How to End Password Reuse on the Web

Ke Coby Wang
University of North Carolina at Chapel Hill
kwang@cs.unc.edu

Michael K. Reiter
University of North Carolina at Chapel Hill
reiter@cs.unc.edu

***Abstract*—We present a framework by which websites can coordinate to make it difficult for users to set similar passwords at these websites, in an effort to break the culture of password reuse on the web today. Though the design of such a framework is fraught with risks to users' security and privacy, we show that these risks can be effectively mitigated through careful scoping of the goals for such a framework and through principled design. At the core of our framework is a private set-membership-test protocol that enables one website to determine, upon a user setting a password for use at it, whether that user has already set a similar password at another participating website, but with neither side disclosing to the other the password(s) it employs in the protocol. Our framework then layers over this protocol a collection of techniques to mitigate the leakage necessitated by such a test. We verify via probabilistic model checking that these techniques are effective in maintaining account security, and since these mechanisms are consistent with common user experience today, our framework should be unobtrusive to users who do not reuse similar passwords across websites (e.g., due to having adopted a password manager). Through a working implementation of our framework and optimization of its parameters based on insights of how passwords tend to be reused, we show that our design can meet the scalability challenges facing such a service.**

I. Introduction

> *The reuse of passwords is the No. 1 cause of harm on the internet.*
>
> Alex Stamos [12]
> Facebook CSO (Jun 2015–Aug 2018)

Password reuse across websites remains a dire problem despite widespread advice for users to avoid it. Numerous studies over the past fifteen years indicate that a large majority of users set the same or similar passwords across different websites (e.g., [8], [59], [65], [14], [39], [54], [70]). As such, a breach of a password database or a phish of a user's password often leads to the compromise of user accounts on other websites. Such "credential-stuffing" attacks are a primary cause of account takeovers [74], [47], allowing the attacker to drain accounts of stored value, credit card numbers, and other personal information [47]. Ironically, stringent password requirements contribute to password reuse, as users reuse strong passwords across websites to cope with the cognitive burden of creating and remembering them [73]. Moreover, notifications to accounts at risk due to password reuse seem insufficient to cause their owners to stop reusing passwords [35].

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23360
www.ndss-symposium.org

It is tempting to view password reuse as inflicting costs on only users who practice it. However, preventing, detecting, and cleaning up compromised accounts and the value thus stolen is a significant cost for service providers, as well. A recent Ponemon survey [57] of 569 IT security practitioners estimated that credential-stuffing attacks incur costs in terms of application downtime, loss of customers, and involvement of IT security that average $1.7 million, $2.7 million and $1.6 million, respectively, per organization per year. Some companies go so far as to purchase compromised credentials on the black market to find their vulnerable accounts proactively (e.g., [12]). Companies also must develop new technologies to identify overtaken accounts based on their use [12]. Even the sheer volume of credential-stuffing attacks is increasingly a challenge; e.g., in November 2017, 43% (3.6 out of 8.3 billion) of all login attempts served by Akamai involved credential abuse [3]. Finally, the aforementioned Ponemon survey estimated the fraud perpetrated using overtaken accounts could incur average losses of up to $54 million per organization surveyed [57]. As such, interfering with password reuse would not only better protect users, but would also reduce the considerable costs of credential abuse incurred by websites.

Here we thus explore a technical mechanism to interfere with password reuse across websites. Forcing a user to authenticate to each website using a site-generated password (e.g., [49]) would accomplish this goal. However, we seek to retain the same degree of user autonomy regarding her selection of passwords as she has today—subject to the constraint that she not reuse them—to accommodate her preferences regarding the importance of the account, the ease of entering its password on various devices, etc. At a high level, the framework we develop enables a website at which a user is setting a password, here called a *requester*, to ask of other websites, here called *responders*, whether the user has set a similar password at any of them. A positive answer can then be used by the requester to ask the user to select a different password. As we will argue in Sec. III, enlisting a surprisingly small number of major websites in our framework could substantially weaken the culture of password reuse.

We are under no illusions that our design, if deployed, will elicit anything but contempt (at least temporarily) from users who reuse passwords across websites. Its usability implications are thus not unlike increasingly stringent password requirements, to which users have nevertheless resigned. However, options for password managers are plentiful and growing, with a variety of trustworthiness, usability, and cost properties (e.g., [62], [28]). Indeed, experts often list the use of a password manager that supports a different password per website to be one of the best things a user can do to reduce her online risk [39]. While there might be users who, despite having a

rich online presence, cannot use a password manager for some reason, we expect them to be few. Of course, nearly anyone capable of using a computer should be able to write down her passwords, as a last resort. Though historically maligned, the practice is now more widely accepted, exactly because it makes it easier to not reuse passwords (e.g., [46], [37]).

There are many technical issues that need to be addressed to make a framework like the one we propose palatable. First, such a framework should not reduce the security of user accounts. Second, the framework should also not decay user privacy substantially, in the sense of divulging the websites at which a user has an account. Third, it is important that the protocol run between a requester and responders should scale well enough to ensure that it does not impose too much delay for setting a password at a website.

Our framework addresses these challenges as follows. To minimize risk to user accounts, we design a protocol that enables the requester to learn if a password chosen by a user is similar to one she set at a responder; neither side learns the password(s) the other input to the protocol, however, even by misbehaving. Our framework leverages this protocol, together with other mechanisms to compensate for leakage necessitated by the protocol's output, to ensure that account security and privacy are not diminished. Among other properties, this framework ensures that the responders remain hidden from the requester and vice-versa. We verify using probabilistic model checking that the success rate of account takeover attempts is not materially changed by our framework for users who employ distinct passwords across websites. Scalability is met in our framework by carefully designing it to involve only a single round of interaction between the requester and responders. And, using observations about password reuse habits, we optimize our framework to detect similar password use with near-certainty while maximizing its scalability.

To summarize, our contributions are as follows:

- We initiate debate on the merits of interfering with password reuse on the web, through coordination among websites. Our goal in doing so is 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. In particular, we show that apparent obstacles to a framework for interfering with password reuse can be overcome through careful scoping of its goals and through reasonable assumptions (Sec. III).
- We propose a protocol for privately testing set membership that underlies our proposed framework (Sec. IV). We prove security of our protocol in the case of a malicious requester and against malicious responders.
- We embed this protocol within a framework to facilitate requester-responder interactions while hiding the identities of protocol participants and addressing risks that cannot be addressed by—and indeed, that are necessitated by—the private set-membership-test protocol (Sec. V). We demonstrate using probabilistic model checking that our framework does not materially weaken account security against password guessing attacks.
- We evaluate implementations of our proposed framework with differing degrees of trust placed in it (Sec. VI). Using password-reuse tendencies, we illustrate how to configure our framework to minimize its costs while ensuring detection of reused passwords with high likelihood. Finally, we demonstrate its scalability through experiments with a working implementation in deployments that capture its performance in realistic scenarios.

II. Related Work

We are aware of no prior work to enable websites to interfere with password reuse by the same user. Instead, server-side approaches to mitigate risks due to password reuse have set somewhat different goals.

Web single sign-on (SSO): SSO schemes such as OAuth (https://oauth.net/), OpenID (http://openid.net), OpenID Connect (http://openid.net/connect/), and Facebook Login (https://developers.facebook.com/docs/facebook-login/), enable one website (an "identity provider") to share a user's account information with other websites ("relying parties"), typically in lieu of the user creating distinct accounts at those relying parties. As such, this approach mitigates password reuse by simply not having the user set passwords at the relying parties. While convenient, SSO exposes users to a range of new attacks, leading some to conclude "the pervasiveness of SSO has created an exploitable ecosystem" [34]. In addition, the identity provider in these schemes typically learns the relying parties visited by the user [21].

Detecting use of leaked passwords by legitimate users: As mentioned in Sec. I, some companies cross-reference account passwords against known-leaked passwords, either as a service to others (e.g., https://www.passwordping.com, https://haveibeenpwned.com) or for their own users (e.g., [12]). While recommended [36], this approach can detect only passwords that are *known* to have been leaked. Because password database compromises often go undiscovered for long periods (as of 2017, 15 months on average [64]), this approach cannot identify vulnerable accounts in the interim.

Detecting leaked passwords by their use in attacks: Various techniques exist to detect leaked passwords by their attempted use, e.g., honey accounts [18] and honey passwords [6], [40], [27], the latter of which we will leverage as well (Sec. V-A1). Alone, these methods do little to detect an attacker's use of a leaked, known-good password for one website at another website where the victim user is known to have an account. Defending against such discriminating attacks would seem to require the victim's use of different passwords at distinct websites, which we seek to compel here.

Detecting popular passwords: Schechter et al. [63] proposed a service at which sites can check whether a password chosen by a user is popular with other users or, more specifically, if its frequency of use exceeds a specified threshold. Our goals here are different—we seek to detect the use of similar passwords by the same user at different sites, regardless of popularity.

Limiting password-based access: Takada [68] proposed to interfere with the misuse of accounts with shared passwords by adding an "availability control" to password authentication. In this design, a user disables the ability to log into her website account at a third-party service and then re-enables it when needed. This approach requires that the attacker be unable to itself enable login, and so requires an additional authentication at the third-party service to protect this enabling.

Website	Users (M)	Website	Users (M)
Facebook	2234	Sina Weibo	431
YouTube	1900	Outlook	400
WhatsApp	1500	Twitter	335
Wechat	1058	Reddit	330
Yahoo!	1000	Amazon	310
Instagram	1000	LinkedIn	303
QQ	803	Quora	300
iCloud	768	Baidu Tieba	300
Taobao	634	Snapchat	291
Douyin/TikTok	500	Pinterest	250

TABLE I: Estimates of active users for selected websites

III. Goals and Assumptions

In this section we seek to clarify the goals for our system and the assumptions on which our design rests.

A. Deployment Goals

It is important to recognize that in order to break the culture of password reuse, we do not require universal adoption of the framework we propose here. Instead, it may be enough to enlist a (surprisingly small) number of top websites. To see this, consider just the 20 websites listed in Table I.[1] For a back-of-the-envelope estimate, suppose that the users of each website in Table I are sampled uniformly at random from the 3.58 billion global Internet users.[2] Then, in expectation an Internet user would have accounts at more than four of them. As such, if just these websites adopted our framework, it would force a large fraction of users to manage five or more dissimilar passwords, which is already at the limit of what users are capable of managing themselves: "If multiple passwords cannot be avoided, four or five is the maximum for unrelated, regularly used passwords that users can be expected to cope with" [2]. We thus believe that enlisting these 20 websites could already dramatically improve password-manager adoption, and it is conceivable that with modest additional adoption (e.g., the top 50 most popular websites), password reuse could largely be brought to an end.

