Combating Credential Stuffing: Protocols to Check for Compromised Credentials and Password Reuse

Mir Masood Ali
University of Illinois at Chicago
mali92@uic.edu

ABSTRACT

Password reuse is claimed to be the most prominent cause of harm on the Internet. Attackers that gain access to a user's password from a breach of one website can stuff the same credentials across multiple other services and gain access with reasonable success. Protecting accounts from credential stuffing is an asymmetrical challenge: attackers have access to billions of stolen credentials, while users and identity providers find it hard to determine which accounts require remediation.

In this report, we analyze protocols that help combat credential stuffing by triggering timely, actionable alerts: two protocols that query a centralized breach repository to check for exposed credentials, and one protocol that adopts a decentralized approach to prevent password reuse.

We provide a summary of the field in general, a critique of existing countermeasures, and potential directions for improvement.

1 INTRODUCTION

Research over the years has shown that users tend to reuse their passwords across multiple websites [1–6]. The scenario is further complicated when a single website suffers a data breach. An adversary can extract credentials from the leaked dataset from one site, and attempt to 'stuff' these credentials in other commonly used websites. A practice called 'credential stuffing', this attack continues to be heavily exploited even today.

A number of services gather reported data breaches and make these compromises available to be queried. Such services democratize access to leaked credentials, helping users avoid reusing previously breached passwords. For the remainder of the report, we refer to such services as *Compromised Credential Checking* (C3) services.

HaveIBeenPwned[1] (HIBP) provides one of the most accessed APIs for credential breach alerting. Deployed by Troy Hunt and using CloudFlare's CDN for efficient access, HIBP has been available for public access since 2018. Its API is used by Firefox for its breach alerting service, Firefox Monitor[2], and by password managers like 1Password[3]; services that are looking to protect users from reusing compromised credentials.

Google released their own C3 service, Google Password Checkup (GPC), in 2019 [7, 8]. Initially deployed as a Chrome extension[4], GPC is now available by default on Google accounts and for other services via an API[5]. GPC provides access to a database of around 4B breached usernames and passwords, gathered from multiple breaches [9].

C3 services are an effective solution for post-breach remedies. Since breaches are only known to websites after a considerable amount of time has passed, an additional useful countermeasure would be to encourage users to set unique passwords across websites, thereby reducing the effectiveness of credential stuffing attacks.

Intercepting users at account creation can be effective in getting users to set unique passwords [10]. These services are currently available through commercial password managers [11, 12]. Wang and Reiter suggested an alternative server-side protocol to perform similar evaluation on individual websites.

This report analyzes services deployed to discourage password reuse, at account creation and after data breaches. We critique existing implementations, evaluate solutions proposed by research in recent years, and finally summarize our recommendations for further improvement.

Primary Papers. The qualifier report requires that at least three papers guide the critique of the addressed field. We list these papers and their contributions below.

(1) Li et al. (CCS '19) [13] provided a formalization of C3 services, discussed vulnerabilities in HIBP and GPC, and provided countermeasures. This report includes their contributions in Section 3.
(2) Thomas et al. (Usenix Security '19) [8] introduced and proposed the GPC protocol. Their work also helped lay down the design principles and threat model for C3 services, both of which were used to guide our critique in Section 3
(3) Wang and Reiter (NDSS '19) [14] proposed a server-side protocol to check for password reuse and intercept users during account creation. We provide a critique of their proposal in Section 4.

2 BACKGROUND

This report assumes that the reader has prior knowledge of hash functions, salting, and public-key cryptography. We expand on a few additional concepts below.

Blinding. A technique using which a message (a hashed value in our case) can be encoded and concealed. The system allows for two parties to exchange encoded information without revealing the actual value of the concealed data.

The blinding technique discussed in Section 3.3 uses a randomized value, in conjunction with Elliptic Curve Cryptography[6]. It'll suffice for the reader to understand that unless randomized, hashed

[1] HaveIBeenPwned: https://haveibeenpwned.com/
[2] Firefox Monitor: https://monitor.firefox.com/
[3] 1Password: https://1password.com/
[4] Chrome extension: https://chrome.google.com/webstore/detail/password-checkup/pncabnpcffmalkkjpajodfhijclecjno
[5] GPC API: https://cloud.google.com/recaptcha-enterprise/docs/check-passwords
[6] ECC: https://avinetworks.com/glossary/elliptic-curve-cryptography/

values can be cracked using pre-computed dictionaries. A randomization technique like blinding in conjunction an ECC implementation, makes it nearly impossible to decipher the encoded value (under the Decisional Diffie Hellman assumption).

Bloom Filter. A bloom filter is a data structure used for efficient lookup, to determine if an element is already part of a set. It comprises a bit vector, where each cell represents a single indexed bit[7].

Adding elements to a bloom filter would require hashing the value, and computing the corresponding cells to be set. This way, any new value can be hashed and queried against the filter to determine if it potentially already exists.

Owing to possibilities of collisions, a bloom filter acts as a *probabilistic data structure*. It tells the user if an element potentially already exists in the set, but can be used to state if the element *definitely does not exist*.

Homomorphic Encryption. Certain public-key cryptography protocols, like ElGamal [15], allow mathematical operations to be carried out on ciphertexts without necessarily requiring these values to be decrypted beforehand.

Consider two plaintext messages, $m1$ and $m2$. If encrypted under a multiplicative homomorphic encryption scheme, $c_1 = \text{Enc}(m_1)$ and $c_2 = \text{Enc}(m_2)$, multiplication of the two ciphertexts would result in a third ciphertext $c_3 = c_1 \times c_2$, such that $\text{Dec}(c3) = m_1 m_2$.

Adversary Models. This report also addresses two types of adversaries in its threat model. A *malicious* adversary is assumed to arbitrarily deviate from the protocol, and alter control flow decisions. An *honest-but-curious* adversary addresses a weaker adversary that cannot deviate from the protocol but attempts to extract information that they should otherwise be unable to access [16].

3 POST-BREACH ALERTING

This section covers protocols adopted by C3 services, helping clients find credentials from past breaches. A number of popular services including Enzoic (previously, PasswordPing) [17], Breach Alarm [18], and Dehashed [19] provide an API to access information from data breaches.

In this section we focus on two types of C3 services: *password-only* C3 services, like HaveIBeenPwned and *username-password* C3 services, like Google's Password Security Checkup.

