usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Protecting accounts from credential stuffing with password breach alerting

Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, and Sarvar Patel, *Google;* Dan Boneh, *Stanford;* Elie Bursztein, *Google*

https://www.usenix.org/conference/usenixsecurity19/presentation/thomas

This paper is included in the Proceedings of the
28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Protecting accounts from credential stuffing with password breach alerting

Kurt Thomas[*] *Jennifer Pullman*[*] *Kevin Yeo*[*] *Ananth Raghunathan*[*]
Patrick Gage Kelley[*] *Luca Invernizzi*[*] *Borbala Benko*[*] *Tadek Pietraszek*[*]
Sarvar Patel[*] *Dan Boneh*[◇] *Elie Bursztein*[*]

Google[*] *Stanford*[◇]

Abstract

Protecting accounts from credential stuffing attacks remains burdensome due to an asymmetry of knowledge: attackers have wide-scale access to billions of stolen usernames and passwords, while users and identity providers remain in the dark as to which accounts require remediation. In this paper, we propose a privacy-preserving protocol whereby a client can query a centralized breach repository to determine whether a specific username and password combination is publicly exposed, but without revealing the information queried. Here, a client can be an end user, a password manager, or an identity provider. To demonstrate the feasibility of our protocol, we implement a cloud service that mediates access to over 4 billion credentials found in breaches and a Chrome extension serving as an initial client. Based on anonymous telemetry from nearly 670,000 users and 21 million logins, we find that 1.5% of logins on the web involve breached credentials. By alerting users to this breach status, 26% of our warnings result in users migrating to a new password, at least as strong as the original. Our study illustrates how secure, democratized access to password breach alerting can help mitigate one dimension of account hijacking.

1 Introduction

The wide-spread availability of usernames and passwords exposed by data breaches has trivialized criminal access to billions of accounts. In the last two years alone, breach compilations like Antipublic (450 million credentials), Exploit.in (600 million credentials), and Collection 1-5 (2.2 billion credentials) have steadily grown as their creators aggregated material shared on underground forums [21,25]. Despite the public nature of this data, it remains no less potent. Previous studies have shown that 6.9% of breached credentials remain valid due to reuse, even multiple years after their initial exposure [51]. Absent defense in depth techniques that expand authentication to include a user's location and device details [12,17], hijackers need only conduct a credential stuffing attack—attempting to log in with every breached credential—to isolate vulnerable accounts.

While users (or identity providers) can mitigate this hijacking risk by resetting an account's password, in practice, discovering which accounts require attention remains a critical barrier. This has given rise to breach alerting services like HaveIBeenPwned and PasswordPing that actively source breached credentials to notify affected users [26, 43]. At present, these services make a variety of tradeoffs spanning user privacy, accuracy, and the risks involved with sharing ostensibly private account details through unauthenticated public channels. One consequence of these tradeoffs is that users may receive inaccurate remediation advice due to false positives. For example, both Firefox and LastPass check the breach status of usernames to encourage password resetting [13,42], but they lack context for whether the user's password was actually exposed for a specific site or whether it was previously reset. Equally problematic, other schemes implicitly trust breach alerting services to properly handle plaintext usernames and passwords provided as part of a lookup. This makes breach alerting services a liability in the event they become compromised (or turn out to be adversarial).

In this paper, we present the design, implementation, and deployment of a new privacy-preserving protocol that allows a client to learn whether their username and password appears in a breach without revealing the information queried. Our protocol offers two main advantages compared to existing schemes. First, our design takes into account the threat of both an adversarial client (e.g., an attacker attempting to steal usernames and passwords from our service) and an adversarial server (e.g., an attacker harvesting usernames and passwords sent to the service). We address these risks using a combination of computationally expensive hashing, k-anonymity, and private set intersection. Second, these privacy requirements allow us to check a client's exact username and password against a database of breached credentials (versus only usernames, or only passwords currently), thus reducing false positives that lead to warning fatigue.

To demonstrate the feasibility of our protocol, we publicly released a Chrome extension that warns users when they log in to a website using one of over 4 billion breached usernames and passwords. While in theory any identity provider or password manager can integrate with our protocol, we opted for in-browser alerting first as it scales to the long tail of domains. Nearly 670,000 users from around the world installed our extension over a period of February 5–March 4, 2019. During this measurement window, we detected that 1.5% of over 21 million logins were vulnerable due to relying on a breached credential—or one warning for every two users. By alerting users to this breach status, 26% of our warnings resulted in users migrating to a new password. Of these new passwords, 94% were at least as strong as the original.

Anonymous telemetry reported by our extension reveals that users reused breached credentials on over 746,000 distinct domains. The risk of hijacking was highest for video streaming and adult sites, where 3.6–6.3% of logins relied on breached credentials. Conversely, users appeared to internalize password security advice (or were forced to do so via password composition policies) specifically for financial and government sites, where only 0.2–0.3% of logins involved breached credentials. Despite variations across industries, our analysis reveals that the threat of credential stuffing extends well into the long tail of the Internet. Absent new forms of authentication, we believe that it is critical to democratize access to breach alerting so that both users and identity providers can proactively resecure their accounts.

In summary, we frame our key contributions as follows:

- We develop and publicly release a new protocol for detecting whether a username and password pair appears in a data breach without revealing the information queried. Our protocol improves on the privacy of existing schemes while also reducing the risk of false positives.
- We outline the technical challenges of deploying this scheme in practice, including the computational overhead, latency, and cost required to mediate access to over 4 billion breached usernames and passwords.
- Based on a real-world deployment, we find that 1.5% of logins across the web involve breached credentials. We caution this is a lower bound as logins are not unique. Roughly one in two of our 670,000 users received a warning.
- Users responded to 26% of our warnings by resetting their password; 94% of new passwords were as strong or stronger than the original passwords.

2 Background and requirements

To start, we establish the design principles and threat model that underpin our breach alerting protocol. We compare these requirements against existing solutions from HaveIBeenPwned and PasswordPing—as well as related cryptographic protocols like private information retrieval and oblivious transfer—to highlight the tradeoffs that all of these approaches make in terms of privacy, overhead, accuracy, and trust.

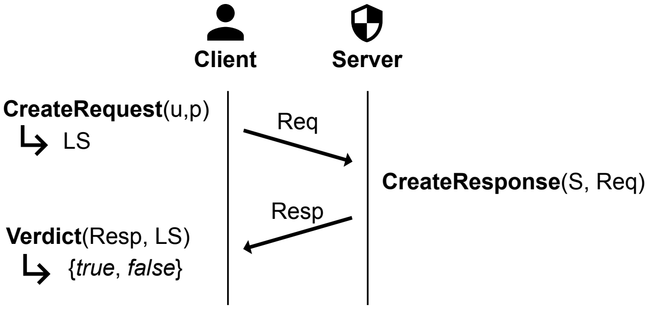


Figure 1: Abstract protocol for a breach alerting service. At a high level, a client generates a request based on some computation over a username and password. The server then returns a response that allows the client to arrive at a verdict for whether their credential is in a breach.

2.1 Abstract protocol

We provide an abstract protocol for our breach alerting service in Figure 1. We reuse these function names and terminology throughout our work. Here, a client with access to a username and password tuple (u,p) executes some computation via $\mathsf{CreateRequest}(\mathsf{u},\mathsf{p})$ that produces a local state LS and request Req that it sends to the breach alerting service. This service stores and regularly updates a database of unsafe credentials $\mathsf{S} = \{(\mathsf{u}_1,\mathsf{p}_1),\ldots,(\mathsf{u}_n,\mathsf{p}_n)\}$. Upon receiving a request, the server accesses its credential store S, runs $\mathsf{CreateResponse}(\mathsf{S},\mathsf{Req})$, and sends the resulting response Resp to the client. Finally, the client arrives at a verdict whether the credential queried was exposed through a breach by calculating $\mathsf{Verdict}(\mathsf{Resp},\mathsf{LS})$. Because new breaches emerge over time, a client should regularly repeat this process as prior verdicts may no longer be valid.

2.2 Design principles

Democratized access: At present, identity providers individually collect breached password data to reset their affected user accounts [4, 58]. This fails to scale to all identity providers, resulting in patchy protection across services and incidents. Any breach alerting service should be accessible to all end users and identity providers, and as such, not require trust between the parties involved. This means we cannot rely on authenticated accounts as a form of rate limiting. We define trust more formally in our threat model in Section 2.3.

Actionable, not informational: Any breach alerting service should provide users with accurate and actionable security advice such as re-securing an account via a password reset. An alert that warns a client about the mere presence of exposed data such as a client's email address, phone number, or physical address lacks a straightforward recovery step and is thus out of scope for our design. Similarly, an alert merely warning a client that password material was exposed (rather than the specific password involved) may lead to false positives.

Breached, not weak: Alerting should only trigger when all the information necessary to access an account (e.g., a username and password) is exposed. While cracking dictionaries (often composed from breached passwords) may include a client's weak, guessable password, any subsequent attack potentially requires multiple guesses and thus represents a smaller threat than full credential exposure. We assume that most online services employ sufficient throttling to make such bruteforcing impractical. Conversely, attacks against exact username and password pairs are actively deployed in the wild. Indeed, Thomas et al. showed that users with non-stale credentials exposed by third-party breaches were ten times more likely to become hijacked than a random user [51]. Our emphasis on breached credentials helps us prioritize scarce user attention [5] and avoid potential warning fatigue similar to other warning models [2]. While migrating users to stronger passwords in general remains an important task, it is out of scope for our design.