A user might continue using similar passwords across sites that do not participate in our framework. Each such reused password may also be similar to one she set at a site that *does* participate in our framework, but likely at only *one* such site. If this reused password is compromised at a non-participating site (e.g., due to a site breach), then the attacker might still use this password in a credential-stuffing attack against the user's accounts at participating sites, as it could today. Again, however, due to our framework, this attack should succeed at only one participating site, not many. Importantly, our framework restricts the attacker from posing queries about the user's accounts as a requester unless it gains the user's consent to do so (see Sec. V-A1). Even if it tricked the user into consenting, it could use such a query to confirm that the compromised password is similar to one set by the same user at *some* participating site, but not *which* site (see Sec. III-C). More generally, in Sec. V-B, we will show quantitatively that our framework offers little advantage to an attacker that can pose a limited number of queries as a requester.

B. User Identifiers

An assumption of our framework is that there is an identifier for a user's accounts that is common across websites. An email address for the user would be a natural such identifier, and as we will describe in Sec. V-A1, this has other uses in our context, as well. Due to this assumption, however, a user could reuse the same password across different websites, despite our framework, if she registers a different email address at each.

Several methods exist for a user to amass many distinct email addresses, but we believe they will interfere little with our goals here. First, some email providers support multiple addresses for a single account. For example, one Gmail account can have arbitrarily many addresses, since Gmail addresses are insensitive to capitalization, insertion of periods ('.'), or insertion of a plus ('+') followed by any string, anywhere before '@gmail.com'. As another example, 33mail (https://33mail.com) allows a user to receive mail sent to <*alias*>@<*username*>.33mail.com for any alias string. Though these providers enable a user to provide a distinct email address to each website (e.g., [52]), our framework could nevertheless extract a canonical identifier for each user. For Gmail, the canonical identifier could be obtained by normalizing capitalization and by eliminating periods and anything between '+' and '@gmail.com'. For 33mail, you@<*username*>.33mail.com should suffice. Admittedly this requires customization specific to each such provider domain, though this customization is simple.

Second, some hosting services permit a customer to register a domain name and then support many email aliases for it (e.g., <*alias*>@<*domain*>.com). For example, Google Domains (http://domains.google) supports 100 email aliases per domain. Since these domains are custom, it might not be tractable to introduce domain-specific customizations as above. However, registering one's own domain as a workaround to keep using the same password across websites presumably saves the user little effort or money (registering domains is not free) in comparison to just switching to a password manager. Going further, a user could manually register numerous email accounts at free providers such as Gmail. Again, this is presumably at least as much effort as alternatives that involve no password reuse. As such, we do not concern ourselves with such methods of avoiding password reuse detection.

This discussion highlights an important clarification regarding our goals: we seek to eliminate *easy* methods of reusing passwords but not ones that require similar or greater effort from the user than more secure alternatives, of which we take a password manager as an exemplar. That is, we do not seek to make it *impossible* for a user to reuse passwords, but rather to make reusing passwords about as difficult as not reusing them. We expect that even this modest goal, if achieved, will largely eliminate password reuse, since passwords are reused today almost entirely for convenience.

[1] User counts were retrieved on December 4, 2018 from https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/, https://www.statista.com/statistics/476196/number-of-active-amazon-customer-accounts-quarter/, http://blog.shuttlecloud.com/the-most-popular-email-providers-in-the-u-s-a/, and https://expandedramblings.com/index.php/{yahoo-statistics/, taobao-statistics/, quora-statistics/}.

[2] Estimate of Internet users was retrieved from https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/ on December 4, 2018.

C. Security and Privacy Goals

The goals we take as more absolute have to do with the privacy of users and the security of their accounts. Specifically, we seek to ensure the following:

- **account location privacy**: Websites do not learn the identities of other websites at which a user has an account.
- **account security**: Our framework strengthens security of user accounts at a site that participates in our framework, by interfering with reuse of similar passwords at other participating sites. Moreover, it does not qualitatively degrade user account security in other ways.

As we will see, **account security** is difficult to achieve, since our framework must expose whether responders' passwords are similar to the one chosen by the user at the requester. However, **account location privacy** hides from the requester each responder from which the requester learns this information. As such, if a user attempts to set the same password at a malicious requester that she has also set at some responder, or if a malicious requester otherwise obtains this password (e.g., obtaining it in a breach of a non-participating site), the malicious requester must still attempt to use that password blindly at participating websites, just as in a credential-stuffing attack today. (The attacker might succeed, but it would succeed without our framework, too.) Moreover, in Sec. V we will detail additional defenses against this leakage to further reduce the risk of attacks from malicious requesters to whom the user does not volunteer this password, and show using formal verification that these defenses are effective.

We conclude this section by summarizing some potential goals that we (mostly) omit from consideration in this paper. From a privacy perspective, we try to hide neither *when* a password is being set at some requester for an account identifier nor the *number* of responders at which an account has been established using that account identifier, simply because we are unaware of common scenarios in which these leakages would have significant practical ramifications. And, while we strive to guarantee **account location privacy** and **account security** even against a requester and responders that misbehave, we generally do not seek to otherwise detect that misbehavior. So, for example, each requester and responder has complete autonomy in determining the passwords that it provides to the protocol as the candidate password submitted by the user and the passwords similar to the one for the account with the same identifier, respectively. As we will see in Sec. VII, such misbehaviors can give rise to denial-of-service opportunities, for which we propose remedies there.

IV. Privately Testing Set Membership

A building block of our framework is a protocol by which a requester R can inquire with a responder S as to whether a password π chosen at R for an account identifier is similar to one already in use at S for the same identifier. If for an account identifier a, the responder S has a set $P(a)$ of passwords similar to that already set at S, then the goal of this protocol is for the requester to learn whether the candidate password π is in $P(a)$. However, any additional information leakage to the requester (about any passwords in $P(a)$ or even the number of passwords in $P(a)$) or to the responder (about π) should be minimized.

This general specification can be met with a private set-membership-test (PMT) protocol. Though several such protocols exist (e.g., [53], [50], [69], [58]), we develop a new one here with an interaction pattern and threat model that is better suited for our framework. In particular, existing protocols require special hardware [69] or more rounds of interaction [53], [50], or leak more information in our threat model [53], [50], [58] than the one we present.

In designing this protocol, we sought guidance from the considerable literature on private set intersection (PSI), surveyed recently by Pinkas et al. [56]. Informally, PSI protocols allow two parties to jointly compute the intersection of the sets that each inputs to the protocol, and ideally nothing else. Furthermore, PSI protocols secure in the malicious adversary model, where one party deviates arbitrarily from the protocol, have been proposed (e.g., [17], [13], [31], [32], [41], [45], [60], [61]). Still, while a PSI protocol would allow R to determine whether $\pi \in P(a)$, without additional defenses it could reveal too much information; e.g., if R input multiple passwords to the protocol, then it would learn which of these passwords were in $P(a)$. Moreover, as Tamrakar et al. [69] argue, PSI protocols are not ideal for implementing PMT due to their high communication complexity and poor scalability.

By comparison, two-party private set-intersection *cardinality* (PSI-CA) protocols are closer to our needs; these protocols output the size of the intersection of each party's input set, and ideally nothing else (e.g., [15], [16], [19], [25], [43]). As with PSI protocols, however, using a PSI-CA protocol without modification to implement PMT would reveal too much information if R input multiple passwords to the protocol. As such, our protocol here is an adaptation of a PSI-CA protocol due to Egert et al. [25, Section 4.4], in which we (i) reduce the information it conveys to only the results of a membership test, versus the cardinality of a set intersection, and (ii) analyze its privacy properties in the face of malicious behavior by a requester or responder (versus only an honest-but-curious participant in their work), accounting for leakage intrinsic in the application for which we use it here.

A. Partially Homomorphic Encryption

Our protocol builds upon a multiplicatively homomorphic encryption scheme $\mathcal{E} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \times_{[\cdot]} \rangle$ with the following algorithms. Below, $z \xleftarrow{\$} Z$ denotes random selection from set Z and assignment to z, and $Y \overset{d}{=} Y'$ denotes that random variables Y and Y' are distributed identically.

- Gen is a randomized algorithm that on input 1^κ outputs a public-key/private-key pair $\langle pk, sk \rangle \leftarrow \mathsf{Gen}(1^\kappa)$. The value of pk uniquely determines a *plaintext space* $\mathbb{G}$ where $\langle \mathbb{G}, \times_{\mathbb{G}} \rangle$ denotes a multiplicative, cyclic group of order r with identity $1_{\mathbb{G}}$, and where r is a κ-bit prime. The randomized function $\$(\mathbb{G})$ returns a new, random $m \xleftarrow{\$} \mathbb{G}$. We let $\mathbb{Z}_r = \{0, \ldots, r-1\}$ and $\mathbb{Z}_r^* = \{1, \ldots, r-1\}$, as usual.
- Enc is a randomized algorithm that on input public key pk and plaintext $m \in \mathbb{G}$ produces a ciphertext $c \leftarrow \mathsf{Enc}_{pk}(m)$. Let $C_{pk}(m)$ denote the set of all ciphertexts that $\mathsf{Enc}_{pk}(m)$ produces with nonzero probability. Then, $C_{pk} = \bigcup_{m \in \mathbb{G}} C_{pk}(m)$ is the ciphertext space of the scheme with public key pk.

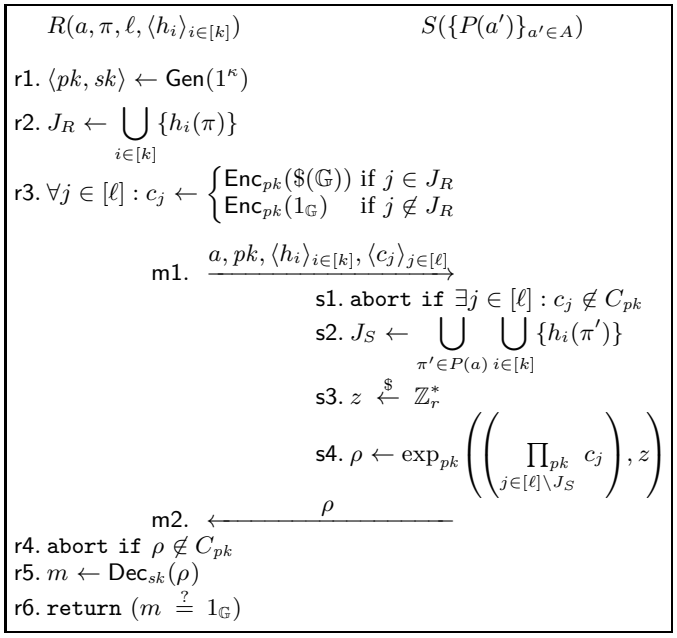


Fig. 1: PMT protocol; see Sec. IV-B. Requester R returns *true* if password π is similar to another password used at responder S for the same account identifier a, i.e., if $\pi \in P(a)$.