3.1 Overview

Generic C3 Protocol. The set $\mathcal{S}$ represents all credentials in the universe. Every C3 service has a server that collects breaches in a database $\tilde{\mathcal{S}}$. The database $\tilde{\mathcal{S}}$ has N entries of either passwords, $\{(w_1, w_2, \ldots w_N)\}$ or username-password pairs, $\{((u_1, w_1), (u_2, w_2), \ldots (u_N, w_N))\}$. We use w to denote passwords in order to avoid confusion with probability distributions discussed later.

A client, who has their own credential, $s = (u, w)$, $s \in \mathcal{S}$ wants to determine if s has been part of a prior breach, i.e., if $s \in \tilde{\mathcal{S}}$, CreateRequest($s$). The client in this case can be the user themselves or a service checking the credential on behalf of the user.

Figure 1 shows an abstract version of the C3 protocol. The client executes a computation over their credential s and sends a request

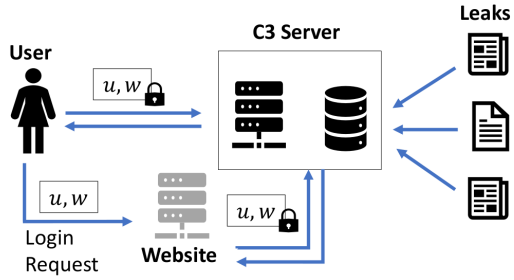

Figure 1: A C3 service allows a client (either the user or a service) to check if their credential has been part of a leak. The C3 server collects leaks from multiple breaches.

across to the C3 server. The client does not necessarily send s in plaintext, and instead an obfuscated version of it to the server.

Upon receiving the request, the C3 server performs a compuation on their database $\tilde{\mathcal{S}}$, and sends a response to the client, CreateResponse($\tilde{\mathcal{S}}$, CreateRequest(s)). In both protocols discussed in this report, the server responds with a set of obfuscated credentials for the client to check against. They do so to avoid having to compute a search of the entire database upon receipt of each request.

The client then computes a set intersection operation to determine if their credential s is part of the response set returned by the server.

Bucketization. When the client sends across a hash of their credential, the server responds with a set of hashed credentials back. The client then performs a *private set intersection* to determine if their credential is part of the returned set. Both C3 services described in this report use *Private Set Intersection* (PSI) as a method for the client to learn the presence of their credential without extracting any further information from the server.

C3 servers store a large database of credentials. HIBP has over over 500M stolen passwords, and GPC has over 4B stolen username-password pairs [8, 20]. These databases are constantly growing as new data breaches are reported. As databases grow, it becomes necessary to ensure that C3 services can effectively find credentials and respond to requests. Given effeciency and bandwidth constraints, servers avoid sending an obfuscated version of the entire dataset back in their response. They instead divide the credentials in their database $\tilde{S}$ into smaller sets, buckets.

Intuitively, clients can then request the server for a bucket based on a 'bucket identifier'. Clients can then perform a PSI operation between their credential and the returned bucket.

In such a scenario, each bucket b serves as an anonymity set, $b \in \beta(\mathcal{S})$, with K credentials. A large enough bucket gives the credential $s \in b$ plausible deniability of their membership, known as K-anonymity.

For the remainder of the section, consider $\mathcal{W}$ to be the set of all passwords in the universe, and p_w denotes the probability distribution of $\mathcal{W}$. We consider $\mathcal{U}$ to be the set of all usernames in the universe, and p to be the joint probability distribution over $\mathcal{U} \times \mathcal{W}$.

[7]Bloom Filter: https://avinetworks.com/glossary/elliptic-curve-cryptography/

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
$\mathcal{S}$	set of leaked credentials, $\mid \tilde{\mathcal{S}} \mid = N$
p	distribution of username-password pairs over $\mathcal{U} \times \mathcal{W}$
p_w	distribution of password over $\mathcal{W}$
$\hat{p}_s$	estimate of p_w used by the C3 server
q	query budget of an attacker
$\tilde{q}$	parameter to FSB, an estimated query budget for the attacker
β	function that maps a credential to a set of buckets
α	function that maps a bucket to the set of credentials it contains

Table 1: The notation used in Section 3.

Our analysis considers $\mathcal{S}$ to be the domain of credentials being checked, with $s \in \mathcal{S}$ denoting a single entry in $\mathcal{S}$. We consider $\mathcal{S} = \mathcal{W}$ for password-only C3 services and $\mathcal{S} = \mathcal{U} \times \mathcal{W}$ for username-password C3 services. $\tilde{S} \subseteq \mathcal{S}$ is a set of N leaked credentials.

We take f to be a cryptographic hash function $\{0,1\}^* \mapsto \{0,1\}^{l)}$ where l is a parameter of the system. The set of buckets, $\mathcal{B}$, is generated from the bucketization function $\beta : \mathcal{S} \mapsto \mathcal{P}(\mathcal{B}) \backslash \{\emptyset\}$ from the initial credential set. A credential can be assigned to multiple buckets and every credential mapped to at least one bucket.

We additionally consider an inverse function to $\beta : \alpha : \mathcal{B} \mapsto \mathcal{P}(\mathcal{S})$. The function α takes a bucket identifier as input and maps it to set of credentials; $\alpha(b) = \{s \in \mathcal{S} \mid b \in \beta(S)\}$. Since $\alpha(b)$ represents the reverse mapping of the set of all credentials, $\mathcal{S}$, the C3 servers in this report deal with a smaller function, $\tilde{\alpha}(b) = \alpha(b) \cap \tilde{\mathcal{S}}$.

Design Principles. Thomas et al.[8] laid out design principles for C3 services. While they laid these out keeping GPC in mind, we use them as guiding principles to provide critiques to either types of C3 services.

(1) **Democratized access:** A C3 service should provide access to any new party that wishes to query it. The protocol for lookup should not rely on trust or use authentication as a form of rate-limiting.
(2) **Actionable, not informational:** The advice that C3 services provide should be actionable, i.e., aimed at having users change their password on the account that has been compromised. Merely informing users about a potential breach, including only an email address, a phone number, or physical address is insufficient towards getting them to take action.
(3) **Breached, not weak:** The warnings displayed by C3 services should only deal with breached credentials, not just the result of weak passwords.
(4) **Near real time:** C3 services discussed in this report have been developed with the intent of interrupting users during account creation or login. Services need to be in a position to check entered passwords almost instantaneously, and C3 services designed to serve such purposes need to complete the protocol in real time.

