, reducing the attack's efficacy.

4.5 Security Evaluation

A MIGP service allows clients to check whether a password similar to the queried one is present in the breach. Current C3 services, such as GPC [121], do not reveal any information about breach data unless a client queries the exact username, password pair. A natural question is: Will moving towards similar-aware C3 services degrade the confidentiality of the username, password pairs in the leaked database?

We formalize the abstract setting of a malicious client that has access to a similarity oracle, assuming it is cryptographically secure (we refer to this as the ideal functionality following parlance from the 2PC literature).

4.5.1 Breach extraction attacks

A MIGP service could be abused by malicious clients that seek to learn about user credentials. This is particularly concerning should a MIGP service have access to relatively new breaches that are not widely available to attackers, making the service a potential target for what we call a *breach extraction attack*. We model such attacks via the security game given in Figure 4.7. In it, the adversary is given access to an oracle that implements the ideal functionality of a MIGP service. Note that the oracle is parameterized by a target password w^* chosen by the game, the query budget q, and a similarity measure τ. In each query, the adversary can send up to m passwords, and each is checked against the target w^* and its variants $\tau(w^*)$. Here we use τ for the server-side variants, but allow

```
MIGP($w'_1, \ldots, w'_m$)                    MIGPGuess($\mathcal{A}', q$):
$q \leftarrow q - 1$                          $w^* \leftarrow_p \mathcal{W}$
if $q \leq 0$ then return none                $\tilde{w} \leftarrow \mathcal{A}'^{\mathsf{MIGP}}$
for $i = 1$ to $m$ do                         if $\tilde{w} = w^*$ return true
  if $w'_i = w^*$ then return ($i$, match)    else return false
  if $w'_i \in \tau(w^*)$ then return ($i$, similar)
return none
```

Figure 4.7: An abstract breach extraction attack security game for a MIGP service parameterized by a number of MIGP protocol invocations q, a distribution of passwords p, a similarity model τ, and a number of client-side variants allowed m.

a malicious client to choose any m passwords for the client-side variants. The goal of the attacker is to guess w^* within the given query budget q.

Finding an optimal guessing strategy for breach extraction is NP-hard. (See Appendix C.3 for details.) However, it is possible to create efficient greedy approximate algorithms (Appendix C.4). We note that Chatterjee et al. [44] explored NP-hardness results and greedy heuristics for typo-tolerant password authentication, where the server returns true or false should the submitted password be a typo of the registered password. But, in our setting, MIGP oracle returns one of three possible answers. Therefore, their setting and results don't directly carry over to our setting.

In Appendix C.4.1, we present an efficient greedy algorithm for the $m = 0$ case, called GreedyMIGP. We now turn to measuring the efficacy of the greedy algorithm to understand the real-world threat of a malicious client attempting to extract data from the MIGP service. We assume the attacker has a guessing budget of $q \leq 10^3$. This setup assumes that the MIGP server will deploy some form of rate-limiting on queries from a client (as discussed in Section 4.7).

Experiment setup. For simulation, we assume the MIGP oracle is instantiated with S_1 data (see Section 4.4.2), and the attacker has access to only S_2. This simulates the situation where the attacker does not know the leaked data present in MIGP, and is trying to learn those breached passwords for a user. We sample 25,000 username, password pairs from S_1. For each pair, the attacker is given the username and required to find the target password. We compute the efficacy of an attack as the fraction of username, password pairs that the attacker can successfully guess. (As per our data division, none of the target usernames are present in S_2, and therefore the attacker cannot attempt a targeted credential tweaking-type attack.) We evaluate the security loss for 10 variants based on Das-R rules, and 100 variants based on wEdit rules. We first experiment with only server-side variant generation ($m = 0$); later in the section, we report the efficacy of breach extraction attacks when allowing client-side variant generation.

The S_2 dataset has 210 million passwords. If the attacker sets $\mathcal{W}$ to all the passwords in S_2, it will make GreedyMIGP very slow to run (Figure C.4). We instead heuristically picked the top one million passwords as $\mathcal{W}$ for the attack. These passwords are used by 24% of users in S_2.

For comparison, we also simulate a C3 service that does not provide checking for similar passwords, which we refer to as MIGP service with $n = 0$. For this case, the attack is simpler: query the MIGP service with the top q passwords, and if any query returns match, output the queried password.

Results. The success probability of the attacker in guessing the target password using $q \in \{10, 100, 1000\}$ queries is shown in Figure 4.8. We explain the β values below; here we focus on the rows with $\beta = 0$. An attacker can learn 6.57% of passwords in less than a thousand guesses against an exact-checking C3 service ($n = 0, \beta = 0$). The attacker's success probability increases to 13.58% for $n = 10$

β	n	$q = 10$	$q = 100$	$q = 1000$
0	0	1.64 ($\pm$ 0.22)	3.36 ($\pm$ 0.35)	6.57 ($\pm$ 0.61)
	10	2.14 ($\pm$ 0.26)	5.22 ($\pm$ 0.56)	13.58 ($\pm$ 0.61)
	10^2	1.69 ($\pm$ 0.17)	3.54 ($\pm$ 0.30)	17.18 ($\pm$ 0.73)
10	0	0.03 ($\pm$ 0.00)	0.36 ($\pm$ 0.08)	2.80 ($\pm$ 0.27)
	10	**1.19** ($\pm$ 0.15)	3.90 ($\pm$ 0.40)	12.12 ($\pm$ 0.45)
	10^2	**0.93** ($\pm$ 0.11)	2.43 ($\pm$ 0.25)	15.67 ($\pm$ 0.61)
10^2	0	0.03 ($\pm$ 0.03)	0.37 ($\pm$ 0.08)	2.50 ($\pm$ 0.30)
	10	0.93 ($\pm$ 0.13)	2.71 ($\pm$ 0.27)	9.91 ($\pm$ 0.42)
	10^2	0.79 ($\pm$ 0.13)	1.52 ($\pm$ 0.26)	10.91 ($\pm$ 0.41)
10^3	0	< 0.01 ($\pm$ 0.00)	0.18 ($\pm$ 0.06)	1.46 ($\pm$ 0.14)
	10	0.76 ($\pm$ 0.11)	1.42 ($\pm$ 0.10)	5.94 ($\pm$ 0.16)
	10^2	**0.72** ($\pm$ 0.09)	0.97 ($\pm$ 0.11)	9.21 ($\pm$ 0.24)
10^4	0	< 0.01 ($\pm$ 0.02)	0.03 ($\pm$ 0.02)	0.27 ($\pm$ 0.03)
	10	0.71 ($\pm$ 0.10)	1.02 ($\pm$ 0.07)	3.34 ($\pm$ 0.23)
	10^2	0.70 ($\pm$ 0.10)	0.92 ($\pm$ 0.11)	**4.87** ($\pm$ 0.12)

Figure 4.8: Attack success rate given different query budgets (q) for different attack scenarios. Here $n = 0$ (first row in each block) emulates existing exact checking C3 services. MIGP oracle uses Das-R and wEdit similarity rules for $n = 10$ and $n = 100$, respectively. The service blocks most frequent β passwords. All success rates are in percent (%) of 25,000 target users sampled from S_1. Standard deviations (shown in parenthesis) are measured across the 5 random folds of these pairs. Lower values imply better security.

and 17.18% for $n = 100$. In the latter case, moving to a MIGP service may lead to a $2.8\times$ increase in an attacker's ability to perform breach extraction attacks.

We observed a counter-intuitive pattern for $q = 10$ and 100: the attack success rate decreases with the increase of n from 10 to 100. This is because large n produces larger balls, making it easier to get in the ball, but harder to identify the correct password given a small query budget $q = 10$. Therefore, for small q, guessing the most weighted password ball may not be the optimal strategy. We plot the attack success for different values q for $n = 10$ and $n = 100$ in Figure 4.9. For query budget $q < 230$, increasing the number of password variants n from 10 to 100 actually decreases this attack's success rate.

Blocklisting. The abuse prevention mechanisms (e.g., slow hashing and API rate limits) used in current C3 systems can only slow down breach extraction attacks, but do not prevent them. We, therefore, propose a simple yet effective mechanism to reduce the attack success: blocklist the top β passwords so that an attacker learns nothing from the MIGP service should a user's password be one of them. The MIGP service can do so by removing all the blocklisted passwords and their variants from its breach database. (This will also reduce storage and bandwidth overhead as we show in Section 4.6.) These popular passwords are anyway unsafe to be used by any user irrespective of whether they are leaked or not. Therefore, a client application can warn the user who is using a password equal to or similar to one of the blocklisted passwords.

For our simulations, we assume the MIGP service blocks the β most frequent passwords according to S_1 and their variants (according to the setup). If an attacker queries the MIGP service with any of the blocklisted passwords the service always responds as none. Of course, the attacker is aware of the set of blocklisted passwords and their variants.

We experiment with different values of β as shown in Figure 4.8; $\beta = 0$ denotes no blocklisting. Blocklisting reduces the attacker's success across all values of n. For $q \leq \beta$, the success probability of an attacker for $n = 10$ and $n = 100$ remains below that of existing C3 services (with $n = 0$ and $\beta = 0$), except for $\beta = q = 10^3$ in $n = 100$. We believe this is due to the higher ball size in case of $n = 100$. In this case, we need to blocklist $\beta = 10^4$ passwords to reduce the attack success rate below existing C3 services. We highlight those numbers in the figure. As the top β passwords are blocklisted, the attacker can't learn if the user has a breached password that is one of the top β passwords. The attacker's best bet is to guess passwords that are outside the top β passwords. The password

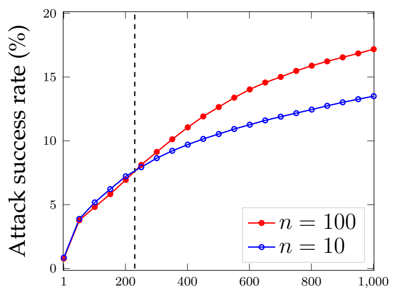

Figure 4.9: Comparison of attack success of breach extraction attack for Das-R ($n = 10$) and wEdit ($n = 100$). for different values of q. For query budgets $q \leq 230$ (black dashed line), success rates are slightly lower for the higher value of n.

distribution follows Zipf's law [126], therefore leading to a significant decrease in breach extraction accuracy.

We also compute the breach extraction success rate against users who do not use weak passwords. The results are shown in Figure C.5 (Appendix C.5). For these users, we found that the relative increase in attack success due to MIGP service is higher, but the absolute success rates are small. For example, for $\beta = 10^4$ and $q = 10^3$, the attack success is 0.27% when $n = 0$, 2.98% when $n = 10$, and 2.56% when $n = 100$. Similar to Figure 4.8, the attack efficacy for $n = 100$ is worse than that of $n = 10$ in most cases. We suspect that this is because our attack is not optimal, especially for a smaller number of guesses ($q \leq 10^3$).