Near real-time: The time that elapses between a client querying a credential and learning its breach status should be near real-time in order to facilitate integration directly with account security flows, password managers, or upon password entry. This potentially constrains the level of privacy protections provided by any protocol due to computational overhead and network latency of any cryptographic primitives involved.

2.3 Threat model

Democratized access hinges on mutual distrust between a client and the server involved in our breach alerting protocol. We develop our threat model with both an adversarial client and adversarial server in mind. In the case of an adversarial client with access to their own breach dataset $\mathsf{D} = \{(\mathsf{u}_1, \mathsf{p}_1), \ldots, (\mathsf{u}_n, \mathsf{p}_n)\}$, the attacker seeks to learn $\mathsf{u} \in \mathsf{S} - \mathsf{D}$ (e.g., a new email to spam), $\mathsf{p} \in \mathsf{S} - \mathsf{D}$ (e.g., a new password to add to a cracking dictionary), or a new credential $(\mathsf{u}, \mathsf{p}) \in \mathsf{S} - \mathsf{D}$. In the case of an adversarial server where a client has access to (u, p), the threat landscape is larger. An adversarial server may learn the client's identity u (even if $\mathsf{u} \in \mathsf{S}$, this enables tracking), a client's password p (even if $\mathsf{p} \in \mathsf{S}$, this identifies active usage), or the credential (u, p) (even if $(\mathsf{u}, \mathsf{p}) \in \mathsf{S}$).

To address these threats, we outline the minimum security and privacy requirements any implementation of the abstract protocol previously outlined in Figure 1 must satisfy. In the security notions discussed below, we work with *anonymity sets* (denoted K) that describe a set of values (in our case, user credentials) that are large enough to give clients plausible deniability about their data even if their membership in K is revealed. These sets must be carefully defined to avoid trivial constructions that are insecure. At a high-level, they must have a sufficiently large support jointly over usernames and passwords (to aid in plausible deniability regarding both), should "partition" the space of credentials in a somewhat uniform manner independent of any actual usernames or passwords, and roughly all values in an anonymity set should be equally likely to be the client's credentials. (A full discussion is deferred to Appendix A.)

Requester credential anonymity: A protocol provides requester credential anonymity if for every credential $(\mathsf{u}_1, \mathsf{p}_1)$, there exists a sufficiently large anonymity set K containing $(\mathsf{u}_1, \mathsf{p}_1)$ such that $\forall (\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{K}$:

$$\mathsf{CreateRequest}(\mathsf{u}_1, \mathsf{p}_1) \approx \mathsf{CreateRequest}(\mathsf{u}_2, \mathsf{p}_2). \qquad (1)$$

For two distributions A and B, we let $A \approx B$ denote the computational indistinguishablity of the two distributions—that no efficient adversary given samples from A and B can distinguish them apart much better than randomly guessing. Thus, clients with credentials from the same anonymity set create requests that are indistinguishable to the server. While a minimum $|\mathsf{K}|$ likely depends on the sensitivities of the client involved, we set an initial threshold at $|\mathsf{K}| > 50,000$. While the IP address tied to a client's request reduces $|\mathsf{K}|$, a client can rely on a mix network such as Tor to prevent this leakage. IP address anonymity is out of scope of our threat model.

Responses with bounded leakage: Given a request for (u, p), the response from a breach alerting service should bound the information leaked, denoted L, about the membership of other credentials in S. To do this, we require an efficient simulator Sim that given only L can act as the server without being noticed by the client:

$$\mathsf{CreateResponse}(\mathsf{S}, \mathsf{Req}) \approx \mathsf{Sim}(\mathsf{L}, \mathsf{Req}). \qquad (2)$$

The presence of a successful simulator shows that the client may learn at most L by looking at responses from the server. Ideally, we want leakage to consist of only the membership of the queried credential and the anonymity set:

$$\mathsf{L} = ([(\mathsf{u}, \mathsf{p}) \in \mathsf{S}], \mathsf{K}). \qquad (3)$$

We can rephrase this security notion as follows. For any $(\mathsf{u}_1, \mathsf{p}_1), (\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{K}$ such that their membership in S is identical, i.e., $[(\mathsf{u}_1, \mathsf{p}_1) \in \mathsf{S}] = [(\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{S}]$:

$$\begin{aligned} &\mathsf{CreateResponse}(\mathsf{S}, \mathsf{CreateRequest}(\mathsf{u}_1, \mathsf{p}_1)) \\ &\approx \mathsf{CreateResponse}(\mathsf{S}, \mathsf{CreateRequest}(\mathsf{u}_2, \mathsf{p}_2)). \end{aligned} \qquad (4)$$

In other words, our security notion implies that the responses to credentials with identical leakage will be computationally indistinguishable.

Inefficient oracle: Learning u, p, or $(\mathsf{u},\mathsf{p}) \in \mathsf{S}$ via the breach alerting service should be equally or less efficient compared to guessing attempts performed on the login portal where the account originates from. Alternatively, a pragmatic attacker should be better off finding a plaintext copy of the breach. Let $\mathsf{t}(\mathsf{f})$ denote the running time of the function f. We capture this for a remote attacker as there being a time period T such that:

$$\mathsf{t}(\mathsf{CreateRequest}(\mathsf{u_i},\mathsf{p_i})) > T, \tag{5}$$

for every $(\mathsf{u}_i,\mathsf{p}_i)$. This requirement extends to an attacker with direct access to S due to an insider risk, a court order, or a breach of the alerting service's database. We frame this as merely checking the membership of a credential:

$$\mathsf{t}\left([(\mathsf{u},\mathsf{p}) \in \mathsf{S}]\right) > T'. \tag{6}$$

Ideally, $T = T'$, such that local access to S provides no advantage compared to the access mediated by the protocol. We consider a protocol where $T > 1$ second to satisfy this requirement.

Resistance to Denial of Service: A response from the server should not require significantly more computation than a request by a client (including bogus requests). As such, it should be difficult for an attacker to find a sequence $(\mathsf{u_1},\mathsf{p_1}),\ldots,(\mathsf{u_n},\mathsf{p_n})$ such that:

$$\sum_i \mathsf{t}(\mathsf{CreateRequest}(\mathsf{u_i},\mathsf{p_i})) \ll \sum_i \mathsf{t}(\mathsf{CreateResponse}(\mathsf{Req_i})) \tag{7}$$

Where $\mathsf{t}(\mathsf{f}(\cdot))$ denotes the running time of the function f.

Non-threats: Some threats are explicitly outside our threat model. These include an attacker attempting to confirm whether a breach they have access to is known to the alerting service (e.g., $\mathsf{D} \subseteq \mathsf{S}$), as well as an attacker learning $|\mathsf{S}|$. Such information may instead be beneficial to have public, allowing the service to publicly communicate which breaches it covers.

2.4 Tradeoffs of existing schemes

Existing breach alerting services include HaveIBeenPwned and PasswordPing, both of which have publicly documented APIs [26, 43]. Clients for each service include the 1Password [48] and LastPass [42] password managers. GitHub relies on a local mirror of HaveIBeenPwned's password dictionary for detection [36]. Firefox uses HaveIBeenPwned to warn users when they browse to a site that previously suffered a data breach, or if users supply their email address to Firefox [13]. We examine the tradeoffs these protocols make in terms of our design principles and threat model, with Table 1 serving as summary.

Query by username: HaveIBeenPwned and PasswordPing both support querying a specific plaintext username u. PasswordPing also supports querying $\mathsf{H}(\mathsf{u})$, the SHA256 hash of a username. In response, both services provide a list of breaches that the specified user was affected by and the class of data exposed (e.g., password, physical address). Lastpass currently relies on the username-only protocol from PasswordPing for breach alerting (after user consent).

Query by	Setup	Actionable, not informational	Breached, not weak	Near real-time	Requester credential anonymity	Inefficient oracle	Bounded leakage response	Resistant to Denial of Service
Username	Plaintext			●			●	●
	Hash			●			●	●
Password	Plaintext	●		●			●	●
	Hash	●		●			●	●
	Hash prefix	●		●	●			●
Domain	Plaintext			●	●	●	●	●
Username, then password	Plaintext, hash	●	●	●		●		●
	Hash, hash	●	●	●		●		●

Table 1: Summary of protocols supported by HaveIBeenPwned and PasswordPing and their tradeoffs according to our design principles and threat model.

In terms of our threat model (see Table 1), $\mathsf{H}(\mathsf{u})$ creates a unique, stable identifier of the user that is possibly reversible via a dictionary attack. This fails our requirement of requester credential anonymity. Likewise, querying u directly leaks the user's identity. Neither $\mathsf{H}(\mathsf{u})$ or u provides a computational hurdle, thus providing an efficient oracle for performing reconnaissance on victims. Knowledge of which breaches a victim is involved in can expose the victim to extortion, similar to recent scams that include breached data to coerce victims into paying the attacker by misrepresenting wider access [31].

Revisiting our design principles, we find that username-only protocols fail to satisfy our requirement of actionable rather than informational breach warnings. Users may have changed their password, or no longer use the account involved. Likewise, isolating responses solely to the types of data exposed fails to alert users to breached passwords that they reuse across multiple sites, where just one of the sites involved might be breached.