Threat Model. The threat model that we consider in this section is combination of those discussed by Thomas et al. [8] and Li et al. [13]. For our purposes we are largely attempting to protect credentials against: (1) a malicious client and (2) an honest-but-curious C3 server.

(1) **Malicious Client.** A malicious client may have access to a dictionary of leaked credentials, $\mathcal{D} = \{(u_1, w_1), ...(u_N, w_N)\}$ or $\mathcal{D} = \{(w_1), ...(w_N)\}$. They may attempt to append their database by finding a new breached credential $s = (u, w)$ or $s = w$, such that $s \in \mathcal{S} - \mathcal{D}$ to their existing dictionary. A malicious server may, additionally, attempt to overwhelm the C3 server with numerous requests, resulting in a Denial-of-Service attack.
   - *Responses with bounded leakage.* Given a request for a credential s, the response from the server should limit the information leaked. For any two credentials $s_1 \in \tilde{\alpha}(b)$ and $s_2 \in \tilde{\alpha}(b)$, $\forall b \in \mathcal{B}$, the response from the server should be identical.

$$\mathsf{CreateResponse}(\tilde{\mathcal{S}}, \mathsf{CreateRequest}(s_1)) \approx \mathsf{CreateResponse}(\tilde{\mathcal{S}}, \mathsf{CreateRequest}(s_2))$$

     Simply stated, the response from the C3 server to credentials with identical linkage will be compuationally indistinguishable.
   - *Inefficient oracle.* The attacker should gain no advantage in learning the presence of a credential using the C3 protocol over checking the same against a plaintext copy of the breach. Taking t(f) as the time taken to compute the funciton f, and T being a time period such that:

$$\mathsf{t}(\mathsf{CreateRequest}(s)) > T$$

     An attacker with direct access to the breached dataset, determining membership of $\tilde{\mathcal{S}}$ can be given as:

$$\mathsf{t}(s \in \tilde{\mathcal{S}}) > T'$$

     Ideally, $T \geq T'$.
   - *Resistance to Denial-of-Service.* The C3 server should not require significantly more computation than the requesting client. Formally, it should be difficult for a malicious client to find a sequence of credentials $s_1, s_2, ...s_n$ such that

$$\sum_i \mathsf{t}(\mathsf{CreateRequest}(s_i)) \ll \sum_i \mathsf{t}(\mathsf{CreateResponse}(\mathsf{Req}_i))$$

(2) **Honest-but-curious Server.** The C3 server may be compromised or, alternatively, the server itself may attempt to learn the client's credentials from their requests.
   - *Requester Credential Anonymity.* A secure C3 protocol provides credential anonymity for every request. For every requested credential $s_1 \in b$, for a bucket $b \in \beta(\mathcal{S})$, such that, $\mid b \mid = K$. Then, $\forall s_2 \in b$

$$\mathsf{CreateRequest}(s_1) \approx \mathsf{CreateRequest}(s_2)$$

     No efficient adversary can distinguish either request apart any better than random guessing.
   - *Known Username Attack (KUA).* A weaker assumption than credential anonymity is when the adversary knows the client's username. In a KUA model, the attacker may have

been successful in linking usernames to queries based on metadata (e.g., originating IP address). The attacker may then narrow down potential passwords and perform login attempts on external websites. In such a scenario, Li et al. [13] modeled their protocols to be resistant against a variable, q, representing the number of attempts that the attacker can make, called the guessing budget.

3.2 Password-only C3 Services

HaveIBeenPwned (HIBP) is a password-only C3 service with over 500M passwords stored in its database. The service stores all breached passwords in its dataset in its SHA-1 hashed form. The entire dataset is available to download, and HIBP claims that offline attacks are out of its threat model since adversaries can access these breaches anyway[8].

HIBP uses a form of bucketization called Hash-prefix-based bucketization (HPB). The service creates a SHA-1 hash of each of the passwords in its database. The service assigns all credentials that share the same hash-prefix to the same bucket.

The number of buckets in the database $\tilde{\mathcal{S}}$ depend on l, the length of the hash prefix, while the number of credentials in each bucket depends on both, l and $\mid \tilde{\mathcal{S}} \mid$. HIBP uses $l = 20$ (5 hex characters) of the hash prefix, giving them the ability to partition $\tilde{\mathcal{S}}$ into 2^{20} buckets.

A client that wants to query the database computes a hash of their password, and only sends the 20-bit hash prefix to HIBP. The server sends a bucket of hashed passwords in response, using which, the client performs a set intersection to determine membership of their password. The number of hashed passwords in the bucket provides K-anonymity. In practice, HIBP has between 381 and 584 passwords in each bucket [21].

This is a weak form of K-anonymity because, (a) SHA-1 is a weak hash that can be cracked faster than commonly used hash functions [22], and (b) an adversary can use a dictionary to perform a brute-force attack and extract other passwords in the bucket offline.

HIBP justifies this choice by stating that leaked passwords are publicly available for adversaries to access and download. They trade privacy for simplicity and efficiency, by adopting a weaker hash and caching on Cloudflare's CDNs.

HIBP exposes a prefix of the user's password to the server, narrowing down the range of possible passwords to those available in one bucket. It also maps each credential to single bucket. Therefore, the set of passwords in each bucket can be ordered by their popularity, and these passwords cannot be found in a different bucket.

Li et al. [13] showed that in the case of HIBP, an attacker that learns the hash prefix and has a guessing budget of q is equivalent to an attacker with a guessing budget of q. $\mid \mathcal{B} \mid$ (and no knowledge of the hash prefix), where $\mid \mathcal{B} \mid$ is the total number of buckets.

For example, an attacker with $q = 10$ guesses, without knowledge of the hash prefix would guess the 10 most popular passwords. After learning the hash prefix, however, they can narrow their attempts to the 10 most popular passwords in the corresponding bucket. Since HIBP has a total of 2^{20} buckets, this attacker is now as capable as an attacker in the wild with a budget of $q = 10 \times 2^{20}$.