Security of client-side variant generation. A client can generate m variants of a password w and check them all in parallel with the server. Due to the limitation of our MIGP protocol there is no way for the server to verify if the client has generated variants truthfully. Thus a malicious client can use this to expand its query budget by a factor of m: The client simply submits the next m passwords computed using GreedyMIGP, to obtain in total $m \cdot q$ queries.

n	m	$q = 10$	$q = 100$	$q = 1000$
10^2	0	0.70 (± 0.10)	1.10 (± 0.05)	4.87 (± 0.12)
10	10	1.02 (± 0.07)	3.34 (± 0.23)	8.49 (± 0.09)
0	10^2	1.45 (± 0.08)	3.06 (± 0.22)	10.60 (± 0.68)

Figure 4.10: Comparing breach extraction attack success rate between generating m variants on the client-side and n variants on the server-side. Here we assume $\beta = 10^4$. The first row is the same as the last row in Figure 4.8.

We show the success rate of this breach extraction attack for different m and n values with $\beta = 10^4$ in Figure 4.10. Allowing $m = 10^2$ variants on the client side increases the attack success by more than twice for any $q \leq 10^3$ compared to allowing only the server side $n = 10^2$ variants. Using the hybrid approach to variant generation with $m = 10$ and $n = 10$ reduces the attack success rate, but still remains significantly higher than $m = 0, n = 100$ setting. Therefore, we suggest that if the breach data is sensitive it is safer to disallow client-side variant generation and apply strict rate limiting.

4.5.2 Security of the MIGP Protocol

Our MIGP protocol (given in Figure 4.2) requires minimal changes to previously proposed [88] and currently deployed [121] protocols. This made deployment simpler, and also the cryptographic security is derived directly from the underlying protocol. Here we briefly summarize the security achieved by MIGP, considering in turn curious servers and malicious client threat models. MIGP communications must be protected with TLS, preventing any manipulations of the buckets or client queries by a network adversary.

As in the prior protocols, the MIGP server learns only the client's queried bucket ID. This reveals some bits of information about the username, but nothing about the queried password, assuming the password and username are independent. (Some users may choose passwords similar to their username, but it's unclear how a malicious server can usefully exploit this practice.) An actively malicious server can modify the result obtained by a client, e.g., by erroneously claiming passwords are not in the breach when, in fact, they are (or vice versa). This attack is possible also for deployed exact equality checking protocols [88, 121]. In theory, one could try to use techniques to prevent this, e.g., by having the server publish a commitment to the dataset and then performing zero-knowledge proofs of (non-)membership [98]. We do not believe this is necessary for breach alerting as such attacks would seem to have low value to attackers.

An encrypted bucket reveals to a client the number of entries in the bucket, and the updated entries and the time of updates to buckets will be revealed over time. This could conceivably have security implications in some contexts. As shown in Section 4.5.1, MIGP services are susceptible to breach extraction attacks, and therefore must employ different forms of rate limiting. Note that a malicious client can submit a query for a bucket for username u but submit an OPRF request for username $u' \neq u$. This is true as well for existing C3 services. Thus rate-limiting should not be based (only) on bucket identifier, and instead on a client identifier (cookie or IP address) or a token mechanism such as PrivacyPass [51].

4.6 Performance Analysis

We implement a prototype of MIGP and conduct experiments to estimate its performance. We also compare it with existing C3 services, such as GPC [121] or IDB [88] (equivalent to MIGP with $n = 0$ and $m = 0$). We experiment with no blocklisting ($\beta = 0$) and blocklisting the $\beta = 10^4$ most frequent passwords (and their variants). We want to measure and compare: (1) storage overhead on the server side, (2) latency of running the protocol, and (3) total bandwidth usage. Here we use as breach dataset $\tilde{\mathcal{S}}$ the entire 1.14 billion username, password pairs from the dataset described in Section 4.4.2.

Prototype implementation details. We implement the MIGP client and server in Python 3.8, with petlib library for elliptic curve operations. We chose the elliptic curve group secp256k1 for $\mathbb{G}$ and set $\ell = 128$. For H_1, we use petlib's hash-to-point function to map the username, password pair to $\mathbb{G}$; internally it uses rejection sampling [72] with $\mathrm{SHA256}$. We also use $\mathrm{SHA256}$ for H_2. Should either of H_1 or H_2 be a slow hash, there will be additional overhead in precomputation. We select the most frequent β passwords for blocklisting based on the dataset $\tilde{\mathcal{S}}$. The server is built using the Flask [22] library and the client uses the requests [21] library to make queries. This prototype implementation is publicly available.[2]

For all the pre-processing of the data, we used a desktop with an Intel Core i9 processor and 128 GB RAM. We did not use any GPU to optimize hash computation in our experiments. For latency and bandwidth comparisons, we run the server (t2.medium) and client (t2.micro) on two different AWS EC2 instances running the Ubuntu 20.04 LTS image and located in two different regions — US-East and US-West.

Precomputation overhead. We precompute the buckets of PRF values for the entire breach dataset. MIGP, in comparison to exact-check C3 protocols, requires

[2] `https://github.com/islamazhar/migp_python`

processing an additional n variants for each leaked password and storing the resulting PRF values. The precomputation on the server involves computing $F_\kappa(x) = \mathsf{H}_2(x, \mathsf{H}_1(x)^\kappa)$, where $x = (u\|w)$ for the breached password and $F_\kappa(x) \oplus 1$, where $x = (u\|\tilde{w})$ for $\tilde{w} \in \tau_n(w)$. We use H_2 to reduce the representation size of the hash to 16 bytes, which saves disk space and bandwidth.

Generating $n = 10$ variants for a password using Das-R similarity rules takes less than 0.02 ms, whereas generating $n = 100$ variants using wEdit similarity rules takes 0.8 ms, on average. If we had used Argon2 as H_2, it would take 95 ms on average for computing the hash of one username, password pair, an estimated 361.5 CPU-years for pre-processing all username, password pairs and their variants on our reference implementation. Should breach data not be particularly sensitive, a deployment can skip slow hashing or use time-lock puzzles instead (see Appendix C.6).

The PRF values are then separated into buckets based on $\mathsf{H}^{(l)}(u)$. Duplicate values could arise when pairs (u, w_1) and (u, w_2) satisfy the condition $\tau_n(w_1) \cap \tau_n(w_2) \neq \emptyset$, which can be common for users with credentials from multiple sites in the known breach. Duplicates should be either omitted (as we do in our prototype) or replaced with other variants. The former is better for performance, but note that the length of buckets now depends on relationships between different passwords — we are unsure whether this can be exploited by an attacker given the large number of users in each bucket.

The total storage cost for 1.14 billion unique username, password pairs and their variants would be 1.67 TB (considering every entry has $n = 100$ unique variants). Blocklisting reduces storage (and bandwidth) requirements, and we

l	w/o blocklisting avg.	std.	w/ blocklisting avg.	std.
16	1,751,666	36,832	1,431,876	30,107
20	109,479	9,192	89,492	7,513
24	6,842	2,297	5,592	1,877

Figure 4.11: Average bucket size of MIGP with $n = 100$ variants for each password on the leak dataset, which contains 1.14 billion unique username, password pairs.

			Client side latency (ms)					
C3 service	Server storage	B/w (MB)	Query Prep$^{\otimes}$	Query Prep$^{\oplus}$	API call	Finalize	Total$^{\otimes}$	Total$^{\oplus}$
IDB-16†	15 GB	0.23	< 1	96	321	< 1	322	417
IDB-20‡	15 GB	0.01	< 1	96	125	< 1	126	221
WR19-Bloom‡	1.0 TB	177.20	6,990	7,065	39,045	< 1	46,035	46,110
WR20-Cuckoo‡	0.8 TB	0.81	59	155	38,560	< 1	38,619	38,715
MIGP-Server‡	1.5 TB	1.43	< 1	95	498	3	501	596
MIGP-Client†	15 GB	0.23	46	10,713	450	38	534	11,202
MIGP-Client‡	15 GB	0.02	48	10,007	390	38	476	10,435
MIGP-Hybrid‡	0.2 TB	1.43	7	953	421	12	440	1,386

† $l = 16$ ‡ $l = 20$ $^{\otimes}$ without rate-limiting $^{\oplus}$ with rate-limiting.

Figure 4.12: Average latency (in milliseconds) for checking one password via different private membership or similarity test protocols used in different C3 services. with different parameters. IDB-16 and IDB-20 do not use any variants. MIGP-Server and MIGP-Client generate 10^2 variants on the server and the client side, respectively. WR19-Bloom uses Bloom filter to reduce the b/w requirement. For rate limiting we use Argon2 as H_2, which takes around 95ms to compute. All latency measurements are averaged over 25 complete API calls with standard deviations $< 10\%$.

found that blocklisting the 10^4 most popular passwords and their variants reduces the database size by 18% to 1.36 TB.

Bucket size selection. To make the private membership test protocols practical, C3 services use bucketing to partition the leaked dataset based on the prefix of the hash of the username. MIGP follows the same approach. However, as MIGP contains n variants of each password, the buckets would be quite large for MIGP for the same number of buckets. Large bucket size will increase communication costs in terms of bandwidth and latency as the client has to download a larger amount of data. We can reduce the bucket size by increasing the number of buckets — by increasing the length of the hash prefix l. The average bucket sizes for different hash prefix lengths for MIGP are shown in Figure 4.11. The bucket size, as expected, decreases exponentially with the prefix length (l). The average bucket size with blocking the most frequent 10^4 passwords for $l = 16$ (which is used by GPC [121]) is 1.43 million, or 22.85 MB. Increasing the length of the bucket identifier l to 20 reduces the bucket size to 1.37 MB.

Latency & bandwidth comparison. We measure and compare the latency and bandwidth requirements for running different compromised credential checking services: IDB-16 (also called GPC) [121], IDB-20 [88], WR19-Bloom [128], WR20-Cuckoo [129], and our protocols MIGP-Server, MIGP-Client, and MIGP-Hybrid. Although WR19-Bloom and WR20-Cuckoo were designed to check user's passwords in multiple web services, these protocols can be used for checking a user's leaked passwords. MIGP-Server, MIGP-Client, and MIGP-Hybrid are the different versions of our protocol with variants generated on the server side ($n = 10^2$), client side ($m = 10^2$), and both ($n = 10, m = 10$). IDB-16 and IDB-20 are implemented following the same construction as MIGP, but with different lengths of prefixes for bucketing and setting $m = n = 0$. For WR19-Bloom and WR20-Cuckoo, we use the corresponding authors' implementation[3] in Go but customize it for the client-server setting.

We pick 25 random passwords from the test data set T and run each C3 service protocol separately for different n, m and prefix length l. We simulate the server data with dummy buckets containing b entries, where value b is ran-

[3]https://github.com/k3coby/pmt-go

domly sampled from the normal distribution with the mean and standard deviation set to the values we computed in Figure 4.11, with $\beta = 10^4$. The server and the client are executed in two different EC2 virtual machines located in two different availability zones in two coasts of the United States. They are connected via a 252 Mbits/sec network link.