Query by password: PasswordPing allows clients to send a plaintext password p, or $\mathsf{H}(\mathsf{p})$ using SHA1, SHA256, or MD5. Both PasswordPing and HaveIBeenPwned provide a more secure alternative, whereby clients supply an N-bit prefix

$\mathsf{H}(\mathsf{p})_{[0:\mathsf{N}]}$. The server then returns all known breached passwords with that prefix, with the client performing the final exactness check locally. PasswordPing uses a 10-hex character prefix of a SHA1, SHA256, or MD5 hash; HaveIBeenPwned uses a 5-hex character prefix of a SHA1 hash. 1Password currently relies on HaveIBeenPwned and the password-prefix approach for breach alerting.

As detailed in Table 1, while supplying p explicitly exposes a client's non-breached password, revealing even $\mathsf{H}(\mathsf{p})$ leads to a potential pre-computed dictionary attack by an adversarial server. This threat is simplified by the lack of salt. As such, both schemes fail to provide requester credential anonymity. In the prefix-based variant, the same attack reduces the search space necessary by 2^N, with the attacker prioritizing guesses based on a password's popularity. With a sufficiently small N, this meets our criteria for anonymity—though weakly. We provide a deeper treatment of our rationale in Appendix A. However, as the response contains multiple passwords per lookup, this does not satisfy our requirement for bounded leakage. An adversarial client can enumerate each bucket to acquire a local copy of all $\mathsf{H}(\mathsf{p})$ for offline cracking to rebuild the underlying password dictionary.[1] While there is a legitimate argument that an attacker could more easily acquire a plaintext copy of the data breach, ideally any such protocol should also work for more sensitive breach data that is not widely accessible.

From a design perspective, we find that password-only protocols run the risk of alerting users to merely weak passwords. If u_1 in a breach shares the same password as u_2 who was not in any breach, there is no way to curate the security advice to both users' circumstances.

Query by domain: Both HaveIBeenPwned and PasswordPing provide a protocol for determining whether a domain was part of a breach. Firefox currently uses HaveIBeenPwned to warn users when they visit a domain that's previously suffered a breach [9]. This alert specifies that if they had an account, their data may no longer be secure. While these domain-only protocols satisfy every requirement laid out in our threat model (assuming the list of insecure domains is locally cached rather than queried), they provide neither actionable advice nor specific insights into breached rather than weak passwords. For example, a site visitor may have registered an account after the breach date. Likewise, domain-only protocols cannot capture the risk of password re-use across breached and non-breached sites.

Query by username, then password: PasswordPing provides a protocol whereby a client first queries u or $\mathsf{H}(\mathsf{u})$ using SHA-256, in turn receiving a salt s associated with that account. The client uses this to calculate $\mathsf{H}(\mathsf{u},\mathsf{p},\mathsf{s})$ via Argon2, sending only the N-bit prefix $\mathsf{H}(\mathsf{u},\mathsf{p},\mathsf{s})_{[0:\mathsf{N}]}$. PasswordPing relies on a 10-hex character prefix. The server responds with all known matching credentials, allowing a client to perform the confirmation locally. This approach satisfies all of our design principles. Additionally, due to the use of Argon2, the hash complexity involved compared to SHA or MD5 satisfies our requirement of an inefficient oracle. While we can bound the leakage of this protocol, it leaks information about both a requester's identity as well as multiple $\mathsf{H}(\mathsf{u},\mathsf{p},\mathsf{s})$ per response enabling offline attacks. (The s prevents pre-computed dictionary attacks.) This protocol bears a close resemblance to ours, but we satisfy all the criteria laid out in Table 1 and show in Section 3.2 how to further protect users' password information when querying by username.

2.5 Alternative cryptographic protocols

Our threat model is closely related to several well-studied cryptographic primitives. These protocols offer stricter privacy guarantees, but are computationally burdensome for a network setting in practice. As such, our threat model uses a relaxed requirement of anonymity. Secure hardware enclaves would also enable stricter privacy guarantees, but current enclaves have been shown to be vulnerable to side-channel and speculative execution attacks [53, 54].

Private Information Retrieval (PIR): PIR protocols, introduced by Chor et al. [6], require that a user be able to query an item from a server without revealing which item was queried. While PIR protocols which are secure against computationally-bounded adversaries [32] exceed our requester anonymity and password secrecy requirements, their security guarantees are one-sided—they allow the server to leak arbitrary information about the database to the clients. Additionally, single-server PIR protocols require communication that is effectively comparable to the size of the database [22]. Multi-server PIR protocols reduce this overhead, and even offer security guarantees against adversarial clients [19], but require that users trust that there is no risk of collusion amongst servers.

Oblivious Transfer (OT): 1-out-of-N OT protocols [8, 46] extend the PIR threat model to also require that a client learns no information about unaccessed elements of the server's database during the query. (Here N refers to the number of database entries.) While OT appears to capture the ideal requirements for a breach alerting protocol, we note that without weakening its security requirements, OT turns out to be a powerful crypto primitive [29] and requires communication overhead proportional to N.

Private Set Intersection (PSI): PSI protocols allow two parties with sets S_1 and S_2 respectively to compute some functions each of $S_1 \cap S_2$ and learn nothing more about each other's sets. We can model our use case as PSI where the client has a singleton set and the server learns nothing (an additional requirement needed in our work not typically seen in PSI).

[1] HaveIBeenPwned provides a direct download to every password in its corpus (hashed via SHA1), so this enumeration step is unnecessary and something the service argues is outside their threat model.

Early works leading to PSI [24,38] are based off of the Diffie-Hellman assumption which we also leverage in our protocol. While PSI protocols based on OT have been shown to be the fastest in practice, they require significant communication overhead that is unsuitable for a network setting [45]. Additionally, they are designed for settings where both parties have large, balanced sets which does not map to our scenario.

2.6 Ethics

Providing a breach alerting service necessitates access to credentials that were illicitly obtained and then released. For our work, we exclusively rely on credential breaches that are now publicly accessible, which any sophisticated attacker is likely to already have access to. As such, we argue that making this information accessible to users and identity providers does not materially increase the potential for harm—but that any protocol should have measures in place to protect against abuse. Passwords exposed by breaches have a history of research applications including improving password strength meters [11,39,57] and studying password use in the wild [10]. Surveyed users have also expressed a positive attitude towards breach alerting services, particularly in the context of password resetting [28]. We believe the potential to reduce account hijacking outweighs any risk of collating already public credential data.

3 Breach alerting protocol

Our design for a data breach alerting protocol relies on a combination of k-anonymity, private set intersection, and computationally expensive hashing to address all the risks outlined in our threat model. Here, we detail the cryptographic primitives we use to implement our protocol and the data exchanged between a client and server. We consider two variants: one that leaks some bits of password material that is secure against a resource-constrained attacker (e.g., the attacker is unable to circumvent k-anonymity and expensive hashing); and one that leaks zero bits of password material, but where clients must spend twice as much time hashing and receive weaker bounds on requester anonymity.

3.1 Resource-constrained attacker variant

CreateDatabase: Prior to any client lookup, the server must construct a secure database containing all known breached credentials. We outline this process in Algorithm 1. The server first canonicalizes the username associated with a credential by removing any capitalization and stripping information related to email providers (e.g., `user@gmail.com` becomes `user`). This step aids in de-duplication while also enabling us to detect reuse across sites that exclusively use usernames rather than email addresses. Post-canonicalization, the server calculates a computationally expensive hash of both the canonical username and credential password. We rely on Argon2 with a configuration that uses a single thread, 256 MB of memory, and a time cost of three.[2]

The server then blinds the 16-byte hash output with a 224-bit secret key b by mapping the hash to the elliptic curve NID_secp224r1 and raising the resulting point to the power b.[3] The server saves only a 2-byte prefix of hash unblinded which it uses for partitioning the entire database, where we denote a partition as S'. Here, hashing satisfies our requirement for an inefficient oracle even in the event that an attacker gains direct access to the underlying database. Blinding serves as an additional layer of defense in the event of a breach, but also to prevent information leakage and ensure requester anonymity and password secrecy via private set intersection (detailed shortly). As the key b has no external dependencies, the server can rotate it regularly by first decrypting old records and then re-blinding with a new key b'.

CreateRequest: When generating a request, a client repeats the same hashing and blinding strategy as the server. We outline this process in Algorithm 2. In contrast to the server, the client adopts its own secret key a which it initializes per request. The resulting request includes the 2-byte hash prefix and the blinded full hash. This 2-byte prefix—while leaking some bits of password material—provides the client with k-anonymity over the universe of all username and password pairs (not just those in breaches). Previous investigations of password usage estimate that users have roughly 6–8 unique passwords [16,44,56]. With an estimated 3.9 billion Internet users in the world [52], if we assume each user has just one unique username, this amounts to an estimated 23.4–31.2 billion unique credential pairs. As a rough approximation then, a user will share their credential prefix with 357,000–476,000 other credentials. Even if an adversarial server were to precompute a dictionary of the most popular passwords, they would have to repeat this process for each individual username. As such, our protocol satisfies our computational requirement for requester anonymity and password secrecy. In the case of an adversarial client, any request for a guessed credential is gated on the successful computation of an expensive hash, thus satisfying our requirement for an inefficient oracle.