- Dec is a deterministic algorithm that on input a private key sk and ciphertext $c \in C_{pk}(m)$, for $m \in \mathbb{G}$ and pk the public key corresponding to sk, produces $m \leftarrow \mathsf{Dec}_{sk}(c)$. If $c \notin C_{pk}$, then $\mathsf{Dec}_{sk}(c)$ returns $\perp$.
- $\times_{[\cdot]}$ is a randomized algorithm that on input a public key pk and ciphertexts $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$ produces a ciphertext $c \leftarrow c_1 \times_{pk} c_2$ chosen uniformly at random from $C_{pk}(m_1 m_2)$. If $c_1 \notin C_{pk}$ or $c_2 \notin C_{pk}$, then $c_1 \times_{pk} c_2$ returns $\perp$. We use $\prod_{pk}$ and $\exp_{pk}$ to denote multiplication of a sequence and exponentiation using $\times_{pk}$, respectively, i.e.,

$$\prod_{i=1}^{z}{}_{pk}\ c_i \stackrel{d}{=} c_1 \times_{pk} c_2 \times_{pk} \ldots \times_{pk} c_z$$

$$\exp_{pk}(c, z) \stackrel{d}{=} \prod_{i=1}^{z}{}_{pk}\ c$$

B. Protocol Description

Our protocol is shown in Fig. 1, with the actions by the requester R listed on the left (lines r1–r6), those by the responder S listed on the right (s1–s4), and messages between them in the middle (m1–m2). In Fig. 1 and below, $[z]$ for integer $z > 0$ denotes the set $\{0, \ldots, z-1\}$.

At a conceptual level, our PMT protocol works as follows. The requester R takes as input an account identifier a, the user's chosen password π, a Bloom-filter [5] length ℓ, and the hash functions $\langle h_i\rangle_{i\in[k]}$ for the Bloom filter (i.e., each $h_i : \{0,1\}^* \to [\ell]$). R computes its Bloom filter containing π, specifically a set of indices $J_R \leftarrow \bigcup_{i\in[k]}\{h_i(\pi)\}$ (line r2). The responder S receives as input a set $P(a')$ of passwords similar to the password for each local account $a' \in A$ (i.e., A is its set of local account identifiers), and upon receiving message m1 computes its own ℓ-sized Bloom filter containing $P(a)$, i.e., indices $J_S \leftarrow \bigcup_{\pi'\in P(a)}\bigcup_{i\in[k]}\{h_i(\pi')\}$ (line s2).[3] The protocol should return *true* to R if $\pi \in P(a)$, which for a Bloom filter is indicated by $J_R \subseteq J_S$ (with some risk of false positives, as will be discussed in Sec. VI-A2).

Our protocol equivalently returns a value to R that indicates whether $[\ell] \setminus J_S \subseteq [\ell] \setminus J_R$, where "\" denotes set difference, without exposing J_S to R or J_R to S. To do so, the requester R encodes J_R as ciphertexts $\langle c_j\rangle_{j\in[\ell]}$ where $c_j \in C_{pk}(1_\mathbb{G})$ if $j \in [\ell]\setminus J_R$ and $c_j \in C_{pk}(m)$ for a randomly chosen $m \xleftarrow{\$} \mathbb{G}$ if $j \in J_R$ (r3). In this way, when S computes ρ in line s4—i.e., by homomorphically multiplying c_j for each $j \in [\ell]\setminus J_S$ and then exponentiating by a random $z \xleftarrow{\$} \mathbb{Z}_r^*$ (s3)—ρ is in $C_{pk}(1_\mathbb{G})$ if $[\ell]\setminus J_S \subseteq [\ell]\setminus J_R$ and otherwise is almost certainly not in $C_{pk}(1_\mathbb{G})$. As such, R returns *true*, indicating that π is similar to the password set at S for account a, if and only if $\mathsf{Dec}_{sk}(\rho) = 1_\mathbb{G}$ (r5–r6).

It is important that both S and R check the validity of the ciphertexts they receive (lines s1 and r4, respectively). For S, implicit in this check is that pk is a valid public key (i.e., capable of being output by Gen). For our implementation described in Sec. VI-A, these checks are straightforward.

C. Security

We now reason about the security of the protocol of Fig. 1 against malicious requesters (Sec. IV-C1) and against malicious responders (Sec. IV-C2). More specifically, our focus in this section is properties that underlie **account security** as informally described in Sec. III; **account location privacy** will be discussed in Sec. V. Proofs for all propositions in this section can be found in our technical report [72].

1) Security against malicious requester: R learns nothing more from executing the protocol in Fig. 1 besides the result $m \stackrel{?}{=} 1_\mathbb{G}$ in line r6 because no other information is encoded in ρ if the responder follows the protocol (i.e., unconditional security). First, if $\rho \notin C_{pk}(1_\mathbb{G})$ then ρ is a ciphertext of any $m \in \mathbb{G} \setminus \{1_\mathbb{G}\}$ with equal probability:

Proposition 1. *If the responder follows the protocol, then* $\mathbb{P}\big(\rho \in C_{pk}(m) \;\big|\; \rho \notin C_{pk}(1_\mathbb{G})\big) = \frac{1}{r-1}$ *for any* $m \in \mathbb{G} \setminus \{1_\mathbb{G}\}$.

Second, if $\rho \in C_{pk}(m)$, it is uniformly distributed in $C_{pk}(m)$:

Proposition 2. *If the responder follows the protocol, then* $\mathbb{P}\big(\rho = c \;\big|\; \rho \in C_{pk}(m)\big) = \frac{1}{|C_{pk}(m)|}$ *for any* $m \in \mathbb{G}$ *and any* $c \in C_{pk}(m)$.

2) Security against malicious responder: The system of which the protocol in Fig. 1 is a component will typically leak the result of the protocol run to the responder. Specifically, if a run of the protocol is immediately followed by another run of the protocol, then this suggests that the protocol returned *true*, i.e., that $\pi \in P(a)$. We will discuss in Sec. V-A2 using extra, "decoy" protocol runs to obscure this leakage. However, for the purposes of this section, we will assume that the result of the protocol is leaked to the responder reliably.

[3] This assumes that all of $P(a)$ will "fit" in an ℓ-sized bloom filter. If not, S can use any subset of $P(a)$ it chooses of appropriate size. This will be discussed further in Sec. VI-A2.

The implications of this leakage to the requirements for the encryption scheme $\mathcal{E}$ are that the requester serves as an oracle for the responder to learn whether one ciphertext ρ of its choosing satisfies $\rho \in C_{pk}(1_{\mathbb{G}})$. The responder could potentially use this oracle to determine which of the ciphertexts $\langle c_j \rangle_{j\in[\ell]}$ that it receives in line m1 satisfy $c_j \in C_{pk}(1_{\mathbb{G}})$ and, in turn, gain information about the password π that the user is trying to set. Indeed, some leakage of this form is unavoidable; e.g., the responder could simply set $\rho = c_0$ and, in doing so, learn whether $c_0 \in C_{pk}(1_{\mathbb{G}})$. Similarly, the responder could set $\rho = c_0 \times_{pk} c_1$; if the protocol returns *true*, then the responder can conclude that both $c_0 \in C_{pk}(1_{\mathbb{G}})$ and $c_1 \in C_{pk}(1_{\mathbb{G}})$.

To capture this leakage and the properties of our protocol more formally, we define a responder-adversary B to be a pair $B = \langle B_1, B_2 \rangle$ of probabilistic algorithms. B_1 takes as input pk and $\langle c_j \rangle_{j\in[\ell]}$ and outputs a ciphertext ρ and a state ϕ.[4] B_2 is provided the oracle response (i.e., whether $\rho \in C_{pk}(1_{\mathbb{G}})$) and the state ϕ and then outputs a set $J_B \subseteq [\ell]$. B is said to *succeed* if $J_B = J_R$, where J_R is the set of indices the requester "set" in its Bloom filter by encrypting a random group element (line r3). More specifically, we define experiment $\mathbf{Expt}^S_{\mathcal{E}}(\langle B_1, B_2 \rangle)$ as follows:

$$
\begin{array}{l}
\texttt{Experiment } \mathbf{Expt}^S_{\mathcal{E}}(\langle B_1, B_2 \rangle): \\
\quad \langle pk, sk \rangle \leftarrow \mathsf{Gen}(1^\kappa) \\
\quad J_R \xleftarrow{\$} \{J \subseteq [\ell] \mid |J| = k\} \\
\quad \forall j \in [\ell] : c_j \leftarrow \begin{cases} \mathsf{Enc}_{pk}(\$(\mathbb{G})) & \text{if } j \in J_R \\ \mathsf{Enc}_{pk}(1_{\mathbb{G}}) & \text{if } j \notin J_R \end{cases} \\
\quad \langle \rho, \phi \rangle \leftarrow B_1\left(pk, \langle c_j \rangle_{j\in[\ell]}\right) \\
\quad J_B \leftarrow B_2\left(\phi, \left(\rho \stackrel{?}{\in} C_{pk}(1_{\mathbb{G}})\right)\right) \\
\quad \texttt{return } (J_B \stackrel{?}{=} J_R)
\end{array}
$$

Then, we analyze the security of our protocol against responder-adversaries B that run in time polynomial in κ by bounding $\mathbb{P}\left(\mathbf{Expt}^S_{\mathcal{E}}(B) = true\right)$.

ElGamal encryption: To prove security against a malicious responder, we instantiate the encryption scheme $\mathcal{E}$ as ElGamal encryption [26], which is implemented as follows.

- $\mathsf{Gen}(1^\kappa)$ returns a private key $sk = \langle u \rangle$ and public key $pk = \langle g, U \rangle$, where $u \xleftarrow{\$} \mathbb{Z}_r$, g is a generator of the (cyclic) group $\langle \mathbb{G}, \times_{\mathbb{G}} \rangle$, and $U \leftarrow g^u$. We leave it implicit that the public key pk and private key sk must include whatever other information is necessary to specify $\mathbb{G}$, e.g., the elliptic curve on which the members of $\mathbb{G}$ lie.
- $\mathsf{Enc}_{\langle g,U \rangle}(m)$ returns $\langle V, W \rangle$ where $V \leftarrow g^v$, $v \xleftarrow{\$} \mathbb{Z}_r$, and $W \leftarrow mU^v$.
- $\mathsf{Dec}_{\langle u \rangle}(\langle V, W \rangle)$ returns WV^{-u} if $\{V, W\} \subseteq \mathbb{G}$ and returns $\perp$ otherwise.
- $\prod^{z}_{i=1 \langle g,U \rangle} \langle V_i, W_i \rangle$ returns $\langle V_1 \ldots V_z g^y, W_1 \ldots W_z U^y \rangle$ for $y \xleftarrow{\$} \mathbb{Z}_r$ if each $\{V_i, W_i\} \subseteq \mathbb{G}$ and returns $\perp$ otherwise. $\langle V_1, W_1 \rangle \times_{\langle g,U \rangle} \langle V_2, W_2 \rangle$ is just the special case $z = 2$.