[8]HIBP Passwords: https://haveibeenpwned.com/Passwords

$\underline{\beta_{FSB}(w)}$:
$\gamma \leftarrow \min\left\{|\mathcal{B}|, \left|\frac{|\mathcal{B}|.\hat{p}_s(w)}{\hat{p}_s(w_{\tilde{q}})}\right|\right\}$
$s \leftarrow f(w)$
If $s + \gamma < |\mathcal{B}|$ then
  $r \leftarrow [s, s + \gamma - 1]$
Else
  $r \leftarrow [0, s + \gamma - 1 \bmod |\mathcal{B}|]$
  $r \leftarrow r \cup [s, |\mathcal{B}| - 1]$

$\underline{\tilde{\alpha}_{FSB}(b)}$:
/* returns $\{w \in \tilde{\mathcal{S}} | b \in \beta(w)\}$ */
$A \leftarrow \mathcal{W}_{\tilde{q}}$
For $w \in \tilde{\mathcal{S}} \backslash \mathcal{W}_{\tilde{q}}$ do
  If $b \in \beta_{FSB}(w)$ then
    $A \leftarrow A \cup \{w\}$
return A

Figure 2: Bucketization function β_{FSB} that assigns passwords to buckets, and Bucket retrieving function $\tilde{\alpha}$ that retrieves passwords assigned to a bucket in FSB. Here: $\hat{p}_s$ is the password distribution, $\mathcal{W}_{\tilde{q}}$ is the set of the $\tilde{q}$ most popular passwords, $\mathcal{B}$ is the set of buckets, and f is the hash function $f : W \rightarrow \mathbb{Z}_{|B|}$, and $\tilde{\mathcal{S}}$ is the set of password in the C3 server's database.

Frequency-Smoothing Bucketization. Li et al. [13] proposed a secure password-only alternative to HIBP's protocol. Frequency Smoothing Bucketization (FSB) assigns each password to multiple buckets based on its probability. They argue that assigning passwords to multiple buckets effectively reduces its conditional probabilities given a bucket identifier.

Implementing FSB would require an estimate of the distribution of human-chosen passwords, denoted in this section as $\hat{p}_s$. They also took into account $\tilde{q}$, an estimate of the query budget of the attacker.

Bucketization Function (β_{FSB}). FSB uses a hash function $f : \mathcal{W} \mapsto \mathbb{Z}_{|\mathcal{B}|}$. The bucketization function is given in Figure 2. The $\tilde{q}$ most popular passwords are added to all $\mid \mathcal{B} \mid$ buckets. Each of the remaining passwords are replicated across buckets proportional to their probability against the $\tilde{q}^{th}$ most popular password. Considering $w_{\tilde{q}}$ as the $\tilde{q}^{th}$ most popular password, a new password w (with probability $\hat{p}_s(w)$) is replicated $\gamma = [\frac{|\mathcal{B}|.\hat{p}_s(w)}{\hat{p}_s(w_{\tilde{q}})}]$ times. Each password is assigned to buckets in the range $[f(w), f(w) + \gamma - 1]$ mod $\mid \mathcal{B} \mid$.

Bucket Retrieving Function ($\tilde{\alpha}$). The FSB solution requires a large number of buckets, $\mid B \mid \approx N$. It may become infeasable to store each bucket separately. A possible solution is to then store buckets as continuous intervals using an interval tree data structure [23], a solution that still remains expensive to store. An interval tree with N entries requires $\mathcal{O}(N \log N)$ storage, $\mathcal{O}(N \log N)$ time to build the treee, and $\mathcal{O}(\log N + |\tilde{\alpha}(b)|)$ time to query a credential.

Estimating password distributions. The implementation of FSB requires that the server provide an approximate probability estimate to each password, and also provide clients a way of doing so before sending their request. Pal et al. [24] used a dataset of credentials in their password strength meter. Li et al. [13] used this dataset to, after trial and error, arrive at a probability distribution that combines a histogram, $\hat{p}_h$ (for popular passwords) and a normalized Markov-model based estimation, $\hat{p}_n$ (for less probable passwords) [25].

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

Security. Any implementation of FSB will require an estimate of the query budget, $\tilde{q}$, of an attacker that the protocol is protecting against. Li et al. [13] proved a theorem that stated that any adversary with a guessing budget lesser than the estimate, $q < \tilde{q}$, gains no additional advantage from learning the FSB bucket identifier. For any adversary with a larger guessing budget, $q > \tilde{q}$, their advantage is bounded by the difference between the estimate and the actual budget, $q - \tilde{q}$, and the probability $\hat{p}_s(w_{\tilde{q}})$ of the $\tilde{q}^{th}$ password.

Evaluation. Li et al. [13] used a breach dataset comprising 436M unique passwords previously used by Pal et al. [24]. They set up a C3 service with a rest API on AWS Lambda[9]. They used a DynamoDb[10] instance to hold each node of the FSB interval tree in a separate cell. Their evaluation showed that an estimate of the password distribution, $\hat{p}_s$ took 8.9MB of space on their dataset. With an estimated query budget of $\tilde{q} = 10^3$, the dataset has an approximate bucket size of 6K passwords (vs. 413 for HIBP) and takes about twice as much time as the default HIBP implementation to complete each password query (527ms vs. 220ms.).

Critique. Queries made to HIBP return hashed passwords, $H(p)$, which can be cracked offline using a pre-computed dicitonary - a threat that is exacerbated by the lack of salt. The scheme violates *requester credential anonymity* against an honest-but-curious server, and further reduce the search space for an attacker attempting to guess a user's queried password.

Since the response also provides access to hashes of other passwords in the bucket, both HIBP and FSB can be used to rebuild the underluing password dictionary, and do not satisfy the requirements for *bounded leakage*.

An FSB implementation requires that the C3 service estimate the probability distribution of passwords, $\hat{p}_s$ (≈8.9MB), and any new client acquire $\hat{p}_s$ before querying the server. Their implementation of the service does not provide *democratized access* to its clients and does not take into account the cost of updating their estimate of $\hat{p}_s$ with every user at regular intervals.

While FSB successfully limits the advantage that an attacker gains from learning a regular HPB bucket identifier, it is an expensive protocol to implement. Li et al. [13] observed an increase in bucket size proportional to the estimated query budget $\tilde{q}$. Their evaluation using the dataset from [24] rendered bucket sizes of 83 for $\tilde{q} = 1$, 852 for $\tilde{q} = 10$, 6299 for $\tilde{q} = 10^2$, and 25191 for $\tilde{q} = 10^3$. It helps to know that these query budgets are orders of magnitude smaller than the recommended query budget required to crack credentials for both online and offline attacks.

Tan et al. [26] recommend protecting credentials against at least $q = 10^8$ for online attacks. Taking this bare minimum as an estimate, $\tilde{q}$ would result in buckets of $> 10^9$ passwords per bucket passed in each query from an FSB-based C3 service. Such bandwidth requirements will be infeasable for a *near real time* implementation.