CreateResponse: A server responds to a request according to Algorithm 3. Given a hash prefix, the server returns all known unsafe credentials S' tied to the prefix. While ideally we could provide the entire blinded contents of S to a client, in practice this is too computationally expensive as $|S|$ scales to billions of records. By partitioning S, we can limit the data downloaded to a client while ensuring membership correctness, at the cost of working with anonymity sets rather than perfect se-

[2] According to libsodium, this amounts to roughly 0.7 seconds on a 2.8 Ghz Core i7 CPU [34].

[3] We use multiplicative notation to refer to elliptic-curve group operations in the paper.

Algorithm 1 CreateDatabase: Store a blinded and strongly hashed copy of all known breached credentials.

```
Require: $S = \{(u_1, p_1), \ldots, (u_n, p_n)\}$, $b = rand()$, and $n = 2$,
    a prefix length
function CreateDatabase($S, b, n$)
    for $(u_i, p_i) \in S$ do
        $u'_i \leftarrow$ Canoncialize($u_i$)
        $H \leftarrow$ Hash($u'_i, p_i$)
        $H^b \leftarrow$ Blind($H, b$)
        $H_{[0:n]} \leftarrow$ ByteSubstring($H, n$)
        PartitionStore($H_{[0:n]}, H^b$)
    end for
end function
```

Algorithm 2 CreateRequest: Client query to determine whether a blinded username and password with a cleartext hash prefix was exposed in a breach.

```
Require: $n$, a prefix length
function CreateRequest($u, p, n$)
    $a \leftarrow$ Rand()
    $u'_i \leftarrow$ Canoncialize($u$)
    $H \leftarrow$ Hash($u', p$)
    $H^a \leftarrow$ Blind($H, a$)
    $H_{[0:n]} \leftarrow$ ByteSubstring($H, n$)
    LocalStore($a$)
    return HstsRequest($H_{[0:n]}, H^a$)
end function
```

Algorithm 3 CreateResponse: Server response for all information known about the cleartext hash prefix.

```
Require: $b = rand()$
function CreateResponse($H_{[0:n]}, H^a$)
    $H^{ab} \leftarrow$ Blind($H^a, b$)
    $S' \leftarrow$ PartitionLookup($H_{[0:n]}$)
    return HstsResponse($H^{ab}, S'$)
end function
```

crecy. As noted in Section 2.5, the best current constructions dictate that without partitioning S, we cannot hope to deploy a scheme with reasonable limits on data downloaded by clients and computation performed by the server. By avoiding any client nonce or salt for hashing, retrieval is entirely static for the server apart from inexpensive blinding (at least compared to hashing). This satisfies our requirement for resistance to denial of service.

Providing S' absent blinding would leak information about other exposed credentials. Instead, we rely on Diffie-Hellman private set intersection [24] which is relatively efficient for a network setting on non-mobile devices [45]. The server returns all known breached credentials blinded with b while providing a client with an index into the doubly-blinded list H^{ab}. This requires the commutative properties of elliptic curve Diffie–Hellman (ECDH) such that the client can decrypt this result to recover H^b during verification, while the remaining contents of S' remain hidden.

More formally, under the random oracle model [3], with Argon2 modeled as a perfect hash function, our hash-and-blind scheme implements an oblivious pseudorandom function (OPRF) against honest-but-curious adversaries under the decisional Diffie-Hellman assumption. When b is kept secret, outputs of the hash-and-blind scheme on any user inputs $(\mathsf{u}_i, \mathsf{p}_i)$ reveal no information about the hashed and blinded output on *any other* $(\mathsf{u}', \mathsf{p}')$. A more technical and detailed note is laid out in Appendix B. This achieves bounded leakage and given only the leakage L as defined in Section 2.3, we can construct a Simulator to simulate the entire response of the server.

Verdict: Finally, a client determines whether their credential was exposed in a breach by finishing the private set intersection protocol as detailed in Algorithm 4. This process is entirely local and, absent independent telemetry, never reveals the verdict of a match to the server.

3.2 Zero-password leakage variant

Our previous approach makes a practical tradeoff between client hashing overhead and revealing some bits of a client's password. (While still protected by a computationally expensive hash and anonymity sets spanning both usernames and passwords, this information can be leaked if an attacker has auxiliary information about the username.) As an alternative, we outline a zero-password leakage variant. In Algorithm 2, a client now calculates a hash prefix of only the username $\mathsf{H}(\mathsf{u}')_{[0:n]}$ along with a blinded hash of the entire credential. Algorithm 1 is modified to create a mapping between $\mathsf{H}(\mathsf{u}'_i)_{[0:n]}$ to $\mathsf{H}(\mathsf{u}'_i, \mathsf{p}'_i)$ and to use it to partition the database by $\mathsf{H}(\mathsf{u}'_i)_{[0:n]}$. This variant still provides the same protection with bounded leakage, denial of service resistance, and an inefficient oracle, and modifies (and reduces) the anonymity set of credentials to only usernames. For an estimated 3.9 billion unique usernames, this amounts to $|\mathsf{K}| = 60,000$.[4] However, this variant ensures that all password material from the client is protected by blinding. In practice, given near real-time constraints, this requires that a client spend twice as much time hashing which is non-negligible.[5] For the purposes of our initial deployment (detailed in Section 5), we opted for the first variant to understand the computational bounds of clients. We now plan to migrate to the zero-password leakage variant.

[4] With no password guessing required, it also enables an attacker to reasonably pre-compute the Argon2 hash of all possible usernames.

[5] This expense can be amortized if the client reuses their username for multiple sites with distinct passwords, or if the client regularly polls the server for the same username to obtain the most recent breach status.

Algorithm 4 Verdict: Final client-side verdict for whether a username or password was exposed in a breach.

```
Require: $a$, secret key for original request
function VERDICT($H^{ab}, S', a$)
    $H^b \leftarrow$ UNBLIND($H^{ab}, a$)
    return $H^b \in S'$
end function
```

3.3 Expansion to metadata

Our protocol currently does not include information on the origin of an exposed credential as metadata (e.g., which service was compromised). In practice, we believe this is the best strategy as origin information is both untrustworthy and often unavailable. For example, large composite breaches such as Collection 1-5 and Antipublic include hundreds of millions of credentials, all of which are unattributed [21,25]. Moreover, metadata expands the size of data downloaded as part of S'.

For completeness, our protocol can be extended to include origin information, or any metadata, by encrypting it with the output of a cryptographically secure key-derivation function such as HKDF [30] applied to $\mathsf{H}(\mathsf{u},\mathsf{p})$. This approach limits access strictly to clients that prove knowledge of the associated, strongly hashed username and password. This is easy to observe; as outlined in Appendix B, the hashed-and-blinded outputs still hide information about other usernames and passwords and hence the derived keys are cryptographically strong and hide the contents of encrypted metadata. This is only done once when creating the database and adds very little overhead to the system. We note that it is crucially important that this metadata not include sensitive personally identifying information as it is not hidden from a compromised service.

3.4 Limitations

Our protocol requires that clients are capable of computing an expensive hash with 256MB of memory. This is a necessary requirement to hamper attackers, but it may also prove untenable for resource-constrained devices. Additionally, our approach requires that clients download a non-negligible amount of data. For context, with 1 billion credentials uniformly split into 2^{16} prefixes, this equates to roughly 15,000 blinded hashes per request. At 29-bytes per item, that amounts to roughly 435KB on average. This grows linearly with the volume of newly discovered credentials.

4 Implementation

We implemented our protocol as a publicly accessible API hosted on Google Cloud. The API mediates access to over 4 billion unique usernames and passwords collected using an approach previously documented by Thomas et al. [51]. Canonicalization further reduces this set to 3.36 billion credentials. We also developed a Chrome extension as a proof of concept client that we could share among early testers to gather telemetry on the frequency and impact of breach notifications in the wild. In practice, other applications that handles credentials can integrate with our service by implementing the client half of our protocol.

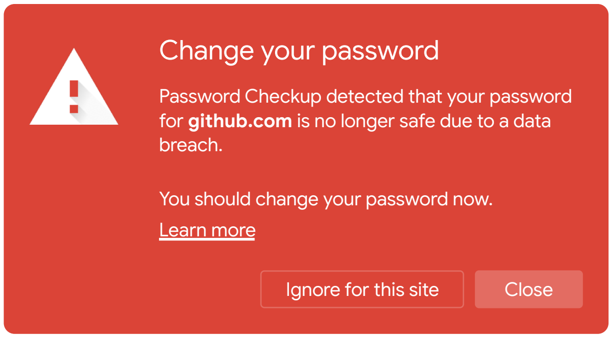
4.1 Client

Our Chrome extension monitors when users submit their username and password on a login page and generates a browser warning for breached credentials detected by our API. We rely on a JavaScript implementation of Argon2 from libsodium for all hashing and a web assembly compilation of OpenSSL for the elliptic curve computation required for private set intersection. Both libraries are open source with multiple years of vetting. Here, we discuss the details behind our extension, the design of our warning dialogues, and the telemetry the extension collects.

Detecting login events: At present, Chrome does not export an API for detecting login events. Instead, our extension registers a callback function to interpose on all `webRequests` that contain form data. When triggered, the extension relies on heuristics to detect whether the form contains a username or password field, such as matching on field names like `password` and `passwd`. If the heuristic fails to detect both a username and password, nothing is sent to our API. We manually tested our detection on the Alexa US Top 50: we successfully captured login events for 40 pages and failed for 4, while the remaining 6 did not have login forms. For the failures, login information was either obfuscated (e.g., a byte blob of all field data), or part of the payload body rather than form data. We are thus cautious when discussing data from our real world deployment in Section 6 that not every domain will be covered by our technique.