Generic group model: We prove the security of our protocol against a responder-adversary B in the generic group model as presented by Maurer [48]. The generic group model allows modeling of attacks in which the adversary B cannot exploit the representation of the group elements used in the cryptographic algorithm. For some problems, such as the discrete logarithm problem on general elliptic curves, generic attacks are currently the best known (though better algorithms exist for curves of particular forms, e.g., [22]). Somewhat like the random oracle model [9], the generic group model is idealized, and so even an algorithm proven secure in the generic group model can be instantiated with a specific group representation that renders it insecure. Still, and also like the random oracle model, it has been used to provide assurance for the security of designs in numerous previous works; e.g., see Koblitz and Menezes [44] for a discussion of this methodology and how its results should be interpreted.

A function $f : \mathbb{N} \to \mathbb{R}$ is said to be *negligible* if for any positive polynomial $\phi(\kappa)$, there is some κ_0 such that $f(\kappa) < \frac{1}{\phi(\kappa)}$ for all $\kappa > \kappa_0$. We denote such a function by $negl(\kappa)$.

Proposition 3. *If $\mathcal{E}$ is ElGamal encryption, then in the generic group model,*

$$\mathbb{P}\left(\mathbf{Expt}^S_{\mathcal{E}}(B) = true\right) \leq 2\binom{\ell}{k}^{-1} + negl(\kappa)$$

for any responder-adversary B that runs in time polynomial in κ.

Our proof of Prop. 3 (see [72]) depends on disclosing to B the result $\rho \stackrel{?}{\in} C_{pk}(1_{\mathbb{G}})$ for only a single ρ or, in other words, on the use of a new public key pk per run of the protocol in Fig. 1 (see line r1). Since for ElGamal, generating a new public key costs about the same as an encryption, reusing a public key saves at most only $1/(\ell+1)$ of the computational cost for R in the protocol, and so we have not prioritized evaluating the security of such an optimization.

Prop. 3 is tight, i.e., there is a generic responder-adversary that achieves its bound (to within a term negligible in κ). This adversary $B = \langle B_1, B_2 \rangle$ performs as follows: B_1 outputs, say, $\rho \leftarrow c_0$ and, upon learning $\rho \stackrel{?}{\in} C_{pk}(1_{\mathbb{G}})$, B_2 guesses J_B to be a k-element subset of $[\ell]$ where $0 \in J_B$ iff $\rho \notin C_{pk}(1_{\mathbb{G}})$. Once $\mathbb{G}$ is instantiated in practice, security rests on the *assumption* that no responder-adversary can do better, i.e., that given the decisional Diffie-Hellman (DDH) instances $\langle c_j \rangle_{j\in[\ell]}$ for public key $\langle g, U \rangle$, no adversary can create a single DDH instance ρ for which the answer enables it to solve the instances $\langle c_j \rangle_{j\in[\ell]}$ with probability better than that given in Prop. 3. Informally, Prop. 3 says that any such adversary would need to leverage the representation of $\mathbb{G}$ to do so.

V. Interfering with Password Reuse

In this section, we propose a password reuse detection framework based on the PMT protocol proposed in Sec. IV.

A. Design

Our password reuse detection framework enables a requester R to inquire with multiple responders as to whether the password π chosen by a user for the account at R with identifier a is similar to another password already set for a

[4] We elide the other values in message m1 from the input to B_1 only because they do not contribute the security of the protocol.

at some responder. The requester does so with the help of a *directory*, which is a (possibly replicated) server that provides a front-end to requesters for this purpose. The directory stores, per identifier a, a list of addresses (possibly pseudonymous addresses, as we will discuss below) of websites at which a has been used to set up an account. We stress that the directory does *not* handle or observe passwords in our framework.

The requesters and responders need not trust each other in our framework, and employ the protocol described in Sec. IV to interact via the directory. More specifically, a user of the requester R selects a password π for her account with identifier a, and submits π to R. R sends the message in line m1 of Fig. 1 to the directory, which it forwards to some subset of m responders, out of the M_a total registered as having accounts associated with a at the directory. (How it selects m is discussed in Sec. VI-C.) The response from responder S_i is denoted $\mathsf{m2}_i$ in Fig. 2. Once the directory collects these responses, it forwards them back to R, after permuting them randomly to prevent R from knowing which responder returned which result (see Sec. V-A1). R then processes each as in lines r4–r6; any of these results that are $true$ indicates that some responder that was queried has a password similar to π set for account a. If any are true, then the requester (presumably) rejects π and asks the user to select a different password (perhaps with guidance to help her choose one that is likely to not be used elsewhere).

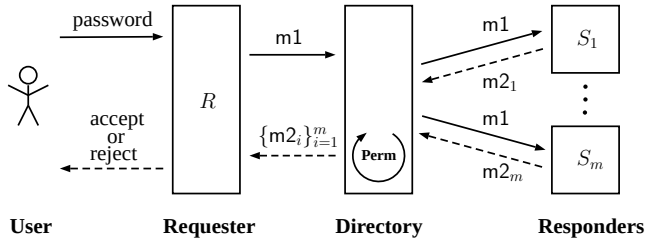


Fig. 2: Password reuse detection framework based on the PMT protocol introduced in Sec. IV.

There are some additional operations needed to support the framework, as well.

- *Directory entry addition*: After a new account is set up, the requester sends its address (discussed below) to the directory to be stored with the account identifier a.
- *Directory entry deletion*: When an account a on a web server (responder) is no longer used, the responder can optionally update the directory to remove the responder's address associated with a.
- *Password change*: When a user tries to change the password of an account, the web server should launch the protocol (as a requester) before the new password is accepted to replace the old.

The requester can communicate with the directory normally (e.g., using TLS over TCP/IP), trusting the directory to mask its identity from each responder to implement **account location privacy** (in which case, the directory behaves as an anonymizing proxy, cf., [7], [33]). Or, if the requester does not trust the directory to hide its identity, then it can communicate with the directory using an anonymizing protocol such as Tor (https://www.torproject.org/, [23]). Similarly, each responder address registered at the directory can be a regular TCP/IP endpoint, if it trusts the directory to hide its identity from others, or an anonymous server address such as a Tor hidden-service address [23] if it does not. In the latter case, the responder should register a distinct anonymous server address at the directory per account identifier a, to prevent identifying the responder by the number of accounts it hosts.

While each website could choose individually whether to trust the directory to hide its identity from others, we will evaluate the performance of our system only when either all websites trust the directory in this sense or none do. We refer to these models in the rest of the paper as the *TALP-directory* model (short for "trusted for **account location privacy**") and the *UALP-directory* model ("untrusted for **account location privacy**"), respectively. We believe that the TALP-directory model would be especially well-suited for deployment by a large cloud operator to serve its tenants, since these tenants already must trust the cloud operator.

Our framework is agnostic to the method by which each responder generates the set $P(a)$ of similar passwords for an account a. We envision it doing so by leveraging existing password guessers (e.g., [14], [70], [71], [75]), seeded with the actual password for the account. In addition, if, say, Google observes a user a set the password `google123`, it could add `twitter123` and `facebook123` to $P(a)$. So as to eliminate the need to store trivial variations of passwords in $P(a)$ and so reduce its size, the responder could reduce all such variants to a single canonical form, e.g., by translating all capital letters to lowercase, provided that requesters know to do the same.

1) Security for each responder: The responder need not retain the elements of $P(a)$ explicitly, but instead should store in $P(a)$ only the hash of each similar password, using a time-consuming cryptographic hash function H, making $P(a)$ more costly to exploit if the site is breached [66]. In particular, this hash function need not be the same as that used to hash the real password during normal login, and so can be considerably more time-consuming. In addition, H can be salted with a salt computed deterministically from user identifier a, so that the salts for a used at different sites are identical. Going further, the responder could proactively generate the set J_S when the password for a is set at S, and dispense of $P(a)$ altogether. However, this precomputation would require the Bloom filter size ℓ and hash functions $\langle h_i \rangle_{i \in [k]}$ to be fixed and known to the responder in advance.

Protecting J_S from disclosure: As shown in Sec. IV-C1, the only information leaked to the requester is the result of the protocol in Fig. 1, i.e., $\rho \stackrel{?}{\in} C_{pk}(1_{\mathbb{G}})$, regardless of the behavior of the requester (Props. 1–2). Still, however, this information can erode the security of J_S over multiple queries. For example, if a malicious requester sets $c_j \leftarrow \mathsf{Enc}_{pk}(m)$ where $m \neq 1_{\mathbb{G}}$ for one Bloom-filter index j and $c_{j'} \leftarrow \mathsf{Enc}_{pk}(1_{\mathbb{G}})$ for $j' \neq j$, then the result of $\rho \stackrel{?}{\in} C_{pk}(1_{\mathbb{G}})$ reveals whether $j \in J_S$. After ℓ such queries, the requester can learn the entirety of J_S and then search for the items stored in the Bloom filter offline.

Our framework mitigates this leakage using three mechanisms. First, each responder serves only PMT queries forwarded through the directory, i.e., by authenticating requests

as coming from the directory. This step is important for the following two mitigations to work.

Second, the directory randomly permutes the $\mathsf{m2}_i$ messages received from responders before returning them to the requester, thereby eliminating any indication (by timing or order) of the responder S_i from which each $\mathsf{m2}_i$ was returned. This largely eliminates the information that a malicious requester can glean from multiple PMT queries. In particular, the method above reveals nothing to the requester except the number of queried responders S_i for which $j \in J_{S_i}$, but not the responders for which this is true.

Third, we involve the user to restrict the number of PMT queries that any requester can make. Assuming a is an email address or can be associated with one at the directory, the directory emails the user upon being contacted by a requester, to confirm that she is trying to (re)set her password at that website.[5] This email could be presented to the user much like an account setup confirmation email today, containing a URL at the directory that user clicks to confirm her attempt to (re)set her password. The directory simply holds message $\mathsf{m1}$ until receiving this confirmation, discarding the message if it times out. (Presumably the requester website would alert the user to check her inbox for this confirmation email.) To avoid requiring the user to confirm multiple attempts to set a password at the requester and so multiple runs of the protocol in Fig. 1 (which should occur only if the user is still not using a password manager), the directory could allow one such confirmation to permit queries from this requester for a short window of time, at the risk of allowing a few extra queries if the requester is malicious. However, except during this window, requester queries will be dropped by the directory.

Leveraging the directory to permute PMT responses and limit PMT queries requires that we place trust in the directory to do so. If desired, this trust can be mitigated by replicating the directory using Byzantine fault-tolerance methods to overcome misbehavior by individual replicas. Ensuring that only user-approved PMTs are allowed can be implemented using classic BFT state-machine replication, for which increasingly practical frameworks exist (e.g., [4], [11]). Permuting PMT responses in a way that hides which S_i returned each $\mathsf{m2}_i$ even from f corrupt directory replicas can be achieved by simply having $f+1$ replicas permute and re-randomize the $\mathsf{m2}_i$ messages in sequence before returning them to the requester.

