Protocols for Checking Compromised Credentials

Lucy Li
Cornell University

Bijeeta Pal
Cornell University

Junade Ali
Cloudflare Inc.

Nick Sullivan
Cloudflare Inc.

Rahul Chatterjee
University of Wisconsin–Madison & Cornell Tech

Thomas Ristenpart
Cornell Tech

ABSTRACT

To prevent credential stuffing attacks, industry best practice now proactively checks if user credentials are present in known data breaches. Recently, some web services, such as HaveIBeenPwned (HIBP) and Google Password Checkup (GPC), have started providing APIs to check for breached passwords. We refer to such services as *compromised credential checking* (C3) services. We give the first formal description of C3 services, detailing different settings and operational requirements, and we give relevant threat models.

One key security requirement is the secrecy of a user's passwords that are being checked. Current widely deployed C3 services have the user share a small prefix of a hash computed over the user's password. We provide a framework for empirically analyzing the leakage of such protocols, showing that in some contexts knowing the hash prefixes leads to a 12x increase in the efficacy of remote guessing attacks. We propose two new protocols that provide stronger protection for users' passwords, implement them, and show experimentally that they remain practical to deploy.

CCS CONCEPTS

• **Security and privacy** → **Authentication**; *Privacy-preserving protocols.*

KEYWORDS

Passwords; authentication; privacy-preserving services

ACM Reference Format:
Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. 2019. Protocols for Checking Compromised Credentials. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3319535.3354229

1 INTRODUCTION

Password database breaches have become routine [9]. 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) [46] was deployed by Troy Hunt and CloudFlare in 2018 and is used by many web services, including Firefox [14], EVE Online [10], and 1Password [5]. Google released a Chrome extension called Password Checkup (GPC) [44, 45] 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) [29, 36], 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

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354229

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 [40] 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 [45], 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.

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 [24, 40, 47] , 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 1. A C3 *server* has access to a breach database $\tilde{\mathcal{S}}$. We can think of $\tilde{\mathcal{S}}$ as a set of size N, which consists of either a set of passwords $\{w_1, \ldots, w_N\}$ or username-password pairs $\{(u_1, w_1), \ldots, (u_N, w_N)\}$. This corresponds to two types of C3 services — *password-only C3 services* and *username-password C3 services*. For example, HIBP [6] is a password-only C3 service,[1] and Google's service GPC [44] is an example of a username-password C3 service.

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 [21]. The HIBP database has 501 million unique passwords [6]. Google's blog specifies that there are 4 billion username-password pairs in their database of leaked credentials [44].

[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.

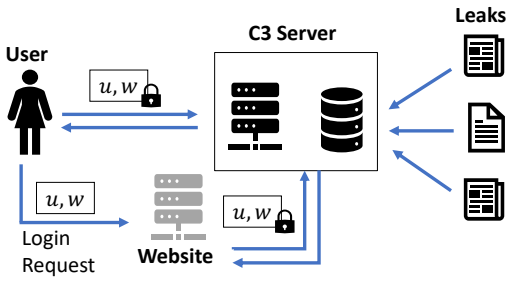

Figure 1: A C3 service allows a client to ascertain whether a username and password appear in public breaches known to the service.

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 [7]. 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 $\tilde{\mathcal{S}}$ 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 [2].

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.

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 [27]. 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 [16, 28]. 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 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) [22, 31, 42, 43] 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) [31] 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

[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.

Credentials checked	Name	Bucket identifier	B/w (KB)	RTL (ms)	Security loss
Password	HIBP	20-bits of SHA1(w)	32	220	12x
	FSB	Figure 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 2: Comparison of different C3 protocols and their bandwidth usage, round-trip latency, and security loss (compared to an attacker that has no bucket identifier information). HIBP [6] and GPC [44] 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.

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 [30, 31].

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

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, $\|\tilde{\mathcal{S}}\| = N$
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: The notation used in this paper.

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

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

Figure 4: The guessing games used to evaluate security.

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, $\alpha(b) = \{s \in \mathcal{S} \,|\, b \in \beta(s)\}$. 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 $\tilde{\alpha}(b)$.

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 [44] and HIBP [6] 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 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 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 4.

Security measure. The goal of an attacker is to learn the user's password. We will focus on online-guessing attacks, where an at-

tacker 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 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.