Warning design: Our extension modifies the DOM of the page where a user entered their breached credential to show a warning similar to Figure 2. In the browser tray, users can reach an extension popup that displays a stateful warning –similar to Figure 3. This gives users a way to see past warnings, in the event that they closed their browser tab before reviewing the warning (or due to a DOM refresh that overwrites our modifications). Additionally, this serves as a secure UI element that runs in isolation of other extensions and pages. Both styles of warning never reveal information about the username or password found in a breach. This design decision limits the context we can provide users, but allows us to avoid storing sensitive credential material that might make persistent local storage a target for attacks.

In designing our warning, we followed emerging advice about data breach notifications [20], proven terminology around data breaches [1,28], and historical studies of browser



Change your password

Password Checkup detected that your password for **github.com** is no longer safe due to a data breach.

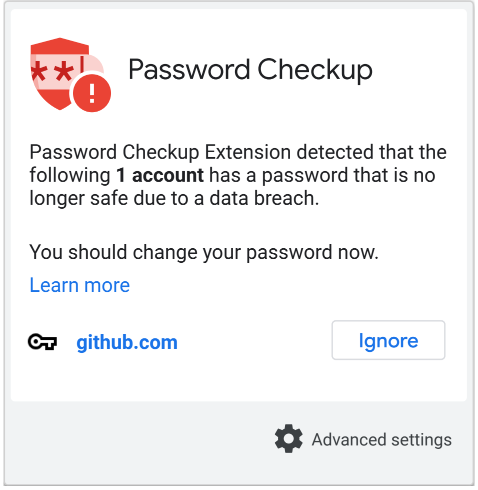
You should change your password now.

Figure 2: In-page warning generated by our extension when we detect that a credential is no longer secure due to a breach.

warnings related to phishing and unsafe network connections [2, 15]. In particular, we provided a clear action—"Change your password"—along with context for the danger behind the event. At the same time, we minimized unnecessary or overly technical information. We also provided a “Learn More” link that explained in greater detail the root cause of the warning and security best practices. In particular, users should (1) reset their password for the affected page; (2) reset their password wherever it was reused; (3) consider a password manager; and (4) consider adopting two-factor authentication. We collected feedback from 550 early testers from our organization before settling on the final design and language of our dialogue.

Compared to other browser warnings where the safest action is to close the tab, breached passwords require users to follow a series of unguided, proactive next steps. We emphasize unguided as there is no canonical account security page for every site to simplify password resetting. While there are industry initiatives to create common reset paths [41], these have yet to materialize. As such, we consider a more formal usability study of the warning experience—and automating the password change process—as future work. We provide a deeper treatment of the effectiveness of our warnings in terms of successful password resets later in Section 6 (in short, a quarter of warnings result in a reset during our observation period).

Identifying user actions: By default, our extension continuously triggers a warning each time the user authenticates with a breached credential. Given the computation and network overhead involved for each API query, if the extension detects a breached credential, it caches a 12-byte prefix of the Argon2 credential hash to avoid generating a new API query for the same credential. This also reduces the latency between a user entering a credential and observing a warning on all subsequent logins to the same domain. Conversely, if a credential was previously not present in a breach, we avoid caching any verdict and perform a new API query on each login. In the future, caching here is also possible if the cache were invalidated upon the server announcing the arrival of a new breach.



Password Checkup

Password Checkup Extension detected that the following **1 account** has a password that is no longer safe due to a data breach.

You should change your password now.

Figure 3: Stateful icon tray warning message to remind users which accounts need their attention. This avoids the transient nature of in-page warnings, which we use to provide better context to users.

For low-value accounts that a user might deem unnecessary to secure, we provide an option to ignore our warning on a per-domain basis as shown in Figure 2 and Figure 3. The extension manages this state by caching a local copy of the domain involved and a 12-byte prefix of the Argon2 hash of the credential that the user ignored (which is necessary in the event the user has multiple accounts on the domain).

We detect when a user resets their exposed password in order to provide a positive feedback signal to the user that their account is no longer at risk. We also purge all cached information about the now stale credential. To do this, we cache a 12-byte Argon2 prefix of an account’s username (with only an 8MB memory requirement)—used only locally— along with a 12-byte prefix of the Argon2 credential hash. If the credential hash changes for the same username, this indicates the user signed in with a new password and that all local state for the credential should be reset. In the event a user merely mistyped their breached password, correct password entry will trigger a new warning and refresh the cache.

Telemetry: We instrument our extension to report anonymous telemetry pertaining to the volume of lookups against our API that result in a breach warning, along with whether users ignore our warnings or reset their passwords. All of these events lack any form of user identifier, precluding the possibility of correlating events or understanding per-user experiences. Each event also includes the domain of the login page involved, which we use to estimate our compatibility with popular sites and to estimate the prevalence of breached passwords across the Internet. For password changes related to breached credentials, we also report the strength of the old and new password to understand whether users as a whole migrate to stronger passwords. We use zxcvbn [57] for strength estimation as it is entirely client-side and open source. This telemetry forms the basis of our analysis of the impact of password breach warnings in the wild, discussed in Section 6. We disclose the data we collect upfront to users in the description of our Chrome Webstore listing.[6] We had all of our telemetry reviewed by a group of internal experts and followed our organization's ethics review process.

4.2 Storage

We partitioned our pre-computed, blinded and hashed credential corpus (totaling roughly 110GB) into 2^{16} slices. We stored each slice as a static file in Google Cloud Storage. We restricted access to these files so that only the server handling requests could fetch content from storage. We also stored the key material necessary to re-blind client-blinded hashes in the same storage system.

4.3 Server

The stateless nature of our credential breach protocol allowed us to implement our serving using Google Cloud Functions. The primary benefit of this approach is that we could scale arbitrarily to the volume of incoming requests while also avoiding dormant compute cycles on pre-requisitioned cloud instances. This design also allowed us to avoid having to reason about the side-effects across requests. We implemented our Cloud Function using the same JavaScript elliptic curve library as our Chrome extension (recall that hashing is not part of the server protocol). We avoid application-layer denial of service attacks—such as sending an arbitrary length string for the server to blind—by blocking malformed requests that do not adhere to the fixed-length blinded hash we expect from a client.

5 Deployment

We made our extension publicly available via the Chrome Web Store and announced it through major media channels. In total, 667,716 users installed our extension over a measurement period of February 5, 2019–March 4, 2019 (UTC).

User demographics: Based on aggregate statistics provided by the Chrome Web Store, 48% of the users who installed our extension were from North America, 29% from Europe, 17% from Asia, and the remaining 6% from around the world. In terms of operating systems, 71% of users who installed the extension used Windows, 14% used MacOS, 13% ChromeOS, and 2% Linux. We note that extensions are unavailable on mobile devices and thus are not present in our device breakdown.

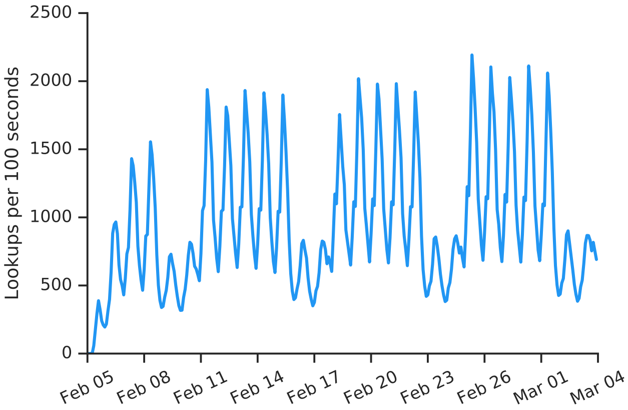


Figure 4: Volume of logins scanned by our extension every 100 seconds. Requests to our API scaled from 0.11 queries per 100 seconds in early testing, to a peak of 2,192 queries per 100 seconds at the end of our measurement window. The dips in the graph reflect lower activity during weekends.

Scaling to requests: Over the course of our measurement window, the lookup volume to our API scaled gracefully from 0.11 lookups per 100 seconds during early testing to a peak of 2,043 lookups per 100 seconds as shown in Figure 4. The diurnal pattern present reflects the geographic concentration of users in North America and Europe. The periodic dips reflect lower login activity over the weekend. By comparing query volume with active user metrics provided by the Chrome Web store, we estimate that an average user generates 3 API requests (e.g., logins) per weekday, and 1.5 requests per weekend. Critically, the diurnal cadence and lack of bursty behavior indicates a lack of large-scale abuse during our measurement window which might otherwise pollute our analysis later in Section 6.

Client overhead: We present a breakdown of the computational overhead and network latency incurred by clients that query our API in Table 2. Overall, a median query took 8.5 seconds to return a verdict, during which a user would continue browsing uninterrupted. Roughly half of this time was spent strongly hashing the user's credential, while the remaining time was spent downloading potential credential matches. Our username hash (used for locally caching state) took a median of 100ms and was a negligible part of this delay. For 10% of users, the overall query time exceeded 18 seconds, half of which was spent in network latency. While part of this lookup overhead can be optimized—credential hashing in native code takes an average of 0.7 seconds—the only way to reduce network latency would be to download fewer breached records, thus reducing the k-anonymity set of our protocol. As such, our current privacy constraints likely remain out of reach for resource-constrained devices, at least for near real time detection.