Limiting utility of a J_S disclosure: The risk that the adversary finds the password for user a at responder S, even with J_S, is small if the user leveraged a state-of-the-art password manager to generate a password that resists even an offline dictionary attack. Even if the user is not already using a password manager, obtaining the account password using this attack should again be expensive if the cryptographic hash function H is costly to compute. Moreover, the attacker can utilize a guessed account password only if it can determine the responder S at which it is set for account a, with which **account location privacy** interferes.

[5]The user can check that the confirmation email pertains to the site at which she is (re)setting her password if the site generates a nonce that it both displays to the user and passes to the directory to include in the confirmation email. The email should instruct the user to confirm this password (re)set only if the nonce displayed by the website matches that received in the email.

Still, to counter any remaining risk in case the attacker finds J_S, we advocate that S form its set $P(a)$ to include *honey passwords* [6], [40], [27]. That is, when the password is set (or reset) at a website for account a, the website chooses a collection of d honey passwords $\hat{\pi}^1, \ldots, \hat{\pi}^d$, as well, via a state-of-the-art method of doing so. It then generates a *cluster* of similar passwords for each of the $d+1$ passwords—we denote the cluster for the real password π by $\Psi(\pi)$ and the honey-password clusters by $\Psi(\hat{\pi}^1), \ldots, \Psi(\hat{\pi}^d)$—with each cluster being the same size ψ. Then, it sets the similar passwords for account a to be the union of these clusters, i.e., $P(a) = \Psi(\pi) \cup \left(\bigcup_{j=1}^{d} \Psi(\hat{\pi}^j)\right)$.

In this way, even if the attacker learns the entire contents of J_S for a responder S, the set J_S will contain at least $d+1$ passwords that appear to be roughly equally likely. If any password in a honey-password cluster is then used in an attempt to log into the account, the website can lock the account and force the user to reset her password after authenticating via a fallback method. The main cost of using honey passwords is a linear-in-d growth in the size of $P(a)$, which reduces the cluster size ψ that can be accommodated by the Bloom-filter size ℓ (which is determined by the requester). We will show in Sec. VI-C, however, that this cost has little impact on interfering with password reuse.

2) Security for the requester: Security for the requester is more straightforward, given Prop. 3 that proves the privacy of J_R against a malicious responder (and from the directory) in the generic group model. Moreover, the requester's identity is hidden from responders either by the directory (in the TALP-directory model) or because the requester contacts the directory anonymously (in the UALP-directory model).

As discussed in Sec. IV-C2 and accounted for in Prop. 3, responders (and the directory) learn the outcome of the protocol, since they see if the requester runs the protocol again. That is, a $true$ result will presumably cause the requester to reject the password and ask the user for another, with which it repeats the protocol. However, because the password is different in each run (which the requester should enforce), the information leaked to responders does not accumulate over these multiple runs. And, the responders learn only that *at least one* response resulted in $true$, not how many or which responders' did so.

Still, if the information leaked by the $false$ result for the password π finally accepted at the requester is of concern, it is possible to obfuscate even this information to an extent, at extra expense. To do so, the requester follows the acceptance of π with a number of "decoy" protocol runs (e.g., each using a randomly chosen J_R set of size k), as if the run on π had returned $true$. The user need not be delayed while each decoy run is conducted. That said, because decoy runs add overhead and because the responder is limited to learn information about π in only a single protocol run (and to learn a limited amount, per Prop. 3), we do not consider decoys further here.

B. Analysis via Probabilistic Model Checking

Probabilistic model checking is a formal method to analyze probabilistic behaviors in a system. In this section, we evaluate the security of our framework against a malicious requester using Storm, a probabilistic model checker [20].

Storm supports analysis of a Markov decision process (MDP), by which we model the attacker targeting a specific account a. That is, we specify the adversary as a set of *states* and possible *actions*. When in a state, the attacker can choose from among these actions nondeterministically; the chosen action determines a probability distribution on the state to which the attacker then transitions. These state transitions satisfy the *Markov property*: informally, the probability of next transitioning to a specific state depends only on the current state and the attacker's chosen action. Storm exhaustively searches all decisions an attacker can make to maximize the probability of the attacker succeeding in its goal. Here, we define this goal to be gaining access to account a on any responder, and so Storm calculates the probability of the attacker doing so under an optimal strategy.

As is common in formal treatments of password guessing (e.g., [42]), we parameterize the attacker with a password *dictionary* of a specified size, from which a's password π_i at each responder S_i is chosen independently and uniformly at random. The base-2 logarithm of this size represents the entropy of the password. We then vary the size of this dictionary to model the attacker's knowledge about a's password choices at responders. We further presume that the similar-password set $P_i(a)$ at each responder S_i is contained in this dictionary (or equivalently we reduce $P_i(a)$ to the subset that falls into the dictionary). For simplicity, we assume that the clusters $\Psi(\pi_i), \Psi(\hat{\pi}_i^1), \ldots, \Psi(\hat{\pi}_i^d)$ that comprise $P_i(a)$ are mutually disjoint and disjoint across responders; so, $|\bigcup_{i=1}^{m} P_i(a)| = \sum_{i=1}^{m} |P_i(a)| = m(d+1)\psi$ where ψ is the size of each cluster. Below, we denote by $Sim = \bigcup_{i=1}^{m} P_i(a)$ the union of all similar-password sets constructed by responders.

The attacker is limited by two parameters. First, we presume that each responder limits the number of consecutive failed logins per account before the account locks, as is typical and recommended (e.g., [36]); we call this number the *login budget* and denote it ζ. Second, our framework limits the PMT queries on a's accounts to those approved by user a when she is (re)setting her password (see Sec. V-A1); we model this restriction as a *PMT budget*. The login budget is per responder, whereas the PMT budget is a global constraint.

We also permit the adversary advantages that he might not have in practice. First, he knows the full set of responders, so he can attempt to log into any of them, and the login budget at each. Second, if he receives a positive response to a PMT query with password π', then the cluster containing π' becomes completely known to him. That is, if $\pi' \in \Psi(\pi_i)$ for the actual account-a password π_i at S_i, then $\Psi(\pi_i)$ is added to the adversary's set of identified clusters, and if $\pi' \in \Psi(\hat{\pi}_i^j)$ for a honey password $\hat{\pi}_i^j$ at S_i, then $\Psi(\hat{\pi}_i^j)$ is added to that set. Critically, however, he learns neither whether the new cluster is the cluster of a real password or a honey password, nor the responder S_i at which the cluster was chosen; both of these remain hidden in our design. Third, each failed login attempt at S_i provides the adversary complete information about the attempted password π', specifically if it is in a honey-password cluster ($\pi' \in \bigcup_{j=1}^{d} \Psi(\hat{\pi}_i^j)$) or simply incorrect ($\pi' \neq \pi_i$).

1) Model Description: A state in our model is defined to include the following items of information: previous adversary PMT queries and their results; the number of PMT queries that remain available to the attacker; the password clusters whose existence in Sim has been confirmed by the adversary via PMTs, to which we refer as the *confirmed* clusters; and per website, the previous adversary login attempts, their results, and the number of login queries remaining at that website.

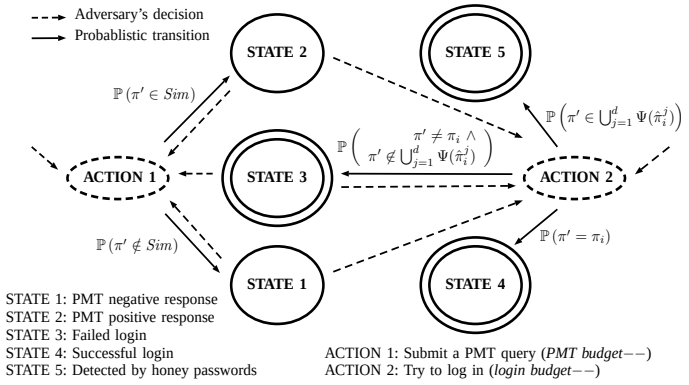


Fig. 3: Abstract MDP automaton for attacker interaction with S_i. π_i is the correct account password; $\hat{\pi}_i^1, \ldots, \hat{\pi}_i^d$ are its honey passwords; π' is an attacker's password guess. Probabilities are conditioned on attacker knowledge gained so far.

Fig. 3 shows an automaton that represents the attacker interacting with one website S_i. The entire model includes multiple such automata, one per website, and the adversary can switch among these automata at each step. Actions and states shown in Fig. 3 represent sets of actions and states in the actual automaton. For example, when the adversary tries to login by submitting a password to the login interface of the website, the password could be chosen from a "confirmed" cluster list or not, which is determined by the adversary. Though these are separate actions in our model, we let ACTION 2 serve as an abbreviation for all such actions in Fig. 3, to simplify the figure. Similarly, a state shown in Fig. 3 represents all states resulting from the same query response but that differ based on the state variables described above. Final states (for interacting with S_i) are indicated by double circles.

If the adversary enters STATE 5 for a website or uses up its login budget for a website, he must switch to another website to continue attacking. The adversary wins if he enters STATE 4 on any one of the websites, while he loses if he uses up the login budget or triggers account locked-down on all websites.

2) Results: The model-checking results, and in particular the impact of various parameters on those results, are summarized in Fig. 4. This figure plots the attacker's success probability, under an optimal strategy, as a function of the password entropy. The leftmost data point in each graph pertains to a dictionary size equal to $|Sim| = m(d+1)\psi$, which is the minimum dictionary size consistent with our model. This minimum dictionary size—representing a large amount of attacker knowledge about the dictionary from which the user chooses her password—is the reason why the attacker succeeds with such high probability. Each graph shows four curves, corresponding to PMT budgets of 0, 3, 6, and 9. The PMT budget of 0 provides a baseline curve that shows the security of each configuration in the absence of our design (though in the optimistic case where user a nevertheless chose different passwords at each website).

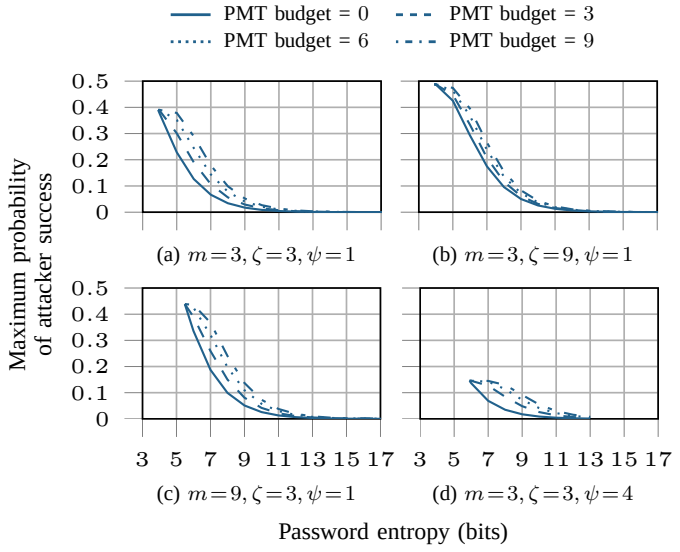


Fig. 4: Maximum probability with which attacker logs into account at some responder, as a function of password entropy. Subfigures show different settings for the number m of responders queried, the login budget ζ, and the cluster size ψ. The number of honey-password clusters is $d = 4$. All subfigures have the same axes.