We report the average latency with the breakdown for preparing the query, calling the API and waiting for the response, and finalizing the result for each protocol evaluation in Figure 4.12. The overall time to prepare for a query takes less than 7 ms, for GPC, IDB, MIGP-Server and MIGP-Hybrid. The total computational cost for the server is very small compared to the client, however, the client spends time downloading the data from the server (leading to higher latency in MIGP compared to GPC and IDB). After the query, the client finalizes the result by computing H_2 of the username, password pair to compare with the bucket entries. Using slow hash function for rate limiting would add about 95 ms to the query preparation to all protocols. MIGP-Client takes 100x more time in query preparation due to generating the variants and checking them. MIGP-Client can be particularly expensive with rate limiting using slow hashes, such as Argon2. It can take more than 10 seconds for a complete run of the protocol run. MIGP-Hybrid strikes a balance between server storage and query preparation time. It reduces the storage cost and query time by a factor of 10 compared to MIGP-Server and MIGP-Client, respectively.

The slowest among all, is WR19-Bloom and WR20-Cuckoo protocols, taking more than 38 sec for one complete protocol execution. The primary contributors to the latency are: (a) before a query, the client has to encrypt each entry of the Bloom filter homomorphically (using Paillier encryption [103]), (b) the client has to send all the encrypted Bloom filter entries to the server, which is quite large (216.8 MB), and (c) the server has to compute large group multiplications over *all* entries in the Bloom filter. WR20-Cuckoo protocol uses Cuckoo hashing [63] which improves overall latency and bandwidth, however, still falls short of being practical due to high computational overhead on the server.

MIGP-Server (with $n = 10^2$ and $l = 20$), MIGP-Client (with $m = 10^2$ and $l = 16$) and MIGP-Hybrid (with $n = 10$, $m = 10$ and $l = 20$) takes less than 534 ms to compute a query if we don't use rate limiting, which is comparable to currently-in-use IDB-16 (with $l = 16$) protocol. The overhead for MIGP-Server stems from the high bandwidth usage due to large bucket sizes. The buckets can be cached on the client-side or served directly from CDNs (such as Cloudflare) as practiced by HIBP [33] to improve performance. Client-side caching of buckets saves fetching the same bucket again for checking different passwords for the same username. As most users have only a few email addresses, this can save significant network bandwidth and time over multiple queries.

4.7 Deployment Discussion

We worked with Cloudflare, a major CDN and security company [27], to deploy the MIGP protocol (1) as a public-facing API similar to HIBP, and (2) as a new breach alerting feature within Cloudflare's web application firewall (WAF) product [24]. MIGP is deployed as an opt-in feature in WAF, which detects login requests to Cloudflare customer websites, extracts username and password fields, and queries a MIGP service deployed on Cloudflare Workers [25]. The result of the MIGP query is added to an HTTP header that is forwarded to the customer login service, informing them should the login request be utilizing a breached credential or ones similar to them. The libraries underlying the MIGP implementation have been open-sourced and are publicly available [23]. In this section, we present some deployment considerations and the lessons we learned.

Deployment details. During pre-processing, the breach database is transformed into MIGP buckets. We post-process the OPRF outputs using HKDF [83] to generate a 21-byte hash value; the last byte is XORed with a one-byte flag denoting whether a bucket entry is an exact match or a variant. Slow hashing is supported by the implementation, but applying slow hashing at the scale is expensive and our initial deployment omits it. We discuss this more below.

The buckets of credentials produced in the pre-processing step are stored in Workers KV [26], a high-performance, distributed key-value store. Our deployment uses 20-bit bucket prefixes with $n = 8$ variants per entry generated using the Das-R rules and without blocklisting popular passwords ($\beta = 0$). The deployment caps each individual bucket size to 25 MB, which under this configuration should support breach data up to 64 billion entries. The MIGP service is able to serve over 50% of client requests in under 135ms, and 95% of requests in under 573ms. Most performance overhead is due to the cost of fetching buckets form Workers KV store, as only frequently accessed buckets are cached at 250 datacenters that are running Workers. Other buckets must be fetched from a centralized data store which adds latency.

Our current implementation does not support client-side variants. As shown earlier (Figure 4.12), enabling client-side variant generation in the future may provide attractive performance benefits. This must be balanced against the risk of breach extraction attacks (see Figure 4.10).

Breach extraction attacks. A key concern as we designed and discussed MIGP deployment at scale was gauging the risk of breach extraction attacks. Any client can attempt to mount such an attack against the public API. The WAF deployment does not necessarily provide malicious web clients with a MIGP oracle: the result of MIGP queries are only shared with the login service and not the client. Login services should not reveal MIGP outputs to unauthenticated clients.

For both deployments we have thus far only utilized datasets that are widely available on underground forums, obviating the concern about breach extraction attacks in the short term. To use more sensitive breaches in the future, further mitigations will need to be enabled, including popular password blocklisting and rate limiting. Our deployment already benefits from rate-limiting of individual IP addresses and other anti-automation techniques [36]. We note that a common rate-limiting approach is to require clients to obtain an API key through some slow (or paid) registration process, but this approach won't work for WAF deployment scenario.

Another rate-limiting approach would be to use slow hashing. Recall that the MIGP protocol uses two hash functions within the OPRF, computing outputs as $\mathsf{H}_2(u\|w, \mathsf{H}_1(u\|w)^\kappa)$. Either of the hashes can be made computationally expensive to both slow down online breach extraction attacks and to make offline hash cracking attacks harder should the MIGP server be compromised. There are nuanced security-computation trade-offs between the choice of which to make slow. If H_2 is expensive the client cannot do offline processing of the slow hash without communicating with the server, which is not true if only H_1 is slow. However, one benefit of having just H_1 slow is that the server can store the intermediate $\mathsf{H}_1(u\|w)^\kappa$ values for faster key updates (see below). Google Password Checkup (GPC) [121] uses a slow hash for H_1 and a fast hash for H_2.

An alternative approach to slow hashing is to use asymmetric hashing, also called proof-of-work [58,73] or time-locked puzzles [114]. We discuss these approaches in more detail in Appendix C.6.

Bucket updates. MIGP services (like other C3 services) may periodically update their leaked data, such as when new breaches are exposed online. This will require adding new credentials to the buckets. Updating the buckets with OPRF outputs under the same key could, in principle, allow an attacker to identify the newly added username, password pairs. Although it is unclear how this leakage can be exploited, it would be better to avoid the leakage entirely. One way is to rotate the OPRF key κ every time there is a new leak. However, recomputing the OPRF output from the stored breach data will be computationally very expensive given the slow hash function. Assuming H_1 is slow, an optimization would be to have the server record the output of the group multiplication $\mathsf{H}_1(u\|w)^\kappa$ in some offline, safe storage. Then the new OPRF outputs can be computed for the new key κ' by raising $\mathsf{H}_1(u\|w)^\kappa$ values to κ'/κ, and applying the fast hash $\mathsf{H}_2(\cdot)$ to them. This approach is similar to the key rotation mechanism used by Pythia [62].

MIGP warnings: effectiveness and usability. To estimate the effectiveness of the MIGP service, we instrumented the WAF deployment to measure the ratio of the number of login attempts that MIGP flagged as similar to the number that

MIGP flagged as an exact match. The average ratio over the period of a week is 0.2 (with 0.01 standard error of the mean), implying that MIGP flags 20% more login attempts compared to an exact-checking C3 system. This represents a significant improvement by MIGP over exact-checking in terms of alerting on credentials that are vulnerable to attacks such as those based on pass2path [104]. Our instrumentation does not record how many WAF-monitored attempts correspond to vulnerable accounts (e.g., attempts will include some number of incorrect submissions and attacks), but customer services can distinguish between these cases and act appropriately.

Prior work has shown that users may not be responsive to breach alerts [111]. We expect that MIGP deployments will face a similar challenge. Server-side breach alerting, like our WAF deployment, allow high-security services to force users to change MIGP-flagged passwords. One open question prompted by our work is how best to communicate to users that their password is similar to a breached password and how to guide them towards safer choices.

4.8 Conclusion

In this work, we tackled the problem of building MIGP, an updated version of C3 systems that can securely warn users from selecting passwords similar to (and same as) a breached password which can be vulnerable to credential tweaking attacks. Via comparing different similarity metrics we show that computing variants of the password using weighted edit distance rules provide the best combination of performance and efficacy. Underlying MIGP is a secure private similarity test (PST) protocol. Despite secure PST, MIGP protocols can still be vulnerable to breach extraction attacks, where an attacker can extract leaked (but not yet public) credentials from a MIGP service. We show that the attacker's success probability can be reduced significantly using blocklisting popular passwords. We implement and show that MIGP achieves computational overhead comparable to C3 services. Finally, we deploy MIGP with Cloudflare and provide nuanced discussions about deploying MIGP in practice.

CHAPTER 5
CONCLUSION

We design a privacy-preserving compromised credential checking system, MIGP, that allows users to query whether their login credentials are at risk due to exposure in a breach. Our main conclusions are as follows.

Choosing similar passwords across websites can be very damaging - We train models, using state-of-the-art deep learning models, that learn how users choose similar password variants. Using these trained models, we build the most effective credential tweaking attack that can effectively compromise accounts with similar passwords in case of a breach in any one of them. In my ongoing work, we investigate how to effectively communicate the threat to the users and guide them towards safer choices. Also, password managers should actively check for similar passwords using PPSM and report them.

More work is needed to put proper after-breach remediation steps in place - We discussed various deployment scenarios for MIGP and collaborated with Cloudflare to integrate MIGP with their web application firewall application. It is deployed in practice and warns the login servers about at-risk credentials. However, more actions are needed to be taken in this direction to diminish the after-effects of a breach, such as — 1) Implementation of a fast, safe, and effective breach reporting system, 2) Building a prompt alerting mechanism to notify users about vulnerable accounts on the compromised and other websites, and 3) Installing a proper messaging system to educate users about selecting a new, random and uncorrelated password at those websites. We need more collaboration between the academic community and industry to address these issues at the core and build a safer online experience after a breach.

APPENDIX A

APPENDIX - CREDENTIAL TWEAKING ATTACK AND DEFENSE

A.0.1 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 $c \in \Sigma \cup \{\text{'st'}, \text{'}\rangle\text{'}\}$ and outputs a probability distribution over the characters in $\Sigma \cup \{\text{'st'}, \text{'}\rangle\text{'}\}$ 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 $c_0 = \text{'st'}$ and an end-of-sequence symbol '$\rangle$'. Therefore, in this model, we can rewrite Equation (2.1) as follows, where v_i is the output of the decoder on input v_{i-1} and c_{i-1}.

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

We used the default neural network architecture and the hyperparameters used in seq2seq [120] 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 2.5 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,

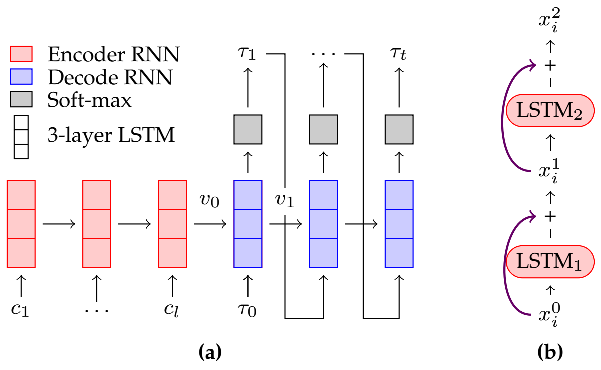

Figure A.1: **(a)** Diagram of encoder-decoder architecture for pass2path learning. **(b)** A 2-layer LSTM cells with residual connection. Here c_i's are characters of input passwords, τ_i's are transitions, and x_i^j's are internal states in neural networks.