[6] *https://chrome.google.com/webstore/detail/password-checkup/pncabnpcffmalkkjpajodfhijclecjno*

Duration	Median	90%	95%
Argon2 username hash	0.1s	0.3s	0.3s
Argon2 credential hash	4.4s	9.8s	12.7s
End-to-end API query	8.5s	18.8s	26.9s

Table 2: Time spent performing API operations including hashing and downloading potentially matching breached credentials.

Cost modeling: A practical reality of running a breach detection service is cost. In our case, cost is intrinsically tied to the k-anonymity privacy that we provide. Every 1,000 invocations of our API costs approximately $0.19 at the current volume of credentials in our storage and for a 2-byte k-anonymity prefix. Data serving makes up 94% of this cost, while the CPU and memory necessary to field requests and to re-encrypt client credentials makes up only 5%. Based on our query volume per user, operating our service for an estimated 500,000 users would cost $85,500 a year. Caching the status of negative breach verdicts would substantially reduce expenses. Our goal in documenting these details is to provide other members of the community a benchmark for the costs of any improved privacy scheme. For our protocol, adding a single bit of privacy nearly doubles our operating expenses while also doubling the network latency for clients.

6 Analysis

We analyzed the anonymous telemetry reported during our measurement window to understand the state of breached passwords across the Internet. Facets we consider include the frequency that users log in with a breached password, the types of sites where reuse is most common, and ultimately whether displaying warnings helps users to address the risk of credential stuffing. We provide a high-level statistical summary of our telemetry in Table 3 . We note that our telemetry is biased towards the users who installed our extension, which is a non-random sample of the Internet population.

6.1 Credential stuffing risk and remediation

Frequency of breached credential reuse: Overall, our API fielded 21,177,237 lookup requests, where a lookup maps to a single login attempt performed by an anonymous user. We detected that 316,531 logins involved breached credentials—roughly 1.5% of all logins. We caution this is a lower bound as we only generate telemetry for breached credentials once before caching the result locally, whereas lookups to non-breached credentials generate telemetry upon each new login. Our detection rate is lower than the 6.9% reported by Thomas et al. [51] for 751 million Google accounts and 1.9 billion breached credentials. Possible reasons include the user population that adopted our extension is more security conscious—thus avoiding reuse as a behavior—or that dormant accounts have a higher reuse rate, which by nature our extension cannot observe as we perform checks at login time. During our 28 day measurement window, if we assume that logins and warnings are uniformly distributed across users, 47.3% of our users received a warning. Our anonymous reporting precludes more detailed per-user statistics. Taken as a whole, our results reveal that global Internet users regularly access accounts that are vulnerable to credential stuffing.

Metric	Value
Extension users	667,716
Logins analyzed	21,177,237
Domains covered	746,853
Breached credentials found	316,531
Warnings ignored	81,368 (26%)
Passwords reset	82,761 (26%)

Table 3: Summary of the anonymous telemetry data reported over the course of our analysis window from February 5–March 4, 2019.

Ignoring breached credentials: Users opted to ignore 81,368—or 25.7%—of the breach warnings we surfaced. We consider three possible explanations. Users may be making an explicit risk assessment that the value of their account is not worth the effort of adopting a new password. Alternatively, users may not be in full control of the account (e.g., a shared household account) [37]. Finally, as our extension does not automate the process of password resetting, users may ignore our warning out of frustration due to a lack of guidance. Regardless of the underlying cause, ignored warnings leave accounts vulnerable to credential stuffing. That said, there is an opportunity here for identity providers to take action and guide users through the password resetting process.

Remediation of breached passwords: Our warnings resulted in users resetting 82,761—or 26.1%—of their breached passwords. Critically, we find that users used this opportunity to migrate to stronger passwords. On average, the passwords we detected as breached had a zxcvbn strength of 1.6. After remediation, this score increased to an average of 2.9. We present a more detailed summary of strength before and after resetting in Figure 5. For context, a score of one indicates a “weak password” that an attacker can guess in under 10^6 attempts. A score of two reflects a password that an attacker can guess in under 10^8 attempts, and a score of three 10^{10} attempts and is considered “strong”.

Overall, 94% of password changes led to a stronger or equal zxcvbn score, while just 6% of changes resulted in a regression to a weaker password. Our results indicate that users of our extension understand stronger password compo-

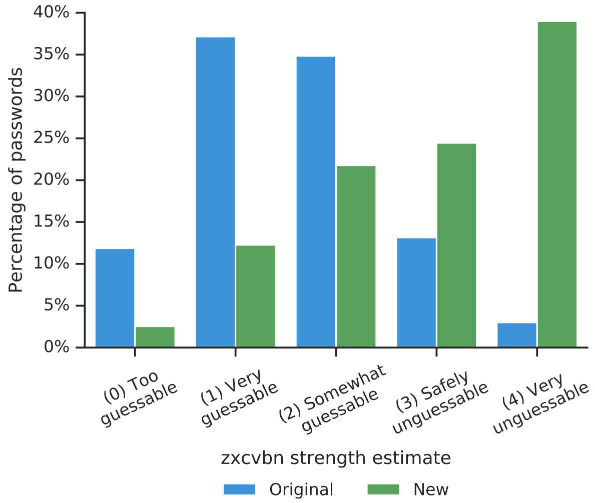


Figure 5: Histogram of zxcvbn password strength for passwords detected as breached and the password adopted by users after remediation. Users migrated towards stronger passwords overall as a result of our warnings.

sition strategies. Equally important, 39% of new passwords achieved the highest possible strength score (up from 3% for the original passwords), a potential sign of the growing prevalence of password managers that automatically compose strong passwords. Our results highlight how surfacing actionable security information can help mitigate the risk of account hijacking.

6.2 Influence of domains on account security

Category: We examine whether the perceived value of an account influences the rate that users rely on reused, breached credentials. To do this, we manually labeled the top 332 domains that received more than 5,000 logins during our measurement period into one of thirteen categories (e.g., finance, email and messaging, and social networking). We used a catch all "Other" category for domains that fell outside this categorization. Combined, logins to these domains accounted for 41% of lookups against our API.

We present a breakdown of the aggregate warning rates and ignore rates across all domains per category in Table 4. Domains that we categorized as related to finance or governments exhibited the lowest rate of reused, breached credentials (0.2–0.3%). Possible explanations include the password composition policies of these domains, the fact that users adhere to popular security advice to have one strong password for their bank, or that the sites actively identify breaches and previously forced password resets. In contrast, entertainment sites like streaming video platforms and adult websites had the highest warning rate for breached credentials (3.6–6.3%). Users may adopt disposable passwords due to perceived lack of risk, or in the case of streaming sites, they may use shared accounts. Surprisingly, users ignored our breach warnings nearly uniformly across categories, with the exception of adult websites. For the latter, users ignored nearly twice as many of our warnings—potentially to hide the domain from our persistent warning tray (see Figure 3 earlier).

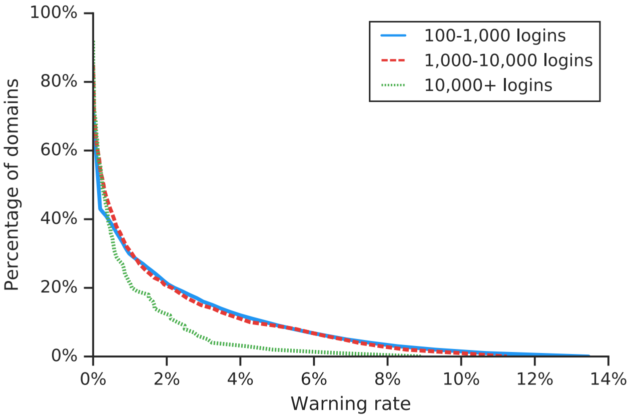


Figure 6: CCDF of the percentage of logins per domain that result in a warning across. We group domains by the volume of logins we observed, including 100-1,000, 1,000–10,000, and 10,000+. Popular sites tend to face less of a threat from credential stuffing, while the long tail of domains remain at risk.

Popularity: We also consider whether more popular sites are less vulnerable to credential stuffing. We present a CCDF of the frequency of warnings per domain versus the volume of logins to the domain during our analysis window in Figure 6. We find that just 6% of domains with 10,000+ logins have a warning rate higher than 3%, compared to 15% of domains with fewer than 10,000+ logins. We believe this gap in security results from larger security investments on the part of popular domains towards proactively resetting passwords and helping users avoid "weak" passwords. While large identity providers can equally take advantage of our API, addressing the long tail of domains affected by credential stuffing likely requires relying on in-browser warnings.