Lastly, password-only C3 services are effective in merely informing users about weak passwords. If two users, u_1 and u_2 share a password that's been part of a breach, it limits the security advice that the user receives. We therefore require a C3 service that takes a username-password pair as input, and provides an *actionable, not informational* alert to the user.

[9]AWS Lambda: https://aws.amazon.com/lambda/

[10]DynamoDb: https://aws.amazon.com/dynamodb/.

3.3 Username-password C3 Services

Google Password Checkup (GPC) is a username-password C3 service with a breach database of over 4B credentials. The service was released as a means to provide context-based, actionable alerts, initially made available as a browser extension, and is now included as a default service on all Google accounts.

Similar to HIBP, GPC also bucketizes its database using their hash-prefix. The service, however, creates its hashes using a more secure alternative, Argon2[11].

GPC secures queried credentials using K-anonymity guaranteed by bucketization, performs a private set intersection between the queried credential and its corresponding bucket, and relies on a computationally expensive hashing function.

The hash-based prefix in question, however, is the first 4-hex characters (16-bits) of the resulting hash. GPC leaks fewer bits than HIBP does, but also includes larger buckets of passwords in its response.

GPC prevents pre-computed dictionary attacks on its 16-byte hash output by blinding it with a 224-bit secret key b. It maps the hash to a point on the ellipic curve NID_secp224r1 and raises the resulting point by the power of b. The resulting blinded hash of the password is then stored in the corresponding bucket.

CreateRequest: The client generates a request by first hashing the username-password pair, and then blinding it using the same method adopted by the server. It, however, uses its own random secret key, a, to blind the hash. It then sends the 2-byte prefix of the hash along with the blinded credential to the server.

CreateResponse: Upon receiving the request, the server fetches the bucket corresponding to the hash-prefix. It also manipulates the blinded value in the request, to help the client execute a PSI with the returned bucket. The protocol is laid out in Figure 3.

Under the random oracle model [27], considering Argon2 to be modeled as a perfect hashing function, the hash-and-blind scheme adopted by GPC is an implementation of an Oblivious Pseudo-random Function (OPRF) against an honest-but-curious adversary under the decisional Diffie-Hellman assumption.

Security. Despite adopting multiple new security measures over HIBP's approach, GPC suffers from similar security vulnerabilities resulting from HPB. Li et al. [13] observed that with a hash prefix length of 16 bits, an adversary under the KUA model could successfully guess 59.7% of the passwords from previously uncompromised accounts in fewer than 1K guesses.

Identifier-based Bucketization.(IDB) Both Li et al. [13] and Thomas et al. [8] independently proposed a more secure albeit computationally expensive variant to the original Hash Prefix-based GPC protocol. They proposed using merely the hash prefix of the account username (identifier), $f(u)_{[0:n]}$, instead of both, the username and password, $f(u,w)_{[0:n]}$ as the bucket identifier. The credentials stored in the bucket can still be a hash-and-blind of the (u,w) pair combined. Changes to the original protocol are included in the boxed code in Figure 3.

[11]GPC uses the following parameters for Argon2 hashing: single thread, 256 MB or memory and time cost of three.

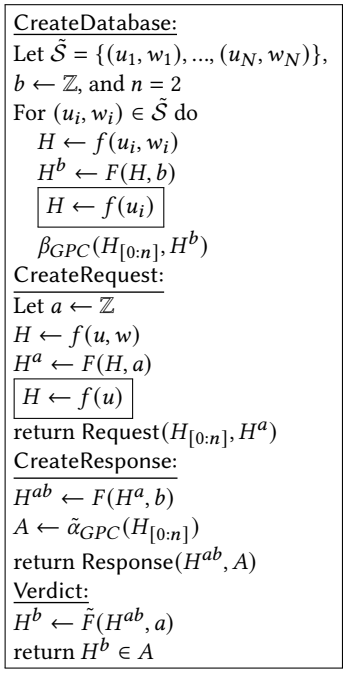

Figure 3: Algorithms for GPC. Here $\tilde{S}$ is the set of leaked credentials with the C3 server, f is the hash function $f : W \rightarrow \mathbb{Z}_{|B|}$, F is a blinding function while $\tilde{F}$ is an inverse function used to unblind values, a and b are secrets used by the client and server respectively. Operations in boxes are additional steps included in the IDB variant.

As a compromise, clients will have to compute an additional computationally expensive hash, but will, however, not have to expose any of their password bits while engaging with the protocol.

Critque. GPC addresses and resolves multiple issues that were observed with HIBP. They blind their hashed credentials, making it difficult for an adversary to infer any new information on the basis of the bucket returned from the server (*bounded leakage*).

The implementation of both GPC and its IDB variant involves an inefficient oracle. The client engaging in the protocol performs multiple hashes, blinds and unblinds values, and performs a PSI operation. The server, on the other hand, only performs a blinding operation[12] for each request before fetching the corresponding bucket. Their solution also makes it difficult for an adversary to launch a *Denial-of-service* attack on the C3 server.

GPC provides greater *requester credential anonymity* to its clients in comparison to HIBP, but is still vulnerable as was shown by Li et al. [13]. Their claim to have satisfied this threat model does not hold in the light of new information. With the IDB variation, however, GPC can potentially at least satisfy the KUA model, compromising bits of the username while sharing no information about the password.

Of the design principles, GPC successfully achieves: (a) *actionable, not informational* alerts with usernames included for context; (b) *breached, not weak* passwords, reducing warning fatigue [28]; (c) *near real time* execution of the protocol as is evident from a median of 8.5s observed by the deployment from Thomas et al. [8].

GPC does not, however, provide democratized access as they claim in their paper. The current implementation provides an API to users who register with Google's Cloud Services, a paid service that requires authentication and billing information.

To their credit, Thomas et al. [8] independently proposed the IDB variant before Li et al. [13] did in the same year. While they recognized the security advantage of the implementation, their API documentation does not currently include instructions for this variant, indicating that relevant code has not been included in their deployment.

The current GPC implementation, is however, a secure method of querying C3 services, especially advantageous to users with a Google account who use Chrome's inbuilt password manager.

4 PRE-BREACH ALERTING

C3 services help inform users after their credentials have been compromised in a data breach. While this is an important service to provide, we need additional steps to limit the effectiveness of credential stuffing by preventing users from reusing passwords in the first place. Compromise of password databases often go undetected for long periods of time (as of 2017, 15 months on average), leaving C3 servers out of sync [29].