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 [127] can guess 13.1%. (See Figure 2.3)

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.

A.0.2 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 [120]. 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) [71] was shown to be effective in avoiding vanishing and exploding gradient problems [118]. We used LSTM blocks wrapped in residual cells. Residual cells, first used for image recognition using deep neural networks [69], "short-circuit" the input of a layer to the output, bypassing internal calculations. (See Figure A.1(b).) We found pass2path achieves slightly better accuracy at noticeably lower training time with residual cells than without it. We implemented pass2path in TensorFlow [30] using the building blocks provided by the library. Each LSTM cell in the model has three hidden layers, each layer with 128 hidden units.

A diagram of the neural network architecture of pass2path is given in Figure A.1(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.

A.0.3 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 [120]: 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 [78] to minimize the cross-entropy loss [112] 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 [125] (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 [120]. 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 [116]. 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 $n \in \{3, 4\}$ [118]. 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.

A.0.4 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

```
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  $e_2 \leftarrow D_{i-1,j-1}$
      $k \leftarrow \arg\min e$
      $D_{i,j} \leftarrow e_k$
      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 $P$
```

Figure A.2: GenPath algorithm for generating sequence of transformations from a pair of passwords.

the distance matrix according to the formula.

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

The pseudocode for generating a path is given in Figure A.2.

A.0.5 Re-training Wang et al. for our dataset

The Wang et al. algorithm specified in [127] 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 $\mathsf{D}^{\mathsf{E}}_{\mathsf{ts}}$.

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, $P\left(\text{'1234'}\right) = C(\text{'1234'})/C(D_4)$, 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.

A.0.6 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 2.3. 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 2.5. 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 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 A.3 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 A.4, we show the relation between

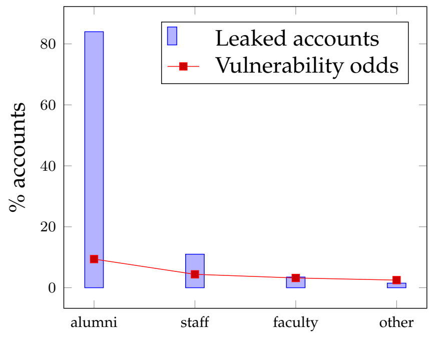

Figure A.3: Distribution of different types of accounts that are found in the breach dataset, and their odds to be vulnerable to one of the three online guessing attacks. The “other” category includes current students, contract workers, and affiliates.

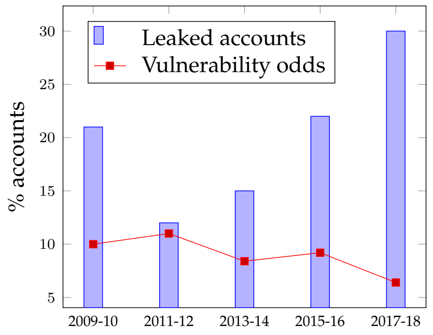

Figure A.4: Distribution of active accounts in the leaked dataset based on the year of their last password reset. The red line shows their odds of being vulnerable to targeted online guessing attacks.

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.

APPENDIX B

APPENDIX - COMPROMISED CREDENTIAL CHECKING SERVICES

B.1 Correlation between username and passwords

In Section 3.3, we choose to model the username and password choices of previously uncompromised users independently.

To check whether this assumption would be valid or not, we randomly sampled 10^5 username-password pairs from the dataset used in Section 3.6 and calculated the Levenshtein edit distance between each username and password in a pair. We have recorded the result of this experiment in Figure B.1.

We found that the mean edit distance between a username and password was 9.4, while the mean password length was 8.4 characters and the mean username length was 10.0 characters. This supports that while there are some pairs where the password is almost identical to the username, a large majority are not related to the username at all.

The statistics on edit distance between username and password in our dataset are similar to the statistics in the dataset used by Wang et al. [127], who

Distance	%
0	1.2
≤ 1	1.7
≤ 2	2.3
≤ 3	3.1
≤ 4	4.6

Figure B.1: Statistics on samples with low edit distance between username and password, as a percentage of a random sample of 10^5 username-password pairs.

determined that approximately 1-2% of the English-website users used their email prefix as their password.

This data does not prove that usernames and passwords are independent. However, even if an attacker gains additional advantage in the few cases where a user chooses their username as their password, the overwhelming majority of users have passwords that are not closely related to their usernames.

B.2 Bandwidth of FSB

To calculate the maximum bandwidth used by FSB, we use the balls-and-bins formula as described in Section 3.3. Each password w is stored in $|\beta(w)|$ buckets, so the total number of balls, or passwords being stored, can be calculated as

$$\begin{aligned} m &= \sum_{w \in \tilde{\mathcal{S}}} |\beta(w)| \\ &= \sum_{w \in \mathcal{W}_{\bar{q}} \cap \tilde{\mathcal{S}}} |\mathcal{B}| + \sum_{w \in \tilde{\mathcal{S}} \setminus \mathcal{W}_{\bar{q}}} \left\lceil \frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil \\ &\le |\mathcal{W}_{\bar{q}} \cap \tilde{\mathcal{S}}| \cdot |\mathcal{B}| + \sum_{w \in \tilde{\mathcal{S}} \setminus \mathcal{W}_{\bar{q}}} \left(\frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1 \right) \\ &\le |\mathcal{B}| \cdot \bar{q} + |\mathcal{B}| \cdot \frac{1}{\hat{p}_s(w_{\bar{q}})} + N \end{aligned}$$

The first equality is obtained by replacing the definition of $\beta(w)$; the second inequality holds because $\lceil x \rceil \le x + 1$; the third inequality holds because $S \subseteq W$.

The number of bins $n = |\mathcal{B}|$, and $m > n \log n$, if $\bar{q} > \log n$. Therefore, the maximum bucket size for FSB would with high probability be no more than $2 \cdot \left(\bar{q} + \frac{1}{\hat{p}_s(w_{\bar{q}})} + \frac{N}{|\mathcal{B}|} \right)$.

B.3 Proof of Theorem 3

First we calculate the general form of the $\mathsf{BucketGuess}_{\beta_{\text{FSB}}}$ advantage. Then, we show that for $q \le \bar{q}$, $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) = \mathsf{Adv}^{\text{gs}}(q)$, and we bound the difference in the advantages for the games when $q > \bar{q}$.

$$\begin{aligned}
\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) &= \sum_{u} \sum_{b} \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} \frac{P\left(W = w_i \wedge U = u\right)}{|\beta_{\text{FSB}}(w_i)|} \\
&= \sum_{b} \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|}
\end{aligned}$$

The second step follows from the independence of usernames and passwords in the uncompromised setting.

We will use $\mathcal{W}_{\bar{q}}$ to refer to the top $\bar{q}$ passwords according to password distribution $\hat{p}_s = p_w$, and $w_{\bar{q}}$ to refer to the $\bar{q}$th most popular password according to $\hat{p}_s$.

For $w \in \mathcal{W}_{\bar{q}}$, we can calculate the fraction in the summation exactly as $\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} = \frac{\hat{p}_s(w)}{|\mathcal{B}|}$.

For any other $w \in \mathcal{W} \setminus \mathcal{W}_{\bar{q}}$, we can bound the fraction using the bound on the number of buckets a password is placed in.

$$\frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \le |\beta_{\text{FSB}}(w)| < \frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1.$$

We can use the lower bound on $|\beta_{\text{FSB}}(w)|$ to find that

$$\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} \le \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}|}.$$

Using the upper bound on $|\beta_{\text{FSB}}(w)|$,

$$\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} > \frac{\hat{p}_s(w)}{\frac{|\mathcal{B}|\cdot\hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1} = \frac{\hat{p}_s(w)\cdot\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}|\cdot\hat{p}_s(w) + \hat{p}_s(w_{\bar{q}})} = \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w)}}$$

Since the values of $\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|}$ are always larger for $w \in \mathcal{W}_{\bar{q}}$, the values of $w_1, \ldots, w_q$ chosen for each bucket will be the top $\bar{q}$ passwords overall, along with the top $q - \bar{q}$ of the remaining passwords in the bucket, ordered by $\frac{\hat{p}_s(\cdot)}{|\beta_{\text{FSB}}(\cdot)|}$.

To find an upper bound on $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q)$,

$$\begin{aligned}
&\sum_b \max_{\substack{w_1,\ldots,w_q\\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|} \\
&\quad \le \sum_b \left(\sum_{w\in\mathcal{W}_{\bar{q}}} \frac{\hat{p}_s(w)}{|\mathcal{B}|} + (q-\bar{q})\frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}|} \right) \\
&\quad = \lambda_{\bar{q}} + (q-\bar{q})\cdot p_{\bar{q}}
\end{aligned}$$

For $q \le \bar{q}$, we have $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) \le \lambda_{\bar{q}}$.

To find a lower bound on $\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta_{\mathsf{FSB}}}(q)$, let $w^*_{\bar{q}+1}, \ldots, w^*_q$ be the $q - \bar{q}$ passwords in $\alpha(b) \setminus \mathcal{W}_{\bar{q}}$ with the highest probability of occurring, according to $\hat{p}_s(\cdot)$.

$$\begin{aligned}
&\sum_b \max_{\substack{w_1,\ldots,w_q \\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|} \\
&\quad > \sum_b \left(\sum_{w\in\mathcal{W}_{\bar{q}}} \frac{\hat{p}_s(w)}{|\mathcal{B}|} + \sum_{i=\bar{q}+1}^{q} \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \right) \\
&\quad \geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \left\lceil \frac{|\mathcal{B}| \cdot \hat{p}_s(w^*_i)}{\hat{p}_s(w_{\bar{q}})} \right\rceil \cdot \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \\
&\quad \geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \frac{|\mathcal{B}| \cdot \hat{p}_s(w^*_i)}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \frac{\hat{p}_s(w^*_i)}{1 + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)\cdot|\mathcal{B}|}} \\
&\quad \geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \hat{p}_s(w^*_i)/2 \geq \lambda_{\bar{q}} + (\lambda_q - \lambda_{\bar{q}})/2 = \frac{\lambda_q + \lambda_{\bar{q}}}{2}
\end{aligned}$$

Therefore, $\Delta_q \geq \frac{\lambda_q - \lambda_{\bar{q}}}{2}$.

Note, for every password to be assigned to a bucket, $|\mathcal{B}| \geq \hat{p}_s(w_{\bar{q}})/\hat{p}_s(w)$, or for all $w \in \mathcal{W}$, $\frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w)\cdot|\mathcal{B}|} \leq 1$.