7 Related Work

Account hijacking threats: Credential stuffing represents just one dimension of account hijacking threats. Other risks include large-scale phishing [7,51], credential or token theft from local machines [50], and even targeted attacks [35,40]. Users have internalized these risks and adopted a security model of joint responsibility between themselves and identity providers [47]. The most prominent solutions to these threats include users adopting two-factor authentication, or identity providers expanding authentication to include other passive

Category	Domains	Total visits	Breakdown	Warning rate	Ignore rate
Finance	90	1,684,851	8.0%	0.3%	18.6%
Email, messaging	47	1,519,795	7.2%	0.5%	14.0%
Social networking	15	1,191,546	5.6%	0.8%	17.8%
Shopping	29	1,007,103	4.8%	1.2%	16.4%
Technology	34	624,702	2.9%	0.7%	16.9%
Business	12	585,797	2.8%	0.7%	20.3%
Education	16	261,563	1.2%	0.9%	26.5%
Gaming	11	201,646	1.0%	0.5%	18.6%
Entertainment	9	168,565	0.8%	6.3%	27.1%
Travel	14	138,968	0.7%	1.8%	19.6%
Government	5	60,967	0.3%	0.2%	16.9%
News	5	54,864	0.3%	1.9%	20.7%
Adult	3	50,408	0.2%	3.6%	38.5%
Other	42	429,786	2.0%	1.0%	17.8%

Table 4: Breakdown of reused, breached passwords for domains receiving more than 5,000 logins, aggregated by business sector. Finance and govt. domains had the lowest usage of breached passwords, compared to entertainment and adult-related domains.

factors such as a user's device and location [12, 17]. The protections we propose in this work are complementary to a defense in depth authentication model, where breach detection represents one additional factor in risk modeling.

Password reuse behaviors: Text passwords continue to be the prevailing mechanism for online authentication. Given the human constraints of memorizing a large number of unique text strings, people have adopted various strategies—including reuse and weak patterns—for managing their growing number of online identities [18, 23, 49, 55]. Florencio and Herley published the first large-scale study of password behavior, where they found both weak and reused passwords were a frequent flaw [16]. More recently, Wash et al. [56] and Pearman et al. [44] observed the password usage behaviors of hundreds of participants over multiple weeks. They estimated that 32% of all entered passwords involved exact reuse. Wash et al. found that users reused their most popular password on an average of 9 sites. Examining breach data directly, Das et al. found that 43–51% of users reused the same password on multiple sites [10]. While automated password filling has become more commonplace—participants used these means 57% of the time [44]—both Pearman et al. and Wash et al. found password managers have yet to be adopted as a tool for password generation. All of these factors compound the threat of credential stuffing, where inverting a single weak password hash can grant an attacker access to multiple sites.

Improving breach alerting protocols: In a contemporaneous work, Li et al. presented a framework for reasoning about leakages resulting from the password-based prefixes used by our protocol and HaveIBeenPwned [33]. The authors show how a password-only prefix (or an attacker with access to the plaintext username in a username-password prefix) can leverage a partition's underlying password distribution to reduce the number of guesses necessary to potentially learn a user's password. To address this, the authors outline a zero-password leakage variant that relies on private set membership in conjunction with a username hash prefix for partitioning, akin to our own model from Section 3.2. Their work provides further motivation for a zero-password leakage protocol, despite its additional computational complexity as we outlined.

8 Conclusion

In this paper, we demonstrated the feasibility of a privacy-preserving protocol that allows a client to query whether their login credentials were exposed in a breach, without revealing the information queried. Our protocol relies on a combination of computationally expensive hashing, k-anonymity, and private set intersection. Our approach improves on existing protocols by taking into account both an adversarial client and server, while also minimizing the chance of false positives. We envision this service being used by end users, password managers, and by identity providers. As a proof-of-concept, we created a cloud service that mediates access to 4 billion usernames and passwords publicly exposed by breaches. We then released a Chrome extension that would query credentials entered at login time against our service. Based on telemetry produced by nearly 670,000 users, we estimated that 1.5% of credentials used across the web are vulnerable to credential stuffing (based on a sample of 21 million logins).

Addressing this problem requires action from both users and identity providers. In the context of our study, 26% of the warnings we generated for breached passwords resulted in users adopting a new password—94% of which were stronger or as strong as the original. Both the volume of user interest and response rate surfaced during our study demonstrate that there is an appetite on the part of users to secure their accounts from credential stuffing. We hope that by making our protocol public, other researchers can improve on the privacy protections, computational bounds, and cost models that we establish. Our protocol is a first step in democratizing access to breach alerting in order to mitigate one dimension of account hijacking.

9 Acknowledgements

We would like to thank Oxana Comanescu, Sunny Consolvo, Ali Zand, and our anonymous reviewers for their feedback and support in designing our breach alerting protocol. This work was partially supported by funding from the NSF.

References

[1] Lillian Ablon, Paul Heaton, Diana Catherine Lavery, and Sasha Romanosky. Consumer attitudes toward data breach notifications and loss of personal information. In *Proceedings of the Workshop on the Economics of Information Security*, 2016.

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the USENIX Security Symposium*, 2013.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the ACM Conference on Computer and Communications Security*, 1993.

[4] Borbala Benko, Elie Bursztein, Tadek Pietraszek, and Mark Risher. Cleaning up after password dumps. *https://security.googleblog.com/2014/09/cleaning-up-after-password-dumps.html*, 2014.

[5] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the New Security Paradigms Workshop*, 2011.

[6] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1995.

[7] Marco Cova, Christopher Kruegel, and Giovanni Vigna. There is no free phish: an analysis of "free" and live phishing kits. In *Proceedings of the Workshop on Offensive Technologies*, 2008.

[8] Claude Crépeau. Equivalence between two flavours of oblivious transfers. In *Conference on the Theory and Application of Cryptographic Techniques*, 1987.

[9] Luke Crouch. When does firefox alert for breached sites? *https://blog.mozilla.org/security/2018/11/14/when-does-firefox-alert-for-breached-sites/*, 2018.

[10] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[11] Xavier De Carné De Carnavalet, Mohammad Mannan, et al. From very weak to very strong: Analyzing password-strength meters. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[12] Periwinkle Doerfler, Maija Marincenko, Juri Ranieri, Angelika Moscicki Yu Jiang, Damon McCoy, and Kurt Thomas. Evaluating login challenges as a defense against account takeover. In *Proceedings of the Web Conference*, 2019.

[13] Peter Dolanjski. Testing firefox monitor, a new security tool. *https://blog.mozilla.org/futurereleases/2018/06/25/testing-firefox-monitor-a-new-security-tool/*, 2018.

[14] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In *Proceedings of the USENIX Security Symposium*, 2015.

[15] Adrienne Porter Felt, Alex Ainslie, Robert W Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving ssl warnings: Comprehension and adherence. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2015.

[16] Dinei Florencio and Cormac Herley. A large scale study of web password habits. In *Proceedings of the International World Wide Web Conference*, 2006.

[17] David Mandell Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who are you? a statistical approach to measuring user authenticity. In *Proceedings of the Symposium on Network and Distributed System Security*, 2016.

[18] Shirley Gaw and Edward W. Felten. Password management strategies for online accounts. In *Proceedings of the Symposium on Usable Privacy and Security*, 2006.

[19] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences*, 2000.

[20] Maximilian Golla, Miranda Wei, Juliette Hainline, Lydia Filipe, Markus Dürmuth, Elissa Redmiles, and Blase Ur. What was that site doing with my facebook password?: Designing password-reuse notifications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2018.

[21] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. *https://www.wired.com/story/collection-leak-usernames-passwords-billions/*, 2019.

[22] Iftach Haitner, Jonathan J Hoch, and Gil Segev. A linear lower bound on the communication complexity of single-server private information retrieval. In *Proceedings of the Theory of Cryptography Conference*, 2008.

[23] Eiji Hayashi and Jason Hong. A diary study of password usage in daily life. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2011.

[24] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the ACM Conference on Electronic Commerce*, 1999.

[25] Troy Hunt. Password reuse, credential stuffing and another billion records in Have I been pwned. *https://www.troyhunt.com/password-reuse-credential-stuffing-and-another-1-billion-records-in-have-i-been-pwned/*, 2017.

[26] Troy Hunt. Have i been pwned? *https://haveibeenpwned.com/*, 2019.

[27] Stanisław Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *Proceedings of the International Conference on Security and Cryptography for Networks*, 2010.

[28] Sowmya Karunakaran, Kurt Thomas, Elie Bursztein, and Oxana Comanescu. Data breaches: user comprehension, expectations, and concerns with handling exposed data. In *Proceedings of the Symposium on Usable Privacy and Security*, 2018.

[29] Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Symposium on Theory of Computing*, 1988.

[30] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the Annual Cryptology Conference*, 2010.

[31] Brian Krebs. Sextortion scam uses recipient's hacked passwords. *https://krebsonsecurity.com/2018/07/sextortion-scam-uses-recipients-hacked-passwords/*, 2018.

[32] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373. IEEE, 1997.

[33] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. *https://rist.tech.cornell.edu/papers/c3.pdf*, 2019.

[34] libsodium. The Argon2 function. *https://libsodium.gitbook.io/doc/password_hashing/the_argon2i_function*, 2019.

[35] William R Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. When governments hack opponents: a look at actors and technology. In *Proceedings of the USENIX Security Symposium*, 2014.

[36] Neil Matatall. New improvements and best practices for account security and recoverability. *https://github.blog/2018-07-31-new-improvements-and-best-practices-for-account-security-and-recoverability/*, 2018.

[37] Tara Matthews, Kerwell Liao, Anna Turner, Marianne Berkovich, Robert Reeder, and Sunny Consolvo. She'll just grab any device that's closer: A study of everyday device & account sharing in households. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2016.