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

and

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

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

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

Note,

$$\Pr\left[\mathrm{Guess}^{\mathcal{A}}(q) \Rightarrow 1\right] = \sum_u \Pr\left[w \in \mathcal{A}(u, q) \wedge U = u\right].$$

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

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

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

$$\begin{aligned}\mathsf{Adv}^{\text{b-gs}}_{\beta}(q) &= \sum_u \sum_b \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr\left[W = w_i \wedge U = u \wedge B = b\right] \\ &= \sum_u \sum_b \max_{\substack{(u_1,w_1),\ldots,(u_q,w_q)\\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{\Pr\left[W = w_i \wedge U = u\right]}{|\beta((u, w_i))|}\end{aligned} \tag{2}$$

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 $\Pr[W = w \mid U = u] = \Pr[W = w]$. Assuming independence of usernames and passwords, we define in the uncompromised setting

$$\lambda_q = \mathsf{Adv}^{\mathrm{gs}}(q) = \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr\left[W = w_i\right]. \tag{3}$$

We give analytical (using Equations 2 and 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 [40, 47]. Analytical bounds on the compromised setting (using Equations 1 and 2) are less informative, so we evaluate this setting empirically in Section 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 [20], assuming the client picks a bucket randomly from the possible set of buckets $\beta(s)$ for a credential s, and $\beta(s)$ also 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 [20] 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.

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 [6] and GPC [44]. We introduce

[3] Though prior work [33, 47] suggests knowledge of the username can improve efficacy of guessing passwords, the improvement is minimal. See Appendix A for more on this.

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 [6]. 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 [6]. 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 [19] . This effectively makes the user's password k-anonymous. However, k-anonymity provides limited protection, as shown by numerous prior works [35, 38, 50] 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) [18]. 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 [44, 45]. 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 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 [45]. 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 point, and finally, exponentiates the elliptic curve point (using elliptic curve group operations) to the power a. Therefore it holds that $(F_a(x))^b = F_{ab}(x)$.

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

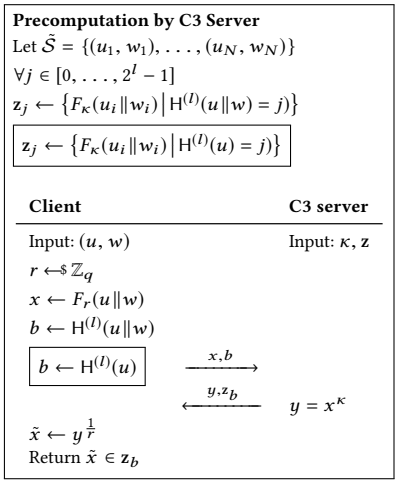

Figure 5: Algorithms for GPC, and the change in IDB given in the box. $F_{(\cdot)}(\cdot)$ **is a PRF.**

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, 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 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 4.1. *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 $|\beta_{\text{HPB}}(s)| = 1$, for any input s, as every password is assigned to exactly one of the buckets. Following the discussion from Section 3, assuming independence of usernames and passwords in the uncompromised setting, we can compute the advantage against game BucketGuess as

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

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 $\mathsf{Adv}^{\text{gs}}(q \cdot |\mathcal{B}|)$. ∎

Theorem 4.1 only provides an upper bound on the security loss. Moreover, for the compromised setting, the analytical formula in Equation (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 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 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 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 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 4.2. *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 $\text{BucketGuess}_{\text{IDB}}$ game by

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

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.

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 creden-

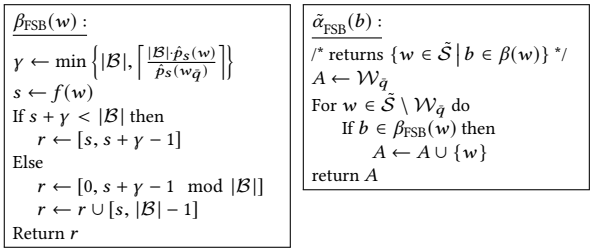

Figure 6: Bucketizing function β_{FSB} for assigning passwords to buckets in FSB. Here $\hat{p}_s$ is the distribution of passwords; $\mathcal{W}_{\bar{q}}$ is the set of top-$\bar{q}$ passwords according to $\hat{p}_s$; $\mathcal{B}$ is the set of buckets; f is a hash function $f: W \mapsto \mathbb{Z}_{|B|}$; $\tilde{\mathcal{S}}$ is the set of passwords hosted by the server.

tial 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 $q \le \bar{q}$, FSB has 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 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 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 [8] 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 $\tilde{\alpha}$.