B.4 Attacks on Correlated Password Queries

An adversary might gain additional advantage in guessing passwords underlying C3 queries when queries are correlated. For example, when creating a new password, a client might have to generate multiple passwords until the chosen password is not known to be in a leak. These human-generated passwords are often related to each other. Users also pick similar passwords across different websites [50, 104, 106, 127]. If such passwords are checked with a C3 server (maybe by a password manager [7]), and the attacker could identify multiple

```
$\mathsf{Corr\text{-}Guess}_{\beta}^{\mathcal{A}}(q)$
$(u, w_1) \leftarrow_p \mathcal{U} \times \mathcal{W}$
$w_2 \leftarrow_{\tau_{(u,w_1)}} \mathcal{W} \setminus \tilde{\mathcal{S}}_w$
$b_1 \leftarrow \beta(w_1);\;\; b_2 \leftarrow \beta(w_2)$
$\{\tilde{w}_1, \ldots, \tilde{w}_q\} \leftarrow \mathcal{A}(u, b_1, b_2)$
return $w_2 \in \{\tilde{w}_1, \ldots, \tilde{w}_q\}$
```

Figure B.2: A game to describe a simple correlated password query scenario. Here, we let $\tilde{\mathcal{S}}_w$ be the set of all passwords in the breach dataset.

queries from the same user (for example, by joining based on the IP address of the client), then the attacker could mount an attack on the correlated queries. As we described, the adversary does need a lot of information to mount such an attack, but the idea is worth exploring, since attacks on correlated queries have not been analyzed before.

Let $\{\tau_{(u,w)}\}$ be a family of distributions, such that for a given $u \in \mathcal{U}$, $w \in \mathcal{W}$, $\tau_{(u,w)}$ models a probability distribution across all passwords related to w for the user u. For example, the probability of user u choosing a password w_2 given that they already have password w_1 is $\tau_{(u,w_1)}(w_2)$.

The attack game for correlated password queries is given in Figure B.2. A client first picks a password w_1 for some web service and learns that the password is present in a leaked data. The client then picks another password w_2, potentially correlated to w_1, that is not known to be in a leak and is accepted by the web service. (For simplicity, we only consider two attempts to create a password. However, our analysis can easily be extended to more than two attempts.) In the game, the password w_2 is chosen from the set of passwords not stored by the server, according to the distribution of passwords from the transformation of w_1. The adversary, given the buckets b_1 and b_2, tries to guess the final password, w_2.

To find the most likely password given the buckets accessed (the maximum a posteriori estimation), an adversary would want to calculate the following:

$$\begin{aligned}
&\arg\max_{w} \ \Pr\left[w_2 = w \ \mid \ b_1, b_2\right] \\
&\qquad = \arg\max_{w} \ \Pr\left[b_1, b_2 \ \mid \ w_2 = w\right] \cdot \frac{P\left(w_2 = w\right)}{P\left(b_1, b_2\right)} \\
&\qquad = \arg\max_{w} \ \Pr\left[b_1, b_2 \ \mid \ w_2 = w\right] \cdot P\left(w_2 = w\right) .
\end{aligned}$$

Note that we view b_1, b_2 as fixed values for the two buckets, not random variables, but we use the notation above to save space. We can separate $\Pr\left[b_1, b_2 \ \mid \ w_2 = w\right]$ into two parts.

$$\begin{aligned}
\Pr\left[b_1, b_2 \ \mid \ w_2 = w\right] &= \Pr\left[b_2 \ \mid \ w_2 = w\right] \cdot \Pr\left[b_1 \ \mid \ w_2 = w, b_2\right] \\
&= \Pr\left[b_2 \ \mid \ w_2 = w\right] \cdot \Pr\left[b_1 \ \mid \ w_2 = w\right]
\end{aligned}$$

The second step follows from the independence of b_1 and b_2 given w_2.

We know that the first term $\Pr\left[b_2 \ \mid \ w_2 = w\right]$ will be 0 if the password w does not appear in bucket b_2. For FSB, the buckets that do contain w have an equally probable chance of being the chosen bucket. For HPB, only one bucket will have a nonzero probability for each password.

$$\Pr\left[b_2 \ \mid \ w_2 = w\right] = \begin{cases} \frac{1}{|\beta(w)|} & \text{if } b_2 \in \beta(w) \\ 0 & \text{otherwise} \end{cases} .$$

Then, to find $\Pr[b_1 \mid w_2 = w]$, we need to sum over all passwords that are in b_1. We define $\mathcal{S}_w$ as the set of all possible passwords.

$$\begin{aligned}\Pr[b_1 \mid w_2 = w] &= \sum_{w_1 \in \mathcal{S}_w} \Pr[b_1 \wedge w_1 \mid w_2 = w] \\ &= \sum_{w_1 \in \alpha(b_1)} \Pr[w_1 \mid w_2 = w] \\ &= \sum_{w_1 \in \alpha(b_1)} \frac{\Pr[w_2 = w \mid w_1] \cdot P(w_1)}{P(w_2 = w)} .\end{aligned}$$

Combining the $\arg\max$ expression with the equations above, the adversary therefore needs to calculate the following to find the most likely w:

$$\begin{aligned}&\arg\max_{w \in \alpha(b_2)} \frac{1}{|\beta(w)| \cdot P(w_2 = w)} \cdot \sum_{w_1 \in \alpha(b_1)} \Pr[w_2 = w \mid w_1] \cdot P(w_1) \\ &= \arg\max_{w \in \alpha(b_2)} \frac{1}{|\beta(w)| \cdot P(w_2 = w)} \cdot \sum_{w_1 \in \alpha(b_1)} \tau_{(u,w_1)}(w) \cdot P(w_1). \qquad \text{(B.1)}\end{aligned}$$

In practice, it would be infeasible to compute the above values exactly. For one, the set of all possible passwords is very large, so it would be difficult to iterate over all of the passwords that could be in a bucket. We also don't know what the real distribution $\tau_{(u,w)}$ is for any given u and w. For our simulations, we estimate the set of all possible passwords in a bucket using the list constructed by the attack from Section 3.6. To estimate $\Pr[w_2 = w \mid w_1]$, we use the password similarity measure described in [104], transforming passwords into vectors and calculating the dot product of the vectors.

To simulate the correlated-query setting, we used the same dataset as in Section 3.6. We first trim the test dataset T down to users with passwords

Protocol	Attack	$q = 1$	10	10^2	10^3
Baseline	single-query	0.2	1.0	2.9	6.4
HPB ($l = 16$)	single-query	18.8	31.9	45.9	58.4
	correlated	8.8	10.3	13.0	26.0
FSB ($\bar{q} = 10^2$)	single-query	0.2	1.0	2.9	8.4
	correlated	2.7	3.3	4.6	11.5

Figure B.3: Comparison of attack success rate given q queries on our correlated password test set. All success rates are in percent (%) of the total number of samples (5,000) guessed correctly.

both present in the leaked dataset and absent from the leak dataset. We then sample 5,000 of these users and randomly choose the first password from those present in the leaked dataset and the second password from the ones not in the leaked dataset. This sampling most closely simulates the situation where users query a C3 server until they find a password that is not present in the leaked data. We assume, as before, the adversary knows the username of the querying user.

For the experiment, we give the attacker access to the leak dataset and the buckets associated with the passwords w_1 and w_2. Its goal is to guess the second password, w_2. The attacker first narrows down the list constructed in the attack from Section 3.6 to only passwords in bucket b_2. As a reminder, we refer to this list of passwords as $\tilde{\alpha}(b_2)$. The attacker then computes the similarity between every pair of passwords in $\tilde{\alpha}(b_2) \times \tilde{\alpha}(b_1)$, which is $\tilde{\alpha}(b_1)$ times the complexity of running a single-query attack (as described in Section 3.6). It reorders the list of passwords $\tilde{\alpha}(b_2)$ using an estimate of the value in Equation (B.1).

The results of this simulation are in Figure B.3. We also measured the success rate of the baseline and regular single-query attacks on recovering the same passwords w_2.

It turns out that this correlated attack performs significantly worse than the single-query attack when the passwords are bucketized using HPB. For FSB, the correlated attack performs better, but not by a large amount. Although there is an improvement in the correlated attack success for FSB, the overall success rate of the attack is still worse than both attacks against HPB.

The overall low success rate of the correlated attacks is likely due to the error in estimating the password similarity, $\tau_{(\cdot,w_1)}(w)$. Though the similarity metric proposed by [104] is good enough for generating ordered guesses for a targeted attack, it doesn't quite match the type of correlation among passwords used in the test set. Even though we picked two passwords from the same user for each test point, the passwords were generally not that similar to each other. About 7% of these password pairs had an edit distance of 1, and only 14% had edit distances of less than 5. The similarity metric we used to estimate $\tau_{(\cdot,w_1)}(w)$ heavily favors passwords that are very similar to each other.

The single-query attack against HPB does quite well already, so the correlated attack likely has a lower success rate because it rearranges the passwords in $\tilde{\alpha}(b_2)$ according to their similarity to the passwords in $\tilde{\alpha}(b_1)$. In reality, only a small portion of the passwords in the test set are closely related. On the other hand, the construction of FSB results in approximately equal probabilities that each password in the bucket was chosen, given knowledge of the bucket. We expect that the success rate for the correlated attack against FSB is higher than that of the single-query attack because the reordering helps the attacker guess correctly in the test cases where the two sampled passwords are similar.

We believe the error in estimation is amplified in the attack algorithm, which leads to a degradation in performance. If the attacker knew τ perfectly and could calculate the exact values in Equation (B.1), the correlated-query attack would perform better than the single-query attack. However, in reality, even if we know that two queries came from the same user, it is difficult to characterize the exact correlation between the two queries. If the estimate is wrong, then the success of the correlated-query attack will not necessarily be better than that of the single-query attack. Given that our attack did not show a substantial advantage for attackers, it is still an open question to analyze how damaging attacks on correlated queries can be.

APPENDIX C

APPENDIX - MIGHT I GET PWNED

C.1 Assessing performance feasibility

We compare the performance of the top four similarity measures — P2P, Das-R, wEdit and PPSM — to understand the feasibility of their deployment as a C3 service. The three generative algorithms perform a PMT with the breached passwords and their variants generated on the server. PPSM computes similarity by mapping the passwords to 100-dimensional vectors and comparing their dot product to a threshold. The resultant list of boolean is summed and sent to the client. Therefore, the implementation of PPSM-based MIGP doesn't require the generation and storage of similar passwords, but involves computing private dot products, comparisons, and summation.

To estimate the cost of performing similarity matching, we use a bucket of 10,000 username, password pairs (without any variants). We use $n = 100$ for P2P, Das-R, and wEdit, and $\theta = 0.83$ for PPSM. The OPRF based PMT protocol is implemented in Python and uses secp256k1 elliptic curve implemented in petlib [20]. We implement the PPSM based protocol using Crypten [80]. Timing experiments were performed on a machine with an Intel Core i9 processor and

Similarity measure	Latency	B/w	Compat.	Storage per bucket	Precomp.
wEdit ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	41 sec
P2P ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	180 sec
Das-R ($n = 100$)	< 1 sec	14 MB	Yes	14 MB	0.5 sec
PPSM ($\theta = 0.83$)	16 sec	1.6 KB	No	8 MB	1 sec

Figure C.1: Performance (latency, bandwidth, storage, etc.) summary of different similarity measures. All the numbers are based on a bucket of size 10^4. The trade-offs are also ranked left to right based on the importance to deployment. Here latency does not include n/w or i/o cost.

128 GB RAM, and here we run the entire protocol within the same machine (without network overhead). P2P uses an Nvidia GTX 1080 GPU along with the processor to run the pass2path neural network for pre-processing.