Services designed to restrict users from reusing passwords can be effective in increasing the possibility that they take action before a breach. Towards this end, this section provides an insight into efforts to combat password reuse.

4.1 Server-side Reuse Checks

Wang and Reiter [14] proposed a server-side protocol to check for reused passwords during account creation. They did so *"to question the zeitgeist in the computer security community that password reuse cannot be addressed by technical means without imposing unduly on user security or privacy."*

They proposed that websites register with a centralized directory that maintains a list of services that a user has an account with. Any new service that the user wishes to register with will then contact all other websites via the directory, to ensure that the new password that the user is trying to set hasn't been previously used.

Their paper claims that the proposed framework does not require unversal adoption. The top 20 websites on the Internet have >3.5B user accounts[13]. If only these top 20 sites were to participate in the proposed protocol, users would then be forced to remember at least 4-5 passwords which is *"the maximum for unrelated, unregulated passwords that users can be expected to cope with"* [30].

Security and Privacy Goals. The proposal took two primary goals into consideration.

(1) **account location privacy**: Querying websites do not learn the presence of user accounts on other participating websites.
(2) **account security**: While improving the security of passwords of a user accounts created on a given website, the

[12] A blinding operation is less expensive to compute in comparison to a hash function.

[13] Statistics from: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/

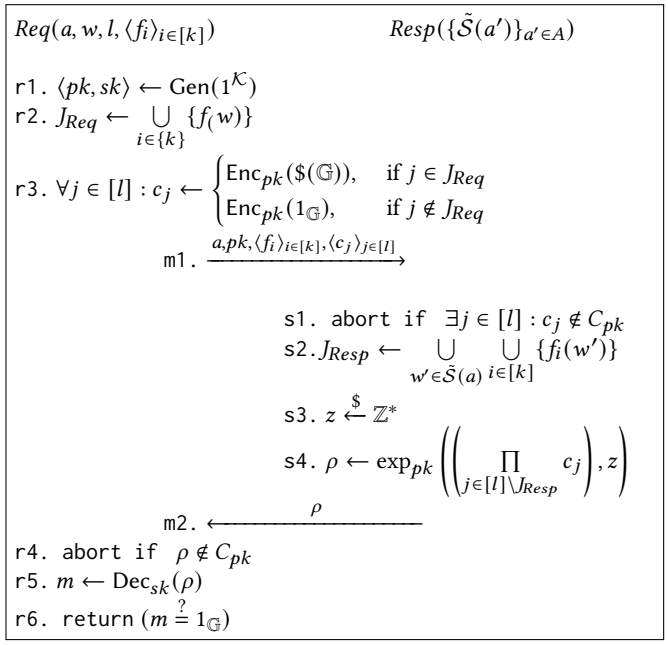

Figure 4: PMT protocol. *Req* **returns *true* if the password w is similar to a password with registered with** *Resp*, **i.e.,** $w \in \tilde{\mathcal{S}}(a)$. **Here,** f **is a hash function,** $f : \mathcal{W} \to \mathbb{Z}$, l **is the length of the bloom filter,** J_{Req} **and** J_{Resp} **are sets of occupied indices of the bloom filter.**

protocol does not qualitatively degrade security in a different aspect.

A requesting site with the user's password performs a *private set-membership-test* (PMT) protocol with every other website that the user has an account with. We lay out the details of the protocol below.

Private Set-Membership-Test. (PMT) Participants in the protocol include a requester, *Req*, who inquires with a responder, *Resp*, as to whether a password, w, chosen by a user with account identifier, a, at *Req* is already in use with *Resp*.The responder has a set of passwords, $\tilde{\mathcal{S}}(a)$, similar to the passwords that a has set with *Resp*.

The PMT protocol uses a multiplicatively homomorphic encryption scheme, Elgamal, $\epsilon = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \times_{[.]})$. The protocol uses $z \xleftarrow{\$} \mathbb{Z}$ to denote random selection from set $\mathbb{Z}$.

- Gen is a randomized encryption algorithm with a public-key/private-key pair, $\langle pk, sk\rangle \leftarrow \mathsf{Gen}(1^{\mathcal{K}})$ from a multiplicative cyclic group $\langle \mathbb{G}, \times_{\mathbb{G}}\rangle$.
- Enc is a randomized encryption algorithm that using public-key pk and plaintext message $m \in \mathbb{G}$. $C_{pk}(m)$ denotes the set of all ciphertexts generated by $\mathsf{Enc}_{pk}(m)$. $C_{pk} = \bigcup_{m\in\mathbb{G}} C_{pk}(m)$ is the ciphertext space under the public key pk.
- Dec is a deterministic algorithm that uses private-key sk and ciphertext $c \in C_{pk}(m)$, and produces $m \leftarrow \mathsf{Dec}_{sk}(c)$. If $c \notin C_{pk}$, then $\perp \leftarrow \mathsf{Dec}_{sk}(c)$.
- $\times_{[.]}$ is a randomized multipliative algorithm that on receiving two ciphertexts $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$ and public-key pk, produces $c \leftarrow c_1 \times_{pk} c_2$ where $c \in C_{pk}(m_1 m_2)$.

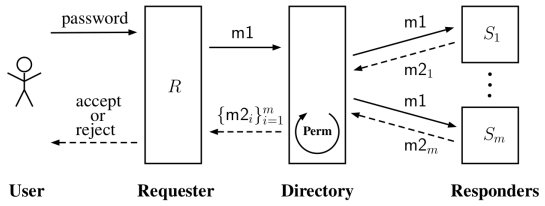

Figure 5: A password reuse detection framework based on the PMT protocol.

The protocol is shown in Figure 4. The requester takes as input the account identifier, a, and the new password w. The requester *Req* creates a bloom filter of length l, computes k hash functions f, and records the set of corresponding indices, J_{Req}. The *Req* then encrypts the entries in the bloom filter with a random value for all indices $j \in J_{Req}$, and the identity of the cyclic group for all other values.

The responder *Resp* computes own hashes on its set of passwords $P(a)$, and records the indices corresponding to the entries in the bloom filter, J_{Resp}. It then returns a value to *Req* indicating $[l]\setminus J_{Resp} \subseteq [l]\setminus J_{Req}$, where "\" indicates set difference. *Resp* performs a multiplicative operation on the ciphertext of indices from $[l]\setminus J_{Resp}$. If the resulting product holds the identity $1_{\mathbb{G}}$, the password w has a similar value in $P(a)$.