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

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 [48] 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 [34], and they are very fast to train and run (unlike neural network based password distribution estimators, such as [37]). 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 [37, 48]. 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

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

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 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.

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 5.1. *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) $\text{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) = \text{Adv}^{\text{gs}}(q)$ *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). We include the full proof for Theorem 5.1 in Appendix C. 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 6 for $\hat{p}_s \neq p_w$, in both the compromised and uncompromised settings.

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 adversary 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.

	$\tilde{\mathcal{S}}$	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 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.

Password breach dataset. We used the same breach dataset used in [40]. The dataset was derived from a previous breach compilation [21] 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 [40] 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 [11]. For all attack simulations, the target user-password pairs are sampled from the test dataset T (or T_{sp}).

In Figure 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 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 pairs from compromised users (users with at least one password present in

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

$\tilde{\mathcal{S}}$), 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 8). For each pair (u, w), we run the games Guess and BucketGuess as specified in Figure 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 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 (1) and Equation (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,

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

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 [40]. 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 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 5.1.

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.

Results. We report the success rates of the attack simulations in Figure 8. The baseline success rate (first row) is the advantage Adv^{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 [6], 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 ac-

Protocol	Params	Bucket size		Uncompromised				Compromised			
		Avg	max	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$
Baseline	N/A	N/A	N/A	0.7 (±0.1)	1.5 (±0.1)	2.9 (±0.3)	5.8 (±0.4)	41.1 (±0.4)	51.1 (±0.8)	53.3 (±0.9)	55.7 (±1.0)
HPB	$l = 20^{\ddagger}$	413	491	32.9 (±0.5)	49.5 (±0.3)	62.5 (±0.4)	71.1 (±0.5)	67.3 (±0.8)	74.5 (±0.6)	79.4 (±0.6)	82.9 (±0.4)
	$l = 16^{\dagger}$	6602	6891	17.9 (±0.5)	33.4 (±0.6)	47.3 (±0.3)	59.7 (±0.2)	61.1 (±0.9)	67.4 (±0.8)	73.6 (±0.6)	78.2 (±0.7)
	$l = 12$	105642	106668	8.2 (±0.4)	17.5 (±0.6)	30.7 (±0.6)	44.4 (±0.4)	56.3 (±1.0)	60.8 (±1.0)	66.5 (±0.8)	72.3 (±0.6)
FSB	$\bar{q} = 1$	83	122	0.7 (±0.1)	4.7 (±0.4)	69.8 (±0.5)	71.1 (±0.5)	53.7 (±0.9)	55.7 (±0.9)	82.6 (±0.4)	83.0 (±0.4)
	$\bar{q} = 10$	852	965	0.7 (±0.1)	1.5 (±0.1)	5.3 (±0.3)	70.8 (±0.5)	52.8 (±0.9)	54.2 (±1.0)	56.0 (±0.9)	83.0 (±0.4)
	$\bar{q} = 10^2$	6299	6602	0.7 (±0.1)	1.5 (±0.1)	2.9 (±0.3)	8.0 (±0.4)	51.9 (±0.8)	53.8 (±0.9)	54.8 (±1.0)	57.1 (±1.0)
	$\bar{q} = 10^3$	25191	25718	0.7 (±1.0)	1.5 (±0.1)	2.9 (±0.3)	5.8 (±0.4)	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 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.

count 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 [11]. 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 $\tilde{\mathcal{S}}$, 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 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

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 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 $\text{Adv}^{\text{gs}}(q)$. There were 5,000 samples each from the uncompromised and compromised settings.

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 $q \leq \bar{q}$.

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 [1] for the server-side computation and Amazon DynamoDB [4] 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 [3]. (HIBP uses Cloudflare as CDN to serve more than 600,000 requests per day [7].) 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 [12].

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 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 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 $\tilde{\mathcal{S}}$ 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 [8] 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 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.

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 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.

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.

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 [44] 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 [44].

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.

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 [5] or Password Checkup [44]. 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 [24, 40, 47]. 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 D.