We summarize the results in Figure C.1. PPSM-based approach to MIGP takes 16 seconds to complete a query, while all other approaches take < 1 second. Note that these measurements, which do not include network latency, should be considered lower bounds on performance. Crypten uses secret-sharing to execute the MPC protocols, therefore requires more than one round trip. The columns of Figure C.1 are ordered from left to right in decreasing order of our perception of how critical this aspect of the protocols is to deployment. As PPSM is slower than other approaches for executing a query, we focus on the generative methods. We leave as an open question whether one can make another 2PC-based protocol fast enough for reasonably sized buckets.

C.2 Rules-based similar passwords generation

We used three rule-based approaches for generating similar passwords: Das [50], a reordered variant of Das which we call Das-R, and wEdit. The top-performing edit rules based on our dataset S_1 are shown in Figure C.2. We also report the percentage of vulnerable password pairs in T explained by each rule. Deleting characters towards the end, and adding SHIFT or CAPS LOCK at the beginning are the most common rules that users use to modify their passwords. While the top rules capture the common transformations, it fails for subtle edits that are otherwise guessed by pass2path. Some example of such

Das-R	wEdit	Rule	(%) of matches
1	1	Del last char	27.7
2	3	Switch 1 char case	20.6
3	2	Del last 2 char	15.2
6	4	Ins '1' at end	13.4
-	6	Ins 'Caps' at beg	7.6
4	5	Del last 3 char	6.6
9	7	Del 1 char	4.7
5	-	Ins '0' at beg	1.5
-	8	Subs '1' at end	1.0
10	-	Ins '0' at end	0.7
-	10	Ins '123' at end	0.5
7	9	Ins 'a' at beg	0.4
8	-	Ins 'q' at beg	0.1

Figure C.2: Rules for generating password variants and the % of password pairs matched by the rule among 9,141 vulnerable pairs found in a randomly sampled 10^5 password pairs. We also show their ranks according to Das-R and wEdit.

pairs are: ('20041981', '200481'), ('thingsome', 'thing.some'), ('nikaprudova', 'nika_prudova'), ('MADRE000', 'padre000'), ('jessiemax1', 'jessie1').

C.3 MIGP security games

Let $\mathcal{W}$ be a set of all possible strings up to some length (say, 30), and $\mathcal{W} \subseteq \mathcal{W}$ be the set containing all possible password strings for users U. We also assign a probability distribution p to all the passwords $\mathcal{W}$ representing the probability of a user $u \in U$ selecting the password $w \in \mathcal{W}$. Recall we defined $\tau : \mathcal{W} \to 2^{\mathcal{W}}$ such that $\tau(w)$ is the set of passwords defined to be similar to w. (In Section 4.4 we show how to instantiate the function τ.) Therefore, if $\tilde{w} \in \tau(w)$, for some leaked password w for a user u, then the MIGP service will respond with "similar" when queried with $(u, \tilde{w})$. We will assume $w \notin \tau(w)$, and $n = \max_w |\tau(w)|$.

The MIGPGuess adversary tries to guess the exact target password w given access to the MIGP$(\cdot)$ oracle. We modify the game in Figure 4.7 to fit the reduction we will use next. The advantage of an attacker is measured as the expected

```
MIGP($w'$)
    $q \leftarrow q + 1$
    if $w' = w$ then return match
    if $w' \in \tau(w)$ then return similar
    else return none
```

```
MIGPGuess($\mathcal{A}'$):
    $w \leftarrow_p \mathcal{W};\ q \leftarrow 0$
    $\tilde{w} \leftarrow \mathcal{A}'^{\mathsf{MIGP}}$
    if $\tilde{w} = w$ then return $q$
    else return $|\mathcal{W}|$
```

Figure C.3: Slightly modified games from Figure 4.7, instead of outputting true or false, MIGPGuess outputs the guess rank.

number of queries to the respective oracles (See Figure C.3) to guess a password, without any limit on the query budget. This is also known as α-guesswork (G_α) with $\alpha = 1$ [39]. G_α measures the expected number of guesses to ensure the probability of guessing a randomly chosen password is at least α. An alternate approach to compute attack success is using q-success rate (λ_q), which measures the probability of guessing a password given q guesses. As shown in [39], these two notions are equivalent in the sense that the guessing strategy that minimizes G_α will also maximize λ_q (if $\alpha \geq \lambda_q$).

The MIGPGuess adversary tries to guess the exact target password w given access to the MIGP$(\cdot)$ oracle. The advantage of the MIGPGuess adversary $\mathcal{A}'$ is defined as the expected guess rank:

$$G^{\mathsf{MIGP}}(\mathcal{A}') = \mathbb{E}\left[\,\mathsf{MIGPGuess}(\mathcal{A}')\,\right].$$

An MIGPGuess-adversary $\mathcal{A}'^*$ is optimal if for all $\mathcal{A}'$ it holds that $G^{\mathsf{MIGP}}(\mathcal{A}'^*) \leq G^{\mathsf{MIGP}}(\mathcal{A}')$. The optimal adversary $\mathcal{A}'^*$ builds a ternary decision tree to query MIGP such that the expected guess rank is minimized. We show that building such a decision tree that minimizes the guesswork is NP-hard, and therefore the optimal attack against MIGP is also NP-hard.

Definition 1 (OMIGP). *Given $(\mathcal{W}, p, \tau)$, we define optimal MIGP guess (OMIGP) problem as building the query tree for $\mathcal{A}'^*$ that minimizes the guesswork for distribution p over $\mathcal{W}$ with MIGP similarity measure being τ.*

Theorem 4. *OMIGP problem is NP-hard.*

Proof: To prove this result, one might be tempted to reuse a result from Chatterjee et al. [44,46], who investigated a similar setting in the context of typo-tolerant password checking. They formalized guessing attacks against a server that allows the user to login with a small set of typos. Although their setting is similar to ours, there is one crucial difference: MIGP reveals whether the query is a match or is similar to a password, but the password typo correction oracle does not and reveal whether the password was an exact or near match. This seemingly minor distinction means we can't use their techniques, and we had to find another proof strategy.

We show that we can reduce the optimal binary decision tree (OBDT) problem to OMIGP in polynomial time. Because OBDT does not have a polynomial time solution [85] then OMIGP also cannot have a polynomial time solution.

Binary decision tree (BDT). The instance of binary decision tree (BDT) is defined as a three tuple (X, p_X, Q), where X be the set of n items $x_1, x_2, \ldots, x_n$, p_X is a probability distribution defined over the elements of X, and Q is a set of m questions (functions) such that $Q_i : X \mapsto \{0, 1\}$. The goal is to find a decision tree that completely characterizes X using the questions in Q as decision nodes. The questions $q \in Q$ are specified in all the internal nodes while the items x_i fill the root nodes of the tree. The expected depth of the tree is defined as $\sum_{x \in X} p_X(x) \cdot d(x)$, where $d(x)$ is the depth — distance from the root — of the element x in the binary tree.

Definition 2 (Optimal BDT problem). *Given (X, p_X, Q), the problem is to build a binary decision tree that has the least expected depth, $\sum_{x \in X} p_X(x) \cdot d(x)$.*

Theorem 5 ([85]). *OBDT problem is NP-hard.*

Proof: This follows from the proof provided by the classic results of Laurent and Rivest [85].

Theorem 6. *OMIGP problem is NP-hard.*

Proof: We show that OMIGP is NP-hard by giving a polynomial time reduction of an arbitrary instance of OBDT problem to OMIGP. That means if there exists a polynomial time solution to OMIGP, then we can solve OBDT in polynomial time as well, which is known to be NP-hard [85].

An instance of OMIGP problem is defined as $(\mathcal{W}, p, \tau)$, where $\mathcal{W}$ are the set of strings, p is a probability distribution over $\mathcal{W}$, and τ is a similarity measure. Given an instance of BDT problem (X, p_X, Q), we can construct an instance of OMIGP problem as follows. For this we set $\mathcal{W} = X \cup Q$ (assuming there is a unique way to represent elements in Q distinct from X); $p(w) = p_X(w)$ if $w \in X$, and 0 otherwise; and $\tau(w) = \{y \mid Q_w(y) = 1\}$ if $w \in Q$, and $\varnothing$ otherwise. Given an optimal ternary decision tree T for a OMIGP problem instance $(\mathcal{W}, p, B)$, we can build a optimal BDT for the problem instance (X, p_X, Q) in polynomial time. As T is the ternary decision tree for OMIGP problem, each node has three children for each type of MIGP output. The leaf nodes of the tree are the passwords in $\mathcal{W}$. The distance of a password w from the root is the number of queries it take to guess w, which we denote as $d(w)$ here. As T is optimal, the guesswork $\sum_{w\in\mathcal{W}} p(w) \cdot d(w)$ is minimum. Also note that, because $p(w) = 0$ if $w \in Q$ (as per the reduction above), $\sum_{w\in X} p(w) \cdot d(w)$ is minimum. This is the same as the property of OBDT. Therefore, we can build the required binary decision tree T' by removing the edges for the exact match of the questions (where $w \in Q$).

Thus we show, we can reduce an instance of OBDT problem into an instance of OMIGP problem, and the solution of OMIGP will provide a solution to OBDT. This contradicts the fact that OBDT is NP-hard, therefore, OMIGP cannot have a polynomial time solution. This concludes the proof. ▮

The BDT problem will have a unique solution only if $m \geq \log_2 n$ and no two objects have the same output for all the questions. Since BDT reduces to OMIGP, the same conditions apply for OMIGP as well.

C.4 Greedy approximation of OMIGP

As shown that finding an optimal guessing strategy that minimizes the expected guess rank is NP-hard. However, attackers could still find approximate solutions that minimize the expected guess rank. We give one such greedy algorithm for OMIGP, which we call GreedyOMIGP, in Figure C.4 based on the greedy algorithm GreedyOBDT for OBDT (shown in Figure C.4 provided by Chakaravarthy et al. [43]). Chakaravarthy et al. also has shown that the approximation factor for the greedy algorithm GreedyOBDT is $\mathcal{O}(() \log |X|)$.