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

[39] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proceedings of the USENIX Security Symposium*, 2016.

[40] Ariana Mirian, Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, , and Kurt Thomas. Hack for hire: Exploring the emerging market for account hijacking. In *Proceedings of The Web Conf*, 2019.

[41] Theresa O'Connor. A well-known url for changing passwords. *https://wicg.github.io/change-password-url/index.html*, 2018.

[42] Password Ping. LastPass selects PasswordPing for compromised credential screening. *https://www.passwordping.com/lastpass-selects-passwordping-for-compromised-credential-screening/*, 2017.

[43] Password Ping. Block attacks from compromised credentials. *https://www.passwordping.com/*, 2019.

[44] Sarah Pearman, Jeremy Thomas, Pardis Emani Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural

habitat. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, 2017.

[45] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *Proceedings of the USENIX Security Symposium*, 2014.

[46] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical report, Tech. rep. TR-81, AikenComputation Laboratory, Harvard University, Cambridge, MA, 1981.

[47] Richard Shay, Iulia Ion, Robert W Reeder, and Sunny Consolvo. "My religious aunt asked why I was trying to sell her viagra": experiences with account hijacking. In *Proceedings of ACM Conference on Human Factors in Computing Systems*, 2014.

[48] Jeff Shiner. Finding pwned passwords with 1password. *https://blog.1password.com/finding-pwned-passwords-with-1password/*, 2019.

[49] Elizabeth Stobert and Robert Biddle. The password life cycle: User behaviour in managing passwords. In *Proceedings of the Symposium on Usable Privacy and Security*, 2014.

[50] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[51] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[52] International Telecommunications Union. Statistics. *https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx*, 2019.

[53] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[54] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[55] Emanuel von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Survival of the shortest: A retrospective analysis of influencing factors on password composition. In *Proceedings of the International Conference on Human-Computer Interaction*, 2013.

[56] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Proceedings of the Symposium on Usable Privacy and Security*, 2016.

[57] Daniel Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *Proceedings of the USENIX Security Symposium*, 2016.

[58] Victoria Woollaston. Facebook and netflix reset passwords after data breaches. *http://www.wired.co.uk/article/facebook-netflix-password-reset*, 2016.

A Anonymity Sets

In this section, we describe properties of anonymity sets (from Section 2.3) in more detail. Recall that anonymity sets are large sets of user credentials that provide plausible deniability about client data even if information about their membership in this set is revealed. Defining and arguing with anonymity sets is challenging and must be done carefully so as to avoid some trivialities. To avoid constructions with vacuous security, we require anonymity sets to have the following properties.

Large marginal supports: Our anonymity sets containing tuples (u,p) must additionally have sufficiently large marginal supports over both usernames and passwords. This ensures that despite there being several possible tuples (u,p), there is sufficiently large ambiguity about whether membership implies a specific username or password. A trivial anonymity set, for example, might have several possible credentials with different passwords all tied to the same username.

More mathematically, given an anonymity set K of size $|\mathsf{K}|$, we require that the size of the following sets:

$$\begin{aligned}\mathsf{SuppUser}(\mathsf{K}) &:= \{\mathsf{u} :\ (\mathsf{u},\mathsf{p}) \in \mathsf{K}\}\,,\\ \mathsf{SuppPwd}(\mathsf{K}) &:= \{\mathsf{p} :\ (\mathsf{u},\mathsf{p}) \in \mathsf{K}\}\,,\end{aligned}$$

both have large cardinalities comparable to that of $|\mathsf{K}|$. Observe that $|\mathsf{SuppUser}(\mathsf{K})|$ indicates how many bits of information about the username is leaked (smaller sets narrow the set of possible users and leak a lot of information). This is similarly true for $|\mathsf{SuppPwd}(\mathsf{K})|$ for passwords.

Hashing both usernames and passwords with cryptographically strong hash functions satisfies these requirements. In fact, it is possible that both sets have cardinalities as large $|\mathsf{K}|$ itself which would imply that for every possible common password there might be a username such that $(\mathsf{u},\mathsf{p}) \in \mathsf{K}$. This is true in our scheme modeling Argon2 as a random oracle.

Schemes that only hash passwords, as noted previously in Section 2.4, might still satisfy a weaker anonymity property. They hide usernames, but depending on how they truncate hashes, password-only schemes might allow for small or large $\mathsf{SuppPwd}$ sets. Thus, they might satisfy these requirements, but only weakly at least with respect to passwords. Furthermore, it is not true that for every password there is a username which might be part of the client's credentials, unlike our scheme. We also note that our trivial example, of having several passwords all tied to the same username, violates our requirement by having $|\mathsf{SuppUser}(\mathsf{K})| = 1$.

Uniformity requirement: This is a more challenging requirement to model mathematically. Intuitively, however, it states that anonymity sets should partition the space of usernames and passwords in a somewhat uniform manner. In other words, over random choices of the system parameters, it should be equally likely for any (u, p) to end up in any anonymity set. A trivial anonymity set violating this requirement would, as an example, only truncate usernames, thereby trivially leaking some information about the username. Truncation does not make it equally likely that any (u, p) can end up in any anonymity set.

Under the reasonable assumption that our hash function is independent of the domain of typical usernames and passwords and does not have any "weak inputs"—domains of inputs where it does not behave like an ideal hash function—this condition is easily satisfied. It is highly improbable that related credentials such as `username`, `username0`, `username123`, will all end up in the same anonymity set.

B Security of the Hash-and-Blind operation

In this section, we outline the security properties satisfied by the hash-and-blind operation, which is an important part of our protocol. Consider a keyed function $F(k,x) := \mathsf{H}(x)^k$ where $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}$ is a hash function mapping strings to a group element. In our construction, H is Argon2, and $\mathbb{G}$ is the elliptic curve $\mathsf{NID\_secp224r1}$.

The work of Jarecki et al. [27] shows that $F(k,\cdot)$ implements an *oblivious* pseudorandom function in the random oracle model assuming the hardness of the decision Diffie-Hellman assumption in $\mathbb{G}$. In this section, we do not elaborate on the details of the proof, but we state what is meant by a pseudorandom function and how $F(k,\cdot)$ can be evaluated obliviously—without the secret key k holder knowing which input they're evaluating on. Pseudorandomness helps us achieve bounded leakage and protects the credentials not queried by the user; obliviousness enables us to implement the Diffie-Hellman blinding based private set intersection within our protocol.

Pseudorandomness: Informally, a function $F(k,\cdot)$ with outputs in $\mathbb{Y}$ is said to be pseudorandom if the function behaves like a random function when evaluated on new inputs. More formally, given outputs $F(k,x_1),\ldots,F(k,x_Q)$ for Q queries $x_1,\ldots,x_Q$ of an adversary's choice, for any other $x' \notin \{x_1,\ldots,x_Q\}$, we require that $F(k,x')$ be computationally indistinguishable from a random element in $\mathbb{Y}$ as long as k is chosen uniformly at random and remains hidden.

When applied to our construction, it implies that a client that sees several possible $\mathsf{H}(\mathsf{u}_i,\mathsf{p}_i)^b$ still cannot distinguish $\mathsf{H}(\mathsf{u}',\mathsf{p}')^b$ from a random element in $\mathbb{G}$ if b is hidden. Hash-and-blind therefore protects the contents of the server database when interacting with clients. For the sake of completeness, we add that this protocol is only secure against honest-but-curious adversaries which assumes that a client might be curious to learn more than it is allowed to, but chooses to honestly follow the protocol.

Obliviousness: A function is said to be evaluated in an oblivious manner if there is a protocol between a client holding an input x and a server holding a function f such that at the end of the protocol, the client learns $f(x)$ and the server learns nothing. In our construction, $f(x) = \mathsf{H}(x)^b$ for some value b. The protocol between the client and server is fairly straightforward (and somewhat implicit in our construction): the client chooses a uniform random value a, sends $\mathsf{H}(x)^a$, receives $\mathsf{H}(x)^{ab}$, and reconstructs

$$f(x) = \left(\mathsf{H}(x)^{ab}\right)^{1/a} = \mathsf{H}(x)^b.$$

Correctness is fairly straightforward. To see why this is oblivious, observe that for any two inputs x_1 and x_2, the distributions $\mathsf{H}(x_1)^r$ and $\mathsf{H}(x_2)^s$ for uniformly drawn values r and s are identical. This implies that the server learns nothing about the client's input x.

When applied to our construction, it states that the component $\mathsf{H}(\mathsf{u},\mathsf{p})^a$ computed in Algorithm 2 allows the client to obliviously evaluate the PRF without revealing to the server information about the credential (u,p).

We end this section with a couple of notes. First, a caveat noting that some information about (u,p) does end up being leaked via the anonymity set, which we capture through our notion of leakage. A direct composition of proofs of security involving anonymity sets and obliviousness might be tricky and will require careful work. Deriving keys from these PRF outputs will additionally require careful applications of KDFs with the right domain separators to avoid re-use of crypto components.

Second, Everspaugh et al. [14] propose an OPRF service that is closely related to our construction here. Our requirements out of an OPRF differs on a couple of key points which does not enable us to use such a service directly: 1) we do not require the notion of *partial* obliviousness in their construction which adds significant computational overheads to their service, and 2) our clients use of an OPRF does not require an immediate evaluation of the PRF, but rather its application to a database to obliviously evaluate its inputs and return potential matches.