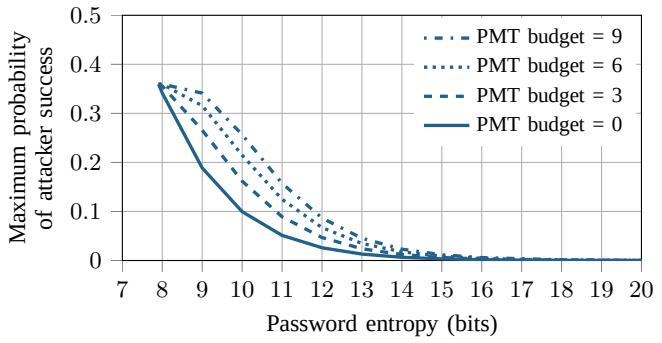


Fig. 5: Maximum probability with which attacker logs into account at some responder, as a function of password entropy, where $m = 12$, $\zeta = 9$, $\psi = 4$, and $d = 4$.

Fig. 4a shows a baseline with small parameters; the other subgraphs show the effect of increasing one parameter at a time. Fig. 4b and Fig. 4c show the impacts of increasing the per-website login budget ζ and the number m of responders queried in each PMT, respectively, which both increase the attacker's probability of success somewhat. Fig. 4d shows that increasing the size ψ of each password (including honey-password) cluster suppresses the success probability.

These graphs show that while growing the PMT budget increases the attacker's probability of success, the amount by which it does so is modest and diminishes as the password entropy grows. Fig. 5 shows somewhat more realistic parameters (though we were limited in growing these calculations to truly realistic sizes by the computational expense of doing so). As shown there, any attacker advantage gained by up to 9 PMT queries all but disappears with a dictionary of size only 2^{14}.

VI. Evaluation and Parameter Optimization

A. Implementation

We built a prototype of our framework to evaluate its performance and scalability, and to inform its parameterization (see Sec. VI-C). We realized the cryptographic parts of our protocol in C and other parts using Go.

1) Cryptography: We used the ElGamal cryptosystem in an elliptic-curve group (EC-ElGamal) as the multiplicatively homomorphic scheme $\mathcal{E}$ in Fig. 1. We realized all cryptographic operations using MIRACL (https://github.com/miracl/MIRACL). Our implementation includes four standardized elliptic curves: secp160r1, secp192r1 (NIST P-192), secp224r1 (NIST P-224) and secp256r1 (NIST P-256) [10], [29]. Elliptic-curve cryptosystems based on these curves can provide security roughly equivalent to RSA with key lengths of 1024, 1536, 2048 and 3072 bits, respectively. The generator g used with each curve has a cofactor of 1 [10], so that the group includes all curve points. This allows the requester and responders to check the validity of ciphertexts (i.e., lines s1 and r4 in Fig. 1) by checking if each ciphertext component is a valid point on the elliptic curve (or the point at infinity).

To make messages shorter and save bandwidth, we enable *point compression* in our implementation. Point compression (e.g., [38, Section A.9.6]) is a technique that compresses each elliptic-curve point to half its original size by using only $y \bmod 2$ in place of its y coordinate value. Correspondingly, *point decompression* reconstructs the point by recovering the y coordinate based on the x coordinate and $y \bmod 2$.

2) Bloom filters: A Bloom filter has a false positive rate of $\approx (1-e^{-k\cdot\frac{n}{\ell}})^k$ where $n = |P(a)|$ denotes the number elements to be inserted into the Bloom filter by the responder, ℓ denotes the length of the Bloom filter and k denotes the number of hash functions (e.g., see [51, pp. 109–110]). As such, the number of hash functions that minimizes false positives is $k_{\text{opt}} = \frac{\ell}{n} \cdot \ln 2$ and in this case, the minimized false positive rate is $2^{-k_{\text{opt}}} = 2^{-\frac{\ell}{n}\cdot\ln 2} \approx (0.6185)^{\frac{\ell}{n}}$. In our framework, k and ℓ are decided by the requester, while n is determined by each responder with the knowledge of k and ℓ received from the requester. In our implementation, the requester chooses $k = 20$ by default, and so each responder then generates a set $P(a)$ of size $n \leq \frac{\ell}{k} \cdot \ln 2$ to ensure a false positive rate of $\approx 2^{-20}$.

3) Precomputation: We use precomputation to optimize the creation of ciphertexts c_j by the requester in our protocol. Specifically, the requester precomputes private key u, public key U, and values $\{V_j\}_{j\in[\ell]}$ and $\{W_j\}_{j\in[\ell]}$, where each $\langle U, V_j, W_j\rangle$ is a valid Diffie-Hellman triple, i.e., $\langle V_j, W_j\rangle \in C_{\langle g,U\rangle}(1_{\mathbb{G}})$. To create a ciphertext c_j of a different group element $m \neq 1_{\mathbb{G}}$, the requester need only multiply W_j by m; thus, line r3 is completed in at most one multiplication per $j \in [\ell]$. In practice, this precomputation could begin once the user enters the account registration web page and continue during idle periods until a password is successfully set.

B. Response Time

In this section, we evaluate the response time of our prototype system as seen by the requester (and in the absence of any user interaction, such as that described in Sec. V-A1), with two goals in mind. First, we want to systematically measure

the effects of various parameter settings on our prototype implementation, to inform the selection of these parameters through an optimization process discussed in Sec. VI-C. We mainly explore two different parameters of our framework: The maximum number of similar passwords $n = |P(a)|$ per responder (as determined by setting the Bloom filter size $\ell = \lceil \frac{20n}{\ln 2} \rceil$ in the protocol), and the number m of responders. The second main goal of our experiments here is to compare the performance of our prototype with and without leveraging Tor for implementing **account location privacy**, i.e., the UALP-directory and TALP-directory models, respectively. In doing so, we hope to shed light on the performance costs of adopting a more pessimistic trust model in which the directory is not trusted to hide the websites where each account identifier a has been used to register an account.

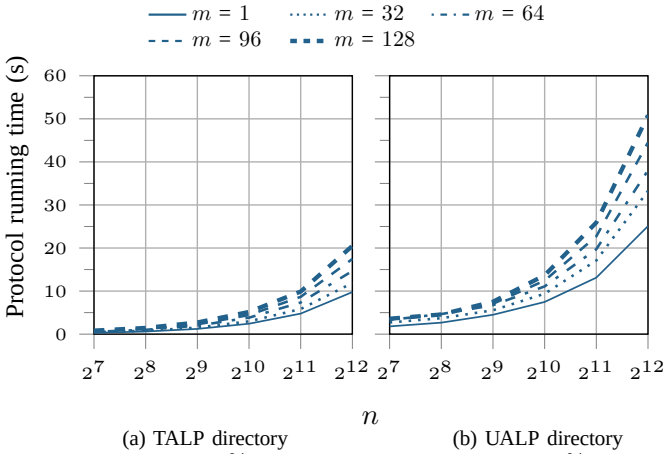
1) Experimental setup: In our evaluations, we set up one requester, one directory, and up to 128 responders, spread across six machines located in our department. The requester and the directory ran on separate machines with the same specification: 2.67GHz $\times$ 8 physical cores, 72GiB RAM, Ubuntu 14.04 x86_64. The (up to) 128 responders were split evenly across four other, identical machines: 2.3GHz $\times$ 32 physical cores with hyper-threading enabled (and so 64 logical cores), 128GiB RAM, Ubuntu 16.04 x86_64. Each of the responders sharing one machine was limited to two logical cores, and had its own exclusive data files, processes, and network sockets. The six machines were on the same $1\mathrm{Gb/s}$ network. Thus, while our results might be pessimistic due to resource sharing among responders, they might also be somewhat optimistic in our TALP-directory experiments due to leveraging LAN communication. (The UALP-directory case is discussed further below.)

Parameters were set to the following defaults unless otherwise specified: $m = 64$, and elliptic-curve key length of 192 bits. In particular, $m = 64$ is conservative based on recent studies. For example, a 2017 study with 154 participants found that users have a mean of 26.3 password-protected web accounts [54], which is quite consistent with other studies (e.g., [30], [67]).

Because the public Tor network is badly under-provisioned for its level of use and so its performance varies significantly over time, in our tests for the UALP-directory model, we utilized a private Tor network with nodes distributed across North America and Europe. Our private Tor network consisted of three Tor authorities, eight normal onion routers, and two special onion routers. The eight normal onion routers were running on eight different Amazon EC2 (m4.large) instances, one located in each of the eight Amazon AWS regions in North America and Europe. Among these onion routers, three were also running as Tor authorities, with one in Europe, one in U.S. West, and the other in U.S. East. Two special onion routers were running on the machine in our department hosting the directory; one ("Exit" in Fig. 6) exclusively served as the exit node of Tor circuits from requesters, and the other ("RP" in Fig. 6) served exclusively as the "rendezvous point" picked by the directory to communicate with Tor hidden services, i.e., the responders. As shown in Fig. 6, each circuit included two more onion routers ("OR" in Fig. 6) chosen at random from among the eight normal onion routers already described.

All datapoints reported in the graphs below are averaged from 50 executions. Relative standard deviations, denoted χ, are reported in figure captions.

Fig. 6: Topology of our UALP-directory experimental setup



(a) TALP directory ($\chi < 6\%$)

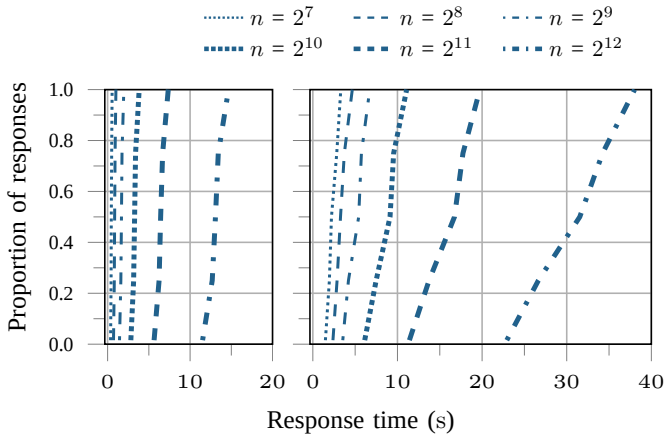
(b) UALP directory ($\chi < 28\%$)

Fig. 7: Response time for various n and m

2) Results: A measure of primary concern for our framework is the response time witnessed by the requester, since this delay will be imposed on the user experience while setting her password. Fig. 7a shows the response time in the TALP-directory model, where the requester connects directly to the directory and the directory connects directly with each responder. In contrast, Fig. 7b shows the response time in the UALP-directory model, and so connections are performed through Tor. Precomputation costs (see Sec. VI-A3) are not included in Fig. 7, as these costs are expected to be borne off the critical path of interacting with the user. Tor circuit setup times are amortized over the 50 runs contributing to each datapoint in Fig. 7b. In practice, we expect this setup cost to be similarly amortized over attempts needed by the user to choose an acceptable (not reused) password, or relegated to a precomputation stage when the user first accesses the requester's account creation/password reset page.