We can reduce a problem instance of finding optimal decision tree for OMIGP to OBDT. This is quite straightforward. We already show the reduc-

```
GreedyOBDT($X, p_X, Q$):
for $i \in \{0,1\}$ and $j \in \{1,\ldots,|Q|\}$ do
    $X_j^i \leftarrow \{x \in X : Q_j(x) = i\}$
T $\leftarrow \varnothing$
if $|X| = 1$ then   return $X$
$j^* \leftarrow \arg\max_j p_X(X_j^0) \cdot p_X(X_j^1)$
T.root $\leftarrow j^*$
T.left $\leftarrow$GreedyOBDT($X_{j^*}^0, p_X, Q$)
T.right$\leftarrow$GreedyOBDT($X_{j^*}^1, p_X, Q$)
return T

GreedyOMIGP($\mathcal{W}, p, \tau$):
T $\leftarrow \varnothing$
if $|W| = 1$ then  return $W$
$j^* \leftarrow \arg\max_j p(B(w_j))(1 - p(B(w_j)))$
T.root $\leftarrow j^*$;   T.middle $\leftarrow j^*$
$W' \leftarrow W \setminus B(w_{j^*})$
T.left $\leftarrow$ GreedyOMIGP($B(w_{j^*}), p, \tau$)
T.right $\leftarrow$ GreedyOMIGP($W', p, \tau$)
return T
```

```
GreedyMIGP($\mathcal{W}, p, , q$):
$\mathcal{W}' \leftarrow \bigcup_{w \in \mathcal{W}} \tau(w) \cup \{w\}$
for $i \leftarrow 1$ to $q$ do
    $\tilde{w}_i \leftarrow \arg\max_{\tilde{w} \in W'} p(B(\tilde{w}))$
    $r \leftarrow$ MIGP($\tilde{w}_i$)
    if $r =$ none then
        $\mathcal{W} \leftarrow \mathcal{W} \backslash B(\tilde{w}_i)$
        for $w \in B(\tilde{w}_i)$ do
            $p(w) \leftarrow 0$
    else if $r =$ similar then
        for $w \in \mathcal{W} \backslash B(\tilde{w}_i)$ do
            $p(w) \leftarrow 0$
        $\mathcal{W} \leftarrow B(\tilde{w}_i)$
        $\mathcal{W}' \leftarrow \bigcup_{w \in \mathcal{W}} \tau(w) \cup \{w\}$
    else if $r =$ match then
        return $\tilde{w}_i$
    $\mathcal{W}' \leftarrow \mathcal{W}' \setminus \{\tilde{w}_i\}$
return $\arg\max_{w \in W} p(w)$
```

Figure C.4: (**Left**) Greedy algorithm for optimal binary decision tree and optimal MIGP. Here $B : \mathcal{W} \mapsto 2^{\mathcal{W}}$, is a function such that $B(\tilde{w}) = \{w \in \mathcal{W} \,\middle|\, \tilde{w} \in \tau(w)\}$. (**Right**) Greedy algorithm for finding q guesses to a MIGP oracle (Figure C.3). Here $B(w') = \{w \in \mathcal{W} \,\middle|\, w' \in \tau(w)\}$ for any $w' \in \mathcal{W}'$.

tion for problem instance of the OBDT to OMIGP in the paper. Therefore, these two problems reduce in an approximation-preserving manner [49], i.e. Any α approximation algorithm for optimal decision tree yields an α approximation algorithm for GreedyOMIGP. Hence, the greedy algorithm GreedyOMIGP also yields the same approximation bound as GreedyOBDT problem.

C.4.1 Greedy heuristic for breach extraction

We present another greedy algorithm GreedyMIGP Figure C.4 which is equivalent to GreedyOMIGP. We define the *ball* $B(\cdot)$ of a variant $\tilde{w} \in \mathcal{W}$ as the set of passwords that share a common variant. That is, $B(\tilde{w}) = \{w \in \mathcal{W} \,\middle|\, \tilde{w} \in \tau(w)\}$.

The probability of a ball $p(B(\tilde{w}))$, also called the weight of a ball, is the sum of the probabilities of the passwords in the ball.

The attacker begins with a set of potential passwords $\mathcal{W}$ of the target user. In iteration i, the attacker picks the guess $\tilde{w}_i$ that has the highest ball weight, and based on the response from the MIGP oracle, it updates the set of potential passwords. In particular, if the response is none, then it removes all the passwords in $B(\tilde{w}_i)$ from $\mathcal{W}$. If the response is similar, then it knows that the target password is one of the passwords in $B(\tilde{w}_i)$, and so it sets the probability of all other passwords to zero and limits the search to the passwords in $B(\tilde{w}_i)$ and their variants. It is important to leave the variants in because they may have ball weight higher than all balls centered on passwords from $B(\tilde{w}_i)$. If the response is match, then it stops and outputs the guess $\tilde{w}_i$ (and wins the game). The greedy algorithm is not optimal but provides a good approximation of the success of the optimal attacker. Whether an efficient algorithm with tighter approximation bounds exists remains an open question.

We show that the expected guesswork due to the greedy algorithm GreedyMIGP is at most $\mathcal{O}(() \log |\mathcal{W}|)$ factor of the minimum expected guesswork $G^{\mathsf{MIGP}}(\mathcal{A}'^{*})$. This approximation factor is quite large, especially when $|\mathcal{W}|$ is very large. Nevertheless, this shows that it is possible to compute approximate solutions that might help an attacker guess a user's leaked password stored in MIGP server more effectively (than existing C3 services).

The complexity of the algorithm is $\mathcal{O}(()q|\mathcal{W}|)$, assuming there is a constant time algorithm to find the ball of a string. Note the attacker can decide on $\mathcal{W}$ that they believe will likely contain the target password. This could be popular passwords from prior public password breaches.

C.5 Breach extraction attack (contd.)

In Figure 4.8, we show the breach extraction attack success from a random sample of 25,000 username, password pairs from T. This includes passwords that are weak, such as the one blocked by MIGP. We therefore investigate the attack success against users who uses passwords not present in the blocklisted set. We sample target passwords ensuring that they do not belong to the set of passwords blocklisted by MIGP. The results are shown in Figure C.5.

β	n	$q = 10$	$q = 100$	$q = 1000$
0	0	1.71 ($\pm$ 0.22)	3.36 ($\pm$ 0.34)	6.57 ($\pm$ 0.61)
	10	2.20 ($\pm$ 0.24)	4.97 ($\pm$ 0.31)	13.38 ($\pm$ 0.54)
	10^2	1.79 ($\pm$ 0.13)	4.87 ($\pm$ 0.18)	17.18 ($\pm$ 0.38)
10	0	1.21 ($\pm$ 0.17)	2.48 ($\pm$ 0.25)	5.64 ($\pm$ 0.51)
	10	**0.14** ($\pm$ 0.06)	1.85 ($\pm$ 0.15)	9.50 ($\pm$ 0.31)
	10^2	**0.03** ($\pm$ 0.02)	0.44 ($\pm$ 0.08)	10.65 ($\pm$ 0.16)
10^2	0	0.78 ($\pm$ 0.10)	1.40 ($\pm$ 0.17)	2.49 ($\pm$ 0.30)
	10	0.12 ($\pm$ 0.02)	1.78 ($\pm$ 0.28)	8.57 ($\pm$ 0.47)
	10^2	0.03 ($\pm$ 0.03)	0.67 ($\pm$ 0.13)	6.46 ($\pm$ 0.27)
10^3	0	0.72 ($\pm$ 0.07)	0.89 ($\pm$ 0.08)	1.46 ($\pm$ 0.14)
	10	**0.23** ($\pm$ 0.02)	0.78 ($\pm$ 0.12)	5.61 ($\pm$ 0.54)
	10^2	0.04 ($\pm$ 0.01)	0.09 ($\pm$ 0.03)	2.07 ($\pm$ 0.27)
10^4	0	$<$ 0.01 ($\pm <$ 0.01)	0.03 ($\pm$ 0.03)	0.27 ($\pm$ 0.03)
	10	$<$ 0.01 ($\pm <$ 0.01)	0.41 ($\pm$ 0.07)	2.98 ($\pm$ 0.25)
	10^2	0.00 ($\pm$ 0.00)	0.26 ($\pm$ 0.21)	**2.56** ($\pm$ 0.20)

Figure C.5: Breach extraction attack success when the target password is not one of the blocked passwords or their variants.

C.6 Rate-Limiting Client Queries

As shown in Section 4.5.1, to reduce the effect of the breach extraction attacks, MIGP must limit access to the service. Cryptographic rate-limiting ensures the client performs significantly more work than the server to make a query.

MIGP can use a slow, computationally expensive hash function such as Argon2 [4] or Scrypt [108] for H_2 (or H_1, see the trade offs in Section 4.7) in the PRF F_κ. For example, computing a slow Argon2 hash with default parameters [18] on a desktop with Intel Core i9 processor and 128 GB RAM takes about 97 ms. However, this also requires the server to compute the slow hash during pre-processing. We estimate that computing F_κ for 1.14 billion unique username, password pairs and their $n = 100$ variants will require approximately 361.5 CPU-years of computational power.

An alternative approach would be to use a time-lock puzzle [93, 114] to slow down client queries to MIGP. Time-lock puzzles, first introduced by Rivest et al. [114], are a type of verifiable delay function (VDF) [38], where knowledge of trapdoor information makes computing a hash function significantly faster. Following the construction in [114], we can set H_2 to be computed as follows. The MIGP server computes a large RSA modulus $N = pq$, where p and q are two large randomly chosen secret primes. Let ν be the cost factor and $\mathsf{H}_2(x) = \mathrm{SHA256}(x)^{2^\nu} \bmod N$, for any binary string $x \in \{0, 1\}^*$. The server can compute H_2 efficiently as $\mathsf{H}_2(x) = \mathrm{SHA256}(x)^{2^\nu \bmod \phi(N)} \bmod N$, where $\phi(N) = (p-1) \cdot (q-1)$. The time complexity of such an operation would be bounded by the size of N in bits. While for the client, which will not know the factors of N, computation of H_2 will need to perform ν squaring modulo N sequentially (each time squaring the prior result). By setting the value of ν accordingly the server can increase the computational cost. The advantage of using time lock puzzles is that repeated squaring is an intrinsically sequential process and can't be parallelized. We estimate that the server would need 0.53 ms to set up a time-lock puzzle that would take 100 ms to solve for the client. This corresponds to approximately 1.8 CPU-years of computational power to finish computing H_2 of 1.14 billion unique username-password pairs and their $n = 100$ variations.

A third alternative approach to throttle client queries is to add a small secret value to the hash function H_2, $\mathsf{H}_2(x) = \mathrm{SHA256}(x \| r)$, where r is randomly chosen from $\{0, 1\}^\nu$ for each username, password pair. The server does not store r (or share it with the client). Therefore, the client has to brute-force the value of r. For example, assuming a (malicious) client can do 10 million $\mathrm{SHA256}$ hashes per second, the server can set the value of ν to be 21 bits, which will in expectation ensure 100 ms client-side computing cost. The server will require approximately 3.1 CPU-hour for precomputation. One drawback of this approach is that the client can parallelize the computation of hashes, and it does not guarantee 2^ν sequential operations, unlike time-lock puzzles.

BIBLIOGRAPHY

[1] Lastpass. `https://lastpass.com`.

[2] 4iQ. `https://4iq.com/`, 2018.

[3] Argon2. `https://www.npmjs.com/package/argon2/`, 2018.

[4] Awslambda. `https://aws.amazon.com/lambda/`, 2018.

[5] Cloudfront. `https://aws.amazon.com/cloudfront/`, 2018.

[6] Dynamodb. `https://aws.amazon.com/dynamodb/`, 2018.

[7] Finding pwned passwords with 1Password. `https://blog.1password.com/finding-pwned-passwords-with-1password/`, 2018.

[8] Have I been pwned: API v2. `https://haveibeenpwned.com/API/v2`, 2018.

[9] I wanna go fast: Why searching through 500m pwned passwords is so quick. `https://www.troyhunt.com/i-wanna-go-fast-why-searching-through-500m-pwned-passwords-is-so`
2018.

[10] Interval tree. `https://en.wikipedia.org/wiki/Interval_tree`, 2018.