Malicious requester. The paper suggests that if the responder correctly performs the protocol, a malicious requester learns nothing more than the existence of a password similar to the one queried.

A malicious requester can possibly generate multiple requests and learn which passwords are still in use. The paper addresses recommends that the directory trigger an email to the user to confirm each request, thereby reducing the scale of the attack. It'll be useful for the reader to know that email is considered an insecure and exploitable mode of authentication [31, 32]. Besides, given that users already verify access to their account after the creating a password, notification fatigue can make the suggested remedy ineffective [28].

Finally, a malicious requester may be able to infer the identity of websites that the user has an account with, thereby using the protocol as a fingerprinting vector. The paper suggests that the directory randomize the order of responses, making it hard to infer the websites that have been queried. While useful in the context of a larger list of websites, the paper acknowledged the difficulty of getting numerous websites to participate in the protocol. If only the 20 most popular websites participate, a malicious requester can effectively narrow down vulnerable sites, reducing their online query budget, $\tilde{q}$, and weakening *account location privacy*.

Malicious Directory. The paper suggests that the directory, being a third-party entity can either be trusted or untrusted. If the third-party directory is considered a trusted entity, websites register regular public URLs with it. Websites can, however, register pseudo addresses for each entry to an untrusted directory, and direct all incoming traffic through Tor. Besides the narrowing down of possible values to $\Pr[1/20]$ for a service with 20 participating

websites, hidden addresses can be deanonymized by an untrusted directory [33].

Since directories have access to the implementation of the consent mechanism, they can bypass such requirements and behave in a manner similar to a *malicious requester*, a threat vector that the paper did not address.

Malicious Responder. A malicious responder may attempt a denial-of-service attack by always returning $\rho \in C_{pk}(1_{\mathbb{G}})$ for any request. Such a response would make it impossible for the user to create an account with the requester.

The paper suggests that the directory randomly generate test password queries, filling the bloom filter J_{Req} with $c_j \in C_{pk} \setminus C_{pk}(1_{\mathbb{G}})$. If the responder returns an incorrect $\rho \in C_{pk}(1_{\mathbb{G}})$, the directory can block it from receiving future queries. Performing audits is an effective solution, but would require creating a bypass of user consent while generating requests, thereby creating a backdoor to launch scalable attacks.

Security for password storage. The proposed PMT protocol will require each responder to generate and store $\tilde{\mathcal{S}}(a)$ for each account - a function that would require access to the user's passwords in plaintext for each request.

An alternative solution suggested that each responder preemptively compute the hashes of passwords for $\tilde{\mathcal{S}}(a)$ and discard these passwords. The hashing function used would need to be different from the one used to store passwords for user identification (usually PBKDF2), and would need to be computationally expensive to make it difficult to crack in the event of a compromise. This solution did not, however, address that hashes require randomization to be secure against offline dictionary attacks.

The paper finally suggests that responders salt these hashes, and all responders share the same salt. Such synchronization adds communication overhead, and reduces the effectiveness of randomization in the event of a data breach at any participating website - factors that were not addressed.

Critique. Following the critiques already included in the discussion of protocol security above, we provide a higher-level overview of our concerns with the proposed protocol, its practicality, efficiency, and need in this subsection.

Wang and Reiter attempted *"to question the zeitgist"* - an attempt that should be commended and encouraged. Such papers have, in the past, managed to generate discussion and have offered fresh perspective [34].

The problem addressed in the paper, however, is the absence of server-side protocols to combat password reuse. The proposed PMT protocol introduces new exploits of user account passwords, and is expensive to set up and maintain.

The PMT protocol slows down the process of account creation, taking an approximate of 5s with a trusted directory (8s for untrusted) for a single query, not taking into account additional user consent requirements.

The deployment of the system would require multiple services agreeing to collaborate in the creation and maintenance of an expensive centralized directory. Even if the top 20 services agree to participate in such a protocol, a practical deployment would require replicating the directory and managing efficiency, security, and synchronization. Practical frameworks for Byzantine fault-tolerance across replicas exist [35, 36], and will be required in any deployment of the PMT service, but cost of such overhead needs evaluation.

We finally argue against the requirement of a server-side check for password reuse. The paper repeatedly refers to the benefits of using a password manager to generate strong, randomized passwords, but does not address additional services that these password managers provide. In the section that follows we address existing alternatives and expand on their benefits.

4.2 Client-side Reuse Checks

Applications, extensions, and built-in password managers provide client-side checks for password reuse across websites. 1Password's Watchtower [11], LastPass security checks [12], Firefox's Lockwise Monitor [37], and Chrome/Google's built-in password checks [38] are examples of services that evaluate all user passwords registered with them.

Password manager implementations have user passwords decrypted locally on the user's device or browser, using keys derived from a master password. The mechanism restricts servers from accessing password vaults without user consent.

With the plaintext and hashed versions of passwords available locally, applications can perform simple equality checks across all registered passwords, and return a warning to users without creating network requests to other websites. These services are not vulnerable to attacks on the PMT protocol described in the previous section.

Since these services combine reuse checks with strength meters and requests to C3 services, they generate warnings and alerts from a single source. Golla et al. [39] recommended an evaluation of the effectiveness of such centralized notifications (vs. those generated by individual websites), and MacGregor [40] performed a user study, providing evidence of comprehension of password reuse in a cross-site context.

The effectiveness of client-side reuse checks boils down to adoption by users and the assumption that users actively register all account credentials with their password manager. Adoption of password managers, however, remains a concern, with users demonstrating trouble in understanding their working and the additional security benefits that they provide [41, 42].

5 DISCUSSION

Despite multiple services deploying solutions to warn users against password reuse, the practice fails to demostrate a tangible impact on user actions. Individual websites deploying their own notification mechanism are ineffective in translating warnings into action, and add to notification fatigue [28, 39].

Both HIBP and GPC provide access to a large breach dataset and receive queries from multiple websites, password managers, and end users. These services, however, lack democratized access, and are yet to adopt recommendations made by security researchers (including their own [8]).

Both, HIBP and GPC provide access to different leak datasets, gathered from multiple sources [9]. Applications usually query a single C3 server, even though all C3 services do not synchronize updates to their database. Querying even a single C3 service is, however, more useful than querying none.

Since C3 services are only useful after a breach has been identified by the responsibe party, strengthening passwords at the outset becomes imperative. Services need to aid users to register strong, unique passwords by providing random generators [38], strength meters [26], and reuse checks [11, 12].