One observation from Fig. 7 is that the response-time cost of mistrusting the directory and so of relying on Tor to implement **account location privacy**, is typically $\geq 2\times$ for the parameters evaluated there. Recall that in Fig. 7b, both the requester–directory and directory–responder communications were routed through two onion routers chosen randomly from Amazon datacenter locations in North America and Europe (see Fig. 6), in contrast to LAN communication in Fig. 7a. The costs of these long-haul hops and Tor-specific processing increased as n grew, due to growth in query message size.

Fig. 7 also shows the impact of more responders (larger m) on the response time witnessed by the requester. The main underlying cause of this effect is the variance in the speeds with which the responders return responses to the directory.



(a) TALP directory ($\chi < 4\%$)

(b) UALP directory ($\chi < 33\%$)

Fig. 8: Response time (horizontal axis) when the proportion of responses on the vertical axis is returned by directory as soon as it is available ($m = 64$)

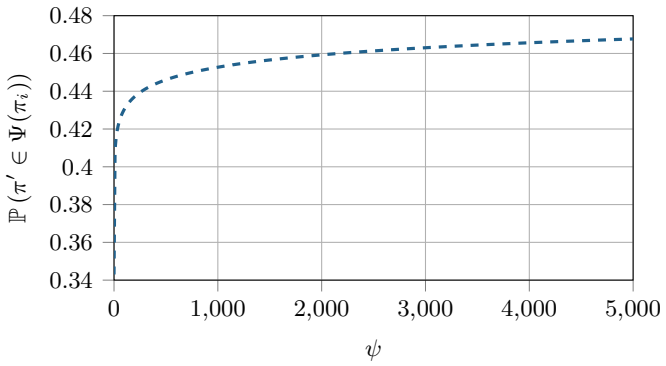


Fig. 9: Estimate of $\mathbb{P}(\pi' \in \Psi(\pi_i))$ for account-a password π_i at S_i and candidate password π' selected by user a at R, per cluster size $\psi = |\Psi(\pi_i)|$ and taken with respect to random selection of the user a and responder S_i; based on [70, Fig. 7]

This variance is small when communication is direct, but it grows substantially when Tor is used, due to the differences in routes taken between the directory and each responder.

These effects are also illustrated in Fig. 8, which shows the response time observed by the requester when the directory returned the proportion of $m = 64$ responses on the vertical axis as soon as that proportion was available to it. For example, Fig. 8b shows that when $n = 2^{10}$, if the directory waited for 75% of the responses (48 responses) before returning them to the requester, the requester observed an average response time of 9.55s (since $(9.55, 0.75)$ is a point on the $n = 2^{10}$ curve).

Recall that the directory forwards the *same* message m1 to all responders in our framework. In the UALP-directory model, using an anonymous communication system that exploits this one-to-many multicast pattern to gain efficiencies while still hiding the multicast recipients (e.g., [55]) could presumably reduce the delays before the directory receives responses, and their variance. We leave this extension to future work.

C. Parameter Optimization

At first glance, the results of Sec. VI-B are perhaps discouraging, since they suggest that the response time of testing with a large number n of similar passwords and at a large number m of queried responders is potentially large, especially in the UALP-directory model (Fig. 7b). In this section we describe an approach to select optimal parameters for use in our framework, specifically parameter values m and n that maximize the likelihood of detecting the use of a similar password, subject to a response-time goal. As we will see, the results are not discouraging at all—a high true detection rate can be achieved within reasonable response-time limits with a surprisingly small n and while querying a modest number m of responders from among the total number of responders M_a registered at the directory for account a.

The reason behind this initially surprising result is the typical manner in which people create new passwords by applying simple, predictable transforms to existing passwords. Numerous studies (e.g., [75], [70]) have found very low variation in the transforms that users leverage to modify their passwords (when they modify their passwords at all). Provided that responder S_i populates $\Psi(\pi_i) \subseteq P_i(a)$ (see Sec. V-A1) by applying these common transforms to its account-a password π_i, the probability that the user's chosen password π' at a requester is contained within $\Psi(\pi_i)$ at a randomly chosen responder S_i is approximately as shown in Fig. 9 (cf., [70, Fig. 7]), as a function of $\psi = |\Psi(\pi_i)|$.[6] As we can see, this probability is already substantial for very small ψ. For example, this probability is ≈ 0.34 for even $\psi = 1$; in other words, users on average employ the same password at $\approx 34\%$ of the websites where they have accounts. Moreover, this probability grows quickly as ψ is increased only slightly.

The key insight here is that if a user chooses its candidate password π' as users typically do, then using a large ψ provides little additional power (Fig. 9) but, since $n = (d+1)\psi$ where d is the number of honey-password clusters, imposes much greater cost (Fig. 7) than using a small ψ. Moreover, suppose we model the true detection rate when querying m randomly chosen responders[7] as $\mathsf{tdr} = 1 - (\mathbb{P}(\pi' \notin \Psi(\pi_i)))^m$, i.e., ignoring the probability of false detections due to the use of a Bloom filter and assuming that the events $\pi' \notin \Psi(\pi_i)$ and $\pi' \notin \Psi(\pi_{i'})$ are independent if $i \neq i'$ (which is perhaps reasonable since the user is forced to set dissimilar passwords at S_i and $S_{i'}$ by our framework). Then, increasing m provides more detection power.

To balance these parameters and the response time of the protocol, we model the response time using

$$t(m, n) = \beta_0 + \beta_1 \cdot n + \beta_2 \cdot m + \beta_3 \cdot n \cdot m$$

Regression analysis using the data in Sec. VI-B yields $\beta_0 = 1.5507$, $\beta_1 = 5.8834 \times 10^{-3}$, $\beta_2 = 2.6209 \times 10^{-3}$ and

[6] Fig. 9 is a log-normal CDF fitted to points selected from [70, Fig. 7] by manual inspection, as we could not obtain the source data for that figure.

[7] The directory should retain its same random choice of m responders across the user's failed attempts to select a password that she has not reused, lest she simply retry the same or a closely related password until a set of m responders at which it is not used is chosen. Alternatively, the requester can be charged with ensuring that the user's attempted passwords are sufficiently different from one another.

		t_{goal} (s)									
		.01	.02	.03	.04	.05	.06	.07	.08	.09	.10
$d=0$	n	1	1	2	2	5	9	13	16	20	23
	m	1	10	17	26	26	26	26	26	26	26
	tdr	.343	.985	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1
$d=4$	n	-	5	5	5	5	10	10	15	20	20
	m	-	1	10	19	26	24	26	26	26	26
	tdr	-	.343	.985	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1
$d=9$	n	-	-	-	10	10	10	10	10	20	20
	m	-	-	-	8	16	24	26	26	26	26
	tdr	-	-	-	.965	.999	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1

(a) TALP directory

		t_{goal} (s)									
		1.60	1.62	1.64	1.66	1.68	1.70	1.72	1.74	1.76	1.78
$d=0$	n	1	2	2	5	8	11	14	17	19	22
	m	16	21	26	26	26	26	26	26	26	26
	tdr	.999	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1
$d=4$	n	5	5	5	5	5	10	10	15	15	20
	m	6	13	20	26	26	26	26	26	26	26
	tdr	.920	.996	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1
$d=9$	n	-	10	10	10	10	10	10	10	20	20
	m	-	3	9	16	22	26	26	26	25	26
	tdr	-	.716	.977	.999	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1	$\approx$1

(b) UALP directory

TABLE II: Choices for m and n computed using optimization in Sec. VI-C with $M_a = 26$

$\beta_3 = 4.7135 \times 10^{-5}$ in the UALP-directory case (root-mean-square error $RMSE = 0.4547$) and $\beta_0 = 6.4595 \times 10^{-3}$, $\beta_1 = 2.2885 \times 10^{-3}$, $\beta_2 = 1.0271 \times 10^{-3}$ and $\beta_3 = 2.0336 \times 10^{-5}$ in the TALP-directory case ($RMSE = 0.1276$). Then, the requester chooses m and n using the following optimization:

$$\begin{aligned} \underset{m,n}{\text{maximize}} \;\; & \mathsf{tdr} = 1 - (\mathbb{P}(\pi' \notin \Psi(\pi_i)))^m \\ \text{subject to } & t(m,n) \leq t_{\text{goal}} \\ & 1 \leq \psi = n/(d+1) \\ & 1 \leq m \leq M_a \end{aligned}$$

where t_{goal} is the requester's desired response time and M_a is the number of responders registered at the directory as having an account for identifier a. The directory can send M_a to the requester in an initial negotiation round before message m1.

This optimization, together with using the curve in Fig. 9 to estimate $\mathbb{P}(\pi' \in \Psi(\pi_i))$ and the regression results above to estimate $t(m,n)$, yields results like those shown in Table II. In these optimizations, we set $M_a = 26$, because recent work found the mean number of password-protected online accounts per user is 26 [54]. The response-time goals t_{goal} used in Table II were chosen simply to show how the optimal m and n vary under stringent response-time constraints. As shown there, for many response-time goals t_{goal}, a true detection rate $\mathsf{tdr} \approx 1$ can be achieved with very small values of n.

As such, the full range of parameter settings explored in Sec. VI-B will rarely be needed. This is fortunate, since small values of n improve the throughput of requester-responder interactions, especially in the TALP-directory model. To see this, Table III shows the throughput of our implementation, measured as the largest number of *qualifying* responses achieved

	m					
n	1	6	11	16	21	26
1	4304	1013	492	325	237	174
10	2415	549	277	188	155	122
20	1478	336	182	129	98	78
30	1076	243	124	86	63	53
40	788	187	94	67	49	40
50	683	159	76	52	39	33
60	611	132	63	43	32	25

(a) TALP directory

	m					
n	1	6	11	16	21	26
1	95	61	42	33	27	22
10	87	59	40	31	25	20
20	78	54	37	28	23	19
30	71	51	35	27	20	16
40	62	44	32	24	18	14
50	53	39	26	20	15	11
60	42	31	20	16	10	10

(b) UALP directory

TABLE III: Maximum qualifying responses per second

as the requests per second were increased, as a function of n and m. In Table IIIa and Table IIIb, a response was *qualifying* if its response time was ≤ 5s and ≤ 8s, respectively.