[11] List of data breaches. `https://en.wikipedia.org/wiki/List_of_data_breaches`, 2018.

[12] Security update - q2 2018. `https://www.eveonline.com/article/pc29kq/an-update-on-security-the-fight-against-bots-and-rmt`, 2018.

[13] Twitter's list of 370 banned passwords. `http://www.businessinsider.com/twitters-list-of-370-banned-passwords-2009-1`
2018. Accessed: 2015-11-06.

[14] Crypto nodejs. https://nodejs.org/api/crypto.html, 2019.

[15] GhostProject. https://ghostproject.fr/, 2019.

[16] Testing Firefox Monitor, a new security tool. https://blog.mozilla.org/futurereleases/2018/06/25/testing-firefox-monitor-a-new-security-tool/, 2019.

[17] Vericlouds. https://my.vericlouds.com/, 2019.

[18] CFFI-based Argon2 Bindings for Python. https://argon2-cffi.readthedocs.io/en/stable/, 2020.

[19] Enzoic: Credential API. https://haveibeenpwned.com/Passwords/, 2020.

[20] A Python library that implements a number of Privacy Enhancing Technologies (PETs). https://github.com/gdanezis/petlib, 2020.

[21] Requests: HTTP for Humans. https://docs.python-requests.org/en/master/, 2020.

[22] Flask - The Python micro framework for building web applications. https://github.com/pallets/flask, 2021.

[23] MIGP library. https://github.com/cloudflare/migp-go, October 2021. Accessed: 2022-02-04.

[24] Privacy-preserving compromised credential checking. https://blog.cloudflare.com/privacy-preserving-compromised-credential-checking/, 2021.

[25] Cloudflare Workers. https://workers.cloudflare.com/, February 2022. Accessed: 2022-02-04.

[26] How kv works. https://developers.cloudflare.com/workers/learning/how-kv-works, February 2022. Accessed: 2022-02-04.

[27] Usage statistics of reverse proxy services for websites. https://w3techs.com/technologies/overview/proxy, Feb 2022.

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

[29] 4iQ. Identities in the Wild: The Tsunami of Breached Identities Continues. `https://4iq.com/wp-content/uploads/2018/05/2018_IdentityBreachReport_4iQ.pdf/`, 2018.

[30] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[31] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[32] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[33] Junade Ali. Optimising caching on pwned passwords (with workers). `https://blog.cloudflare.com/optimising-caching-on-pwnedpasswords`, 2018.

[34] Junade Ali. Validating Leaked Passwords with k-Anonymity. `https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/`, 2018.

[35] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin. On weighted balls-into-bins games. *Theoretical Computer Science*, 409(3):511–520, 2008.

[36] Alex Bocharov. Cloudflare bot management: machine learning and more. `https://blog.cloudflare.com/cloudflare-bot-management-machine-learning-and-more/`, 2020.

[37] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[38] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[39] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy (SP)*, pages 538–552. IEEE, 2012.

[40] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy*, 2012.

[41] William E Burr, Donna F Dodson, and William T Polk. *Electronic authentication guideline*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2004.

[42] Julio 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-databa` Dec, 2017.

[43] Venkatesan T Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, Pranjal Awasthi, and Mukesh Mohania. Decision trees for entity identification: Approximation algorithms and hardness results. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–62, 2007.

[44] Rahul Chatterjee, Anish Athalye, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. password typos and how to correct them securely. *IEEE Symposium on Security and Privacy*, 2016.

[45] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 481–498. IEEE, 2015.

[46] Rahul Chatterjee, Joanne Woodage, Yuval Pnueli, Anusha Chowdhury, and Thomas Ristenpart. The typtop system: Personalized typo-tolerant password checking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 329–346. ACM, 2017.

[47] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255. ACM, 2017.

[48] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[49] Pierluigi Crescenzi. A short guide to approximation preserving reductions. In *Proceedings of Computational Complexity. Twelfth Annual IEEE Conference*, pages 262–273. IEEE, 1997.

[50] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[51] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on PETS*, 2018(3):164–180, 2018.

[52] Xavier De Carné De Carnavalet, Mohammad Mannan, et al. From very weak to very strong: Analyzing password-strength meters. In *NDSS*, volume 14, pages 23–26, 2014.

[53] Matteo Dell'Amico and Maurizio Filippone. Monte carlo strength evaluation: Fast and reliable password checking. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 158–169. ACM, 2015.

[54] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: An empirical analysis. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[55] Martin MA Devillers. Analyzing password strength. *Radboud University Nijmegen, Tech. Rep*, 2010.

[56] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In C. Cachin and J. Camenisch, editors, *Eurocrypt 2004*, pages 523–540. Springer-Verlag, 2004. LNCS no. 3027.

[57] Zeev Dvir and Sivakanth Gopi. 2-server pir with sub-polynomial communication. In *Proceedings of the forty-seventh Annual ACM Symposium on the Theory of Computing*, pages 577–584. ACM, 2015.

[58] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology – CRYPTO 1992*, pages 139–147. Springer, 1992.

[59] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[60] Verizon Enterprise. 2017 data breach investigations report, 2017.

[61] Verizon Enterprise. 2020 Databreach Investigation Report. `https://enterprise.verizon.com/resources/reports/2020/2020-data-breach-investigations-report.pdf`, 2020.

[62] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, Thomas Ristenpart, and Cornell Tech. The Pythia PRF service. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 547–562. USENIX Association, 2015.

[63] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[64] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptography–EUROCRYPT*, pages 1–19. Springer, 2004.

[65] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[66] Paul A Grassi, JL Fenton, EM Newton, RA Perlner, AR Regenscheid, WE Burr, JP Richer, NB Lefkovitz, JM Danker, YY Choong, et al. Nist special publication 800-63b. digital identity guidelines: Authentication and lifecycle management. *Bericht, NIST*, 2017.

[67] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. `https://www.wired.com/story/collection-leak-usernames-passwords-billions/`, 2018.

[68] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, pages 155–175. Springer, 2008.

[69] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[70] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. PassGAN: A deep learning approach for password guessing. *arXiv preprint arXiv:1709.00440*, 2017.

[71] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[72] Thomas Icart. How to hash into elliptic curves. In *Annual International Cryptology Conference*, pages 303–316. Springer, 2009.

[73] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure information networks*, pages 258–272. Springer, 1999.

[74] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. In *ASIACRYPT*, pages 233–253. Springer, 2014.

[75] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[76] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[77] Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. Password Monitor: Safeguarding passwords in Microsoft Edge. `https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/`, 2021.

[78] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[79] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017(4):177–197, 2017.

[80] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[81] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829. ACM, 2016.

[82] Saranga Komanduri. Modeling the adversary to evaluate password strength with limited samples. 2016.

[83] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[84] Marie-Sarah Lacharité and Kenneth G Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018(1):277–313, 2018.

[85] Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[86] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[87] Chun-Ta Li and Min-Shiang Hwang. An efficient biometrics-based remote user authentication scheme using smart cards. *Journal of Network and computer applications*, 33(1):1–5, 2010.

[88] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM CCS*, pages 1387–1403, 2019.

[89] Xiong Li, Jianwei Niu, Muhammad Khurram Khan, and Junguo Liao. An enhanced smart card based remote user password authentication scheme. *Journal of Network and Computer Applications*, 36(5):1365–1371, 2013.

[90] Yue Li, Haining Wang, and Kun Sun. A study of personal information in human-chosen passwords and its security implications. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[91] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, pages 689–704. IEEE Computer Society, 2014.

[92] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24. IEEE, 2006.

[93] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In *Annual Cryptology Conference*, pages 39–50. Springer, 2011.

[94] Silvere Mavoungou, Georges Kaddoum, Mostafa Taha, and Georges Matar. Survey on threats and attacks on mobile networks. *IEEE Access*, 4:4543–4572, 2016.

[95] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 173–186. ACM, 2013.

[96] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.

[97] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean and accurate: Modeling password guessability using neural networks.

[98] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *IEEE Symposium on Foundations of Computer Science*, pages 80–91. IEEE, 2003.

[99] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[100] A Naranyanan and V Shmatikov. Robust de-anonymization of large datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008*, 2008.

[101] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 364–372. ACM, 2005.

[102] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *International Conference on Financial Cryptography and Data Security*, pages 158–172. Springer, 2011.

[103] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238. Springer, 1999.

[104] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity using neural networks. *IEEE Symposium on Security and Privacy*, 2019.

[105] Bijeeta Pal, Mazharul Islam, Marina Sanusi, Nick Sullivan, Luke Valenta, Tara Whalen, Christopher Wood, Thomas Ristenpart, and Rahul Chattejee. Might i get pwned: A second generation compromised credential checking service.

[106] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain 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*, pages 295–310. ACM, 2017.

[107] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[108] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. 2015.

[109] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015.

[110] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[111] Jennifer Pullman, Kurt Thomas, and Elie Bursztein. Protect your accounts from data breaches with Password Checkup. `https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html`, 2019.

[112] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

[113] Josyula R Rao, Pankaj Rohatgi, Helmut Scherzer, and Stephane Tinguely. Partitioning attacks: or how to rapidly clone some gsm cards. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 31–41. IEEE, 2002.

[114] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

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

[116] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[117] Frank Stajano. Pico: No more passwords! In *International Workshop on Security Protocols*, pages 49–81. Springer, 2011.

[118] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[119] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[120] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[121] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*. USENIX, 2019.

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

[123] Nirvan Tyagi, Sofí Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious prf, with applications. In *Advances in Cryptography–EUROCRYPT*. Springer, 2022.

[124] Blase Ur, Felicia Alfieri, Maung Aung, Lujo Bauer, Nicolas Christin, Jessica Colnago, Lorrie Faith Cranor, Henry Dixon, Pardis Emami Naeini, Hana 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*, pages 3775–3786. ACM, 2017.

[125] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[126] Ding Wang, Haibo Cheng, Ping Wang, Xinyi Huang, and Gaopeng Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11):2776–2791, 2017.

[127] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1242–1254. ACM, 2016.

[128] Ke Coby Wang and Michael K. Reiter. How to end password reuse on the web. In *26th Annual Network and Distributed System Security Symposium*. Internet Society, Feb 2019.

[129] Ke Coby Wang and Michael K. Reiter. Detecting stuffing of a user's credentials at her own accounts. In *29th USENIX Security Symposium*. USENIX Association, Aug 2020.

[130] 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)*, pages 162–175, 2009.

[131] Dan Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *Proc. USENIX Security*, 2016.

[132] Wikipedia. Email address. 2018.

[133] Christopher Makoto Wilt, Jordan Tyler Thayer, and Wheeler Ruml. A comparison of greedy search algorithms. In *Third Annual Symposium on Combinatorial Search*, 2010.

[134] Joanne Woodage, Rahul Chatterjee, Yevgeniy Dodis, Ari Juels, and Thomas Ristenpart. A new distribution-sensitive secure sketch and popularity-proportional hashing. In *Annual International Cryptology Conference*, pages 682–710. Springer, 2017.

[135] Lei Zhang, Sushil Jajodia, and Alexander Brodsky. Information disclosure under realistic assumptions: Privacy versus optimality. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 573–583. ACM, 2007.

[136] 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)*, pages 176–186, 2010.