9 RELATED WORK

Private set intersection. The protocol task facing C3 services is private set membership, a special case of private set intersection (PSI) [29, 36]. 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. [30] proposed an efficient PSI protocol for unequal set sizes based on oblivious pseudo-random 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 [22, 31, 42, 43] have similarly prohibitive bandwidth usage.

Private information retrieval (PIR) [23] 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 [17, 39] 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 [26] than the bucketization-based C3 schemes, with communication complexity sub-polynomial 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. [45], 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 [40]), FSB represents the best known approach.

Other C3 services include, for example, Vericlouds [15] and GhostProject [13]. 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. [49], who introduced a new type of secure sketch [25] for password typos, and Lacharite et al.'s [32] frequency-smoothing encryption. While similar in that they use distributional knowledge, their constructions do not apply in our setting.

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.

ACKNOWLEDGMENTS

We would like to thank the authors of [45] for sharing their work with us prior to publication. This work was supported in part by NSF grants CNS-1564102, CNS-1514163, and CNS-1704527.

REFERENCES

[1] 2018. Argon2. https://www.npmjs.com/package/argon2/.
[2] 2018. AWSlambda. https://aws.amazon.com/lambda/.
[3] 2018. CloudFront. https://aws.amazon.com/cloudfront/.
[4] 2018. DynamoDb. https://aws.amazon.com/dynamodb/.
[5] 2018. Finding pwned passwords with 1Password. https://blog.1password.com/finding-pwned-passwords-with-1password/.
[6] 2018. Have I Been Pwned: API v2. https://haveibeenpwned.com/API/v2.
[7] 2018. 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-quick/.
[8] 2018. Interval Tree. https://en.wikipedia.org/wiki/Interval_tree.
[9] 2018. List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches.
[10] 2018. SECURITY UPDATE - Q2 2018. https://www.eveonline.com/article/pc29kq/an-update-on-security-the-fight-against-bots-and-rmt.
[11] 2018. Twitter's List Of 370 Banned Passwords. http://www.businessinsider.com/twitters-list-of-370-banned-passwords-2009-12. Accessed: 2015-11-06.
[12] 2019. Crypto Nodejs. https://nodejs.org/api/crypto.html.
[13] 2019. GhostProject. https://ghostproject.fr/.
[14] 2019. Testing Firefox Monitor, a New Security Tool. https://blog.mozilla.org/futurereleases/2018/06/25/testing-firefox-monitor-a-new-security-tool/.
[15] 2019. Vericlouds. https://my.vericlouds.com/.
[16] 4iQ. 2018. Identities in the Wild: The Tsunami of Breached Identities Continues. https://4iq.com/wp-content/uploads/2018/05/2018_IdentityBreachReport_4iQ.pdf/.
[17] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies* 2016, 2 (2016), 155–174.
[18] Junade Ali. 2018. Optimising Caching on Pwned Passwords (with Workers). https://blog.cloudflare.com/optimising-caching-on-pwnedpasswords.
[19] Junade Ali. 2018. Validating Leaked Passwords with k-Anonymity. https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/.
[20] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin. 2008. On weighted balls-into-bins games. *Theoretical Computer Science* 409, 3 (2008), 511–520.
[21] Julio Casal. Dec, 2017. 1.4 Billion Clear Text Credentials Discovered in a Single Database. https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14.
[22] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1243–1255.
[23] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 41–50.
[24] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. 2014. The Tangled Web of Password Reuse.. In *NDSS*, Vol. 14. 23–26.
[25] Y. Dodis, L. Reyzin, and A. Smith. 2004. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In *Eurocrypt 2004*, C. Cachin and J. Camenisch (Eds.). Springer-Verlag, 523–540. LNCS no. 3027.
[26] Zeev Dvir and Sivakanth Gopi. 2015. 2-server PIR with sub-polynomial communication. In *Proceedings of the forty-seventh Annual ACM Symposium on the Theory of Computing*. ACM, 577–584.
[27] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. 2018. I never signed up for this! Privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies* 2018, 1 (2018), 109–126.
[28] Verizon Enterprise. 2017. 2017 Data breach investigations report.
[29] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient private matching and set intersection. In *Advances in Cryptography–EUROCRYPT*. Springer, 1–19.
[30] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. 2017. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies* 2017, 4 (2017), 177–197.
[31] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient batched oblivious PRF with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 818–829.
[32] Marie-Sarah Lacharité and Kenneth G Paterson. 2018. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology* 2018, 1 (2018), 277–313.
[33] Yue Li, Haining Wang, and Kun Sun. 2016. 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*. IEEE, 1–9.
[34] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. 2014. A Study of Probabilistic Password Models. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 689–704.
[35] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. 2006. l-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 24–24.
[36] Catherine Meadows. 1986. 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*. IEEE, 134–134.
[37] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. [n.d.]. Fast, lean and accurate: Modeling password guessability using neural networks.
[38] A Naranyanan and V Shmatikov. 2008. Robust de-anonymization of large datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008*.
[39] Femi Olumofin and Ian Goldberg. 2011. Revisiting the computational practicality of private information retrieval. In *International Conference on Financial Cryptography and Data Security*. Springer, 158–172.