Server-side checks to password reuse would require consensus and an expensive deployment, which, even if implemented would create new vulnerabilities, and slow down the creation of accounts on independent websites.

Attempts at centralizing password-based alerting services, however, need to rely on client-side solutions to ensure safety and security. Client-side implementations require that the general public adopt and trust applications that access and store user passwords, and query C3 services on their behalf.

Researchers in recent years show consensus on a few fundamental recommendations: (a) choose a strong, random, and unique password for each website, (b) opt for two-factor authentication wherever possible, (c) use a password manager. A password manager can help resolve the former two, but researchers need to actively engage with commercial deployments to make the adoption of such services a de facto aid to authentication on the Internet.

REFERENCES

[1] A. S. Brown, E. Bracken, S. Zoccoli, and K. Douglas, "Generating and remembering passwords," *Applied Cognitive Psychology*, vol. 18, no. 6, pp. 641–651, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/acp.1014

[2] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse." in *Network and Distributed Systems Security (NDSS) Symposium*, 2014, pp. 1–15.

[3] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: User attitudes and behaviors," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, ser. SOUPS '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 1–20. [Online]. Available: https://doi.org/10.1145/1837110.1837113

[4] I. Ion, R. Reeder, and S. Consolvo, ""...no one can hack my mind": Comparing expert and non-expert security practices," in *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*. Ottawa: USENIX Association, Jul. 2015, pp. 327–346. [Online]. Available: https://www.usenix.org/conference/soups2015/proceedings/presentation/ion

[5] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 295–310. [Online]. Available: https://doi.org/10.1145/3133956.3133973

[6] C. Wang, S. T. Jan, H. Hu, D. Bossart, and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 196–203. [Online]. Available: https://doi.org/10.1145/3176258.3176332

[7] J. Pullman, K. Thomas, and E. Bursztein, "Protect your accounts from data breaches with Password Checkup." [Online]. Available: https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html

[8] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, "Protecting accounts from credential stuffing with password breach alerting," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1556–1571. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/thomas

[9] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein, "Data breaches, phishing, or malware? understanding the risks of stolen credentials," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1421–1434. [Online]. Available: https://doi.org/10.1145/3133956.3134067

[10] S. G. Lyastani, M. Schilling, S. Fahl, M. Backes, and S. Bugiel, "Better managed than memorized? studying the impact of managers on password strength and reuse," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 203–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lyastani

[11] "About Watchtower privacy in 1Password." [Online]. Available: https://support.1password.com/watchtower-privacy/

[12] "How Secure is My Password | LastPass Password Checker." [Online]. Available: https://www.lastpass.com/how-secure-is-my-password

[13] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for checking compromised credentials," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1387–1403. [Online]. Available: https://doi.org/10.1145/3319535.3354229

[14] K. C. Wang and M. K. Reiter, "How to end password reuse on the web," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019, pp. 1–15. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_06A-5_Wang_paper.pdf

[15] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[16] W. Du and M. J. Atallah, "Secure multi-party computation problems and their applications: A review and open problems," in *Proceedings of the 2001 Workshop on New Security Paradigms*, ser. NSPW '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 13–22. [Online]. Available: https://doi.org/10.1145/508171.508174

[17] "Detect Compromised Credentials and Prevent Account Takeover." [Online]. Available: https://www.enzoic.com

[18] "How Safe Is Your Password?" [Online]. Available: https://breachalarm.com/

[19] "Find out if a password hack has exposed your password online." [Online]. Available: https://dehashed.com/

[20] J. Casal, "1.4 Billion Clear Text Credentials Discovered in a Single Database," Dec. 2020. [Online]. Available: https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14

[21] J. Ali, "Validating Leaked Passwords with k-Anonymity," Feb. 2018. [Online]. Available: https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/

[22] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 570–596.

[23] Wikipedia contributors, "Interval tree — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Interval_tree&oldid=989941483, 2020, [Online; accessed 27-January-2021].

[24] B. Pal, T. Daniel, R. Chatterjee, and T. Ristenpart, "Beyond credential stuffing: Password similarity models using neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 417–434.

[25] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 689–704.

[26] J. Tan, L. Bauer, N. Christin, and L. F. Cranor, "Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1407–1426. [Online]. Available: https://doi.org/10.1145/3372297.3417882

[27] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 62–73. [Online]. Available: https://doi.org/10.1145/168588.168596

[28] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 257–272. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/akhawe

[29] "2018 Credential Spill Report | Shape Security." [Online]. Available: https://www.shapesecurity.com/reports/2018-credential-spill-report

[30] A. Adams and M. A. Sasse, "Users are not the enemy," *Commun. ACM*, vol. 42, no. 12, p. 40–46, Dec. 1999. [Online]. Available: https://doi.org/10.1145/322796.322806

[31] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan, "The password reset mitm attack," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 251–267.

[32] J. Chen, V. Paxson, and J. Jiang, "Composition kills: A case study of email sender authentication," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2183–2199. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-jianjun

[33] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas, "Circuit fingerprinting attacks: Passive deanonymization of tor hidden services," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 287–302. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kwon

[34] S. Savage, "Lawful device access without mass surveillance risk: A technical design discussion," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1761–1774. [Online]. Available: https://doi.org/10.1145/3243734.3243758

[35] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362.

[36] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 277–290. [Online]. Available: https://doi.org/10.1145/1629575.1629602

[37] "New password security features come to Firefox with Lockwise." [Online]. Available: https://blog.mozilla.org/firefox/password-security-features

[38] "Google Password Manager." [Online]. Available: https://passwords.google.com

[39] M. Golla, M. Wei, J. Hainline, L. Filipe, M. Dürmuth, E. Redmiles, and B. Ur, ""what was that site doing with my facebook password?": Designing password-reuse notifications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1549–1566. [Online]. Available: https://doi.org/10.1145/3243734.3243767

[40] R. MacGregor, "User Comprehension of Password Reuse Risks and Mitigations in Password Managers," Ph.D. dissertation, Dalhousie University, Apr. 2020. [Online]. Available: http://hdl.handle.net/10222/78416

[41] S. Pearman, S. A. Zhang, L. Bauer, and N. Christin, "Why people (don't) use password managers effectively," in *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 319–333. [Online]. Available: https://www.usenix.org/conference/soups2019/presentation/pearman

[42] H. Ray, F. Wolf, R. Kuber, and A. J. Aviv, "Why older adults (don't) use password managers," in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ray