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

Figure 11: Statistics on samples with low edit distance between username and password, as a percentage of a random sample of 10^5 username-password pairs.

[40] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. 2019. Beyond Credential Stuffing: Password Similarity using Neural Networks. *IEEE Symposium on Security and Privacy* (2019).

[41] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. 2017. Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 295–310.

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

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

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

[45] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. 2019. Protecting Accounts from Credential Stuffing with Password Breach Alerting. In *USENIX Security Symposium*. USENIX.

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

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

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

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

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

A CORRELATION BETWEEN USERNAME AND PASSWORDS

In Section 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 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 11.

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. [47], who 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 BANDWIDTH OF FSB

To calculate the maximum bandwidth used by FSB, we use the balls-and-bins formula as described in Section 3. Each password w is stored in $|\beta(w)|$ buckets, so the total number of balls, or passwords being stored, can be calculated as

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

The first equality is obtained by replacing the definition of $\beta(w)$; the second inequality holds because $\lceil x \rceil \leq x + 1$; 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 $2 \cdot \left(\bar{q} + \frac{1}{\hat{p}_s(w_{\bar{q}})} + \frac{N}{|\mathcal{B}|} \right)$.

C PROOF OF THEOREM 5.1

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

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

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.

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

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

Figure 12: 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.

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 $\text{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q)$,

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

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

To find a lower bound on $\text{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q)$, 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 $\hat{p}_s(\cdot)$.

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

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

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

D 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 [24, 40, 41, 47]. If such passwords are checked with a C3 server (maybe by a password manager [5]), and the attacker could identify multiple 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 12. 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:

$$\begin{aligned}
&\operatorname*{argmax}_w \Pr[w_2 = w \mid b_1, b_2] \\
&\quad= \operatorname*{argmax}_w \Pr[b_1, b_2 \mid w_2 = w] \cdot \frac{\Pr[w_2 = w]}{\Pr[b_1, b_2]} \\
&\quad= \operatorname*{argmax}_w \Pr[b_1, b_2 \mid w_2 = w] \cdot \Pr[w_2 = w]\ .
\end{aligned}$$

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 $\Pr[b_1, b_2 \mid w_2 = w]$ into two parts.

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

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

We know that the first term $\Pr[b_2 \mid w_2 = w]$ 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

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 13: 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.

bucket. For HPB, only one bucket will have a nonzero probability for each password.

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

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.

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

Combining the argmax expression with the equations above, the adversary therefore needs to calculate the following to find the most likely w:

$$\begin{aligned} &\operatorname*{argmax}_{w \in \alpha(b_2)} \frac{1}{|\beta(w)| \cdot \Pr[w_2 = w]} \cdot \sum_{w_1 \in \alpha(b_1)} \Pr[w_2 = w \mid w_1] \cdot \Pr[w_1] \\ &= \operatorname*{argmax}_{w \in \alpha(b_2)} \frac{1}{|\beta(w)| \cdot \Pr[w_2 = w]} \cdot \sum_{w_1 \in \alpha(b_1)} \tau_{(u,w_1)}(w) \cdot \Pr[w_1]. \end{aligned} \tag{4}$$

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 6. To estimate $\Pr[w_2 = w \mid w_1]$, we use the password similarity measure described in [40], 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 6. We first trim the test dataset T down to users with passwords 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 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 6). It reorders the list of passwords $\tilde{\alpha}(b_2)$ using an estimate of the value in Equation (4).

The results of this simulation are in Figure 13. 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 [40] 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 (4), 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.