This 3s difference between the standards for *qualifying* in the two tests was needed because we constructed the UALP-directory test to capture as faithfully as possible the Tor costs that a real deployment would incur. Notably, even though the m responders queried per request were chosen from only 64 responders in total (the configuration was the same as in Sec. VI-B), no two requests were allowed to use the same Tor circuit, since they would be unable to do so in a real deployment, where different addresses for the same responder are stored for different user accounts at the directory. (The exception is if the requests were for the same user and at the same responder.) So, each request necessitated construction of new Tor circuits to its responders, which increased response times commensurately.

To put Table III in context, a throughput of 50 qualifying responses per second is enough to enable each of the 312 million Internet users in the U.S.[8] to setup or change passwords on about 5 accounts per year. Moreover, we believe the numbers in Table III to be pessimistic, in that in each request, the m responders were chosen from only $M_a = 64$ responders in total, versus from likely many more in practice. Still, based on Table IIIb, a deployment using the UALP-directory model would presumably require adaptations of Tor for our use-case (e.g., [55]) and distribution of the directory.

We note, however, that even a non-replicated directory should easily handle the *storage* requirements of our design. With 3.58 billion active Internet users worldwide and an average of 26 password-protected accounts per user, the storage of a Tor hidden service address for each user account at each website amounts to only $\approx$ 1.5TB of state. In the TALP-directory model, the storage requirements would be even less.

VII. Denials of Service

Our design introduces denial-of-service opportunities for misbehaving requesters, responders, or the directory. We discuss these risks here, as well as methods to remedy them.

Perhaps the most troubling is a responder who returns $\rho \in C_{pk}(1_{\mathbb{G}})$ regardless of the request ciphertexts $\langle c_j \rangle_{j \in [\ell]}$ in message m1, thereby giving the requester reason to reject the user's chosen password even when the user's chosen password

[8]This estimate was retrieved from https://www.statista.com/topics/2237/internet-usage-in-the-united-states/ on December 4, 2018.

is not similar to others she set elsewhere. This denial-of-service attack would frustrate users, but fortunately a responder that misbehaves in this way can be caught by simple audit mechanisms. For example, at any point, the directory could generate a message m1 in which each $c_j \in C_{pk} \setminus C_{pk}(1_{\mathbb{G}})$ and for which it knows the private key sk corresponding to pk; if a responder responds with $\rho \in C_{pk}(1_{\mathbb{G}})$, then the directory has proof that the responder is lying and, e.g., can simply remove the responder from future queries. In principle, a requester could also generate such audit queries, though doing so would require the directory to suspend the user-consent mechanism in Sec. V-A1. In this case, a detection would enable the requester to learn that either one of the responders is misbehaving (but it would need help from the directory to figure out which one) or that the directory is misbehaving (in which case it would need to report it to some managing authority).

Other misbehaviors can render our framework silently ineffective while they persist. For example, a malicious directory could simply not query responders at all, instead forging the response ρ_i purportedly from each S_i to indicate *no* password reuse (i.e., $\rho_i \in C_{pk} \setminus C_{pk}(1_{\mathbb{G}})$). Again, a simple audit (knowingly attempting to reuse a password at a requester) can detect such misbehavior. Presuming such misbehaviors will occur rarely and be remedied quickly, we believe our framework will suffice to discourage password reuse even if it *usually* works.

As our framework enables the requester to perform precomputation to reduce its costs on the critical path of protocol execution, the critical-path computation cost of the protocol is greater for the responder than it is for the requester. This is even more true for misbehaving requesters that replay the same request, in an effort to occupy directory and responder resources. Of course, this concern is not unique to our framework, and various techniques to stem such denials of service exist that would be amenable to adoption in our framework (e.g., [24], [1]). In addition, steps detailed in Sec. V-A1 to require user consent (through clicking on a confirmation URL) to complete the protocol could interfere with such attacks. In the worst case, however, responders and the directory can refuse requests until the flood subsides, albeit temporarily reducing the utility of our framework to the status quo today.

VIII. Conclusion

Adams and Sasse famously declared, "Users are not the enemy" [2]. While we do not mean to suggest otherwise, it has also long been understood in a variety of contexts that users must be compelled to adhere to security policies, as otherwise they will not do so. Despite decades of haranguing users to stop reusing passwords, their adoption of methods to manage passwords more effectively has been painfully slow. This, in turn, has given rise to credential abuses that inflict considerable costs on service operators (see Sec. I).

We believe it is now time to consider imposing technical measures to interfere with the use of similar passwords across websites. In this paper we have presented one possible method for doing so, by coordinating password selection across websites so that similar passwords cannot be used for the same account identifier. Our framework combines a set-membership-test protocol (Sec. IV) with a variety of other defenses (Sec. V-A) to implement **account security** and **account location privacy**, the former of which we confirm via probabilistic model checking (Sec. V-B). Finally, we leveraged tendencies of how users reuse passwords to optimize the parameters for our framework, enabling it to be effective with surprisingly modest costs (Sec. VI-C).

Acknowledgment

We are grateful for comments on previous versions of this paper from Prof. Marina Blanton and anonymous reviewers. This work was supported in part by NSF grant 1330599.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM TOIT*, vol. 5, no. 2, May 2005.

[2] A. Adams and M. A. Sasse, "Users are not the enemy," *CACM*, vol. 42, Dec. 1999.

[3] Akamai, "[state of the internet]/security, Q4 2017 report," https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q4-2017-state-of-the-internet-security-report.pdf, 2017.

[4] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *44th IEEE/IFIP DSN*, Jun. 2014.

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *CACM*, vol. 13, no. 7, Jul. 1970.

[6] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *ESORICS*, ser. LNCS, vol. 6345, Sep. 2010.

[7] J. Boyan, "The Anonymizer: Protecting user privacy on the web," *Computer-Mediated Communication Magazine*, vol. 4, no. 9, Sep. 1997.

[8] A. S. Brown, E. Bracken, S. Zoccoli, and K. Douglas, "Generating and remembering passwords," *Applied Cognitive Psychology*, vol. 18, no. 6, 2004.

[9] R. Canetti, O. Goldreich, and S. Halevi, "The random oracle methodology, revisited," *JACM*, vol. 51, no. 4, Jul. 2004.

[10] Certicom Research, "SEC 2: Recommended elliptic curve domain parameters," http://www.secg.org/SEC2-Ver-1.0.pdf, 2000, standards for Efficient Cryptography.

[11] A. Clement, M. K. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché, "UpRight cluster services," in *22nd ACM SOSP*, Oct. 2009.

[12] K. Collins, "Facebook buys black market passwords to keep your account safe," https://www.cnet.com/news/facebook-chief-security-officer-alex-stamos-web-summit-lisbon-hackers/, Nov. 9 2016.

[13] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung, "Efficient robust private set intersection," in *7th ACNS*, ser. LNCS, vol. 5536, 2009.

[14] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *ISOC NDSS*, 2014.

[15] A. Davidson and C. Cid, "An efficient toolkit for computing private set operations," in *22nd ACISP*, ser. LNCS, vol. 10343, Jul. 2017.

[16] E. De Cristofaro, P. Gasti, and G. Tsudik, "Fast and private computation of cardinality of set intersection and union," in *11th CANS*, ser. LNCS, vol. 7712, 2012.

[17] E. De Cristofaro, J. Kim, and G. Tsudik, "Linear-complexity private set intersection protocols secure in malicious model," in *ASIACRYPT*, ser. LNCS, vol. 6477, 2010.

[18] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren, "Tripwire: Inferring internet site compromise," in *IMC*, Nov. 2017.

[19] S. K. Debnath and R. Dutta, "Secure and efficient private set intersection cardinality using Bloom filter," in *18th ISC*, ser. LNCS, vol. 9290, Sep. 2015.

[20] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "A Storm is coming: A modern probabilistic model checker," in *29th CAV*, ser. LNCS, vol. 10427, 2017.

[21] A. Dey and S. A. Weis, "PseudoID: Enhancing privacy for federated login," in *3rd PETS*, Aug. 2010.

[22] C. Diem, "On the discrete logarithm problem in elliptic curves," *Compositio Mathematica*, vol. 147, no. 1, Jan. 2011.

[23] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *13th USENIX Security*, Aug. 2004.

[24] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *CRYPTO*, ser. LNCS, vol. 740, 1993.

[25] R. Egert, M. Fischlin, D. Gens, S. Jacob, M. Senker, and J. Tillmanns, "Privately computing set-union and set-intersection cardinality via Bloom filters," in *20th ACISP*, ser. LNCS, vol. 9144, 2015.

[26] T. ElGamal, "A public-key cryptosystem and a signature scheme based on discrete logarithms," *IEEE TOIT*, vol. 31, no. 4, 1985.

[27] I. Erguler, "Achieving flatness: Selecting the honeywords from existing user passwords," *IEEE TPDS*, vol. 13, no. 2, 2016.

[28] T. Ferrill, "The 6 best password managers," https://www.csoonline.com/article/3198507/security/the-6-best-password-managers.html, Jan. 11 2018.

[29] "Federal Information Processing Standards (FIPS) 186-4, Digital Signature Standard (DSS)," http://dx.doi.org/10.6028/nist.fips.186-4, Jul. 2013, National Institute of Standards and Technology (NIST).

[30] D. Florêncio and C. Herley, "A large-scale study of web password habits," in *16th WWW*, 2007.

[31] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas, "Efficient set intersection with simulation-based security," *J. Cryptology*, vol. 29, no. 1, 2016.

[32] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *EUROCRYPT*, ser. LNCS, vol. 3027, May 2004.

[33] E. Gabber, P. B. Gibbons, D. M. Kristol, Y. Matias, and A. Mayer, "Consistent, yet anonymous, web access with LPWA," *CACM*, vol. 42, no. 2, Feb. 1999.

[34] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "O single sign-off, where art thou? An empirical analysis of single sign-on account hijacking and session management on the web," in *27th USENIX Security*, Aug. 2018.

[35] 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 *25th ACM CCS*, Oct. 2018.

[36] P. A. Grassi *et al.*, "Digital Identity Guidelines: Authentication and Lifecycle Management," https://doi.org/10.6028/NIST.SP.800-63b, Jun. 2017, NIST Special Publication 800-63B.

[37] D. Hayes, "Why you should write down your passwords and never reuse them," https://pressupinc.com/blog/2014/04/write-passwords-never-reuse/, Apr. 9 2014.

[38] IEEE-SA Standards Board, "IEEE standard specifications for public-key cryptography," https://doi.org/10.1109/IEEESTD.2000.92292, 2000, IEEE Standard 1363-2000.

[39] I. Ion, R. Reeder, and S. Consolvo, "'... no one can hack my mind': Comparing expert and non-expert security practices," in *SOUPS*, 2015.

[40] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *ACM CCS*, 2013.

[41] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian, "Scaling private set intersection to billion-element sets," in *18th Financial Crypto*, ser. LNCS, vol. 8437, Mar. 2014.

[42] J. Katz, R. Ostrovsky, and M. Yung, "Efficient and secure authenticated key exchange using weak passwords," *JACM*, vol. 57, no. 1, Nov. 2009.

[43] L. Kissner and D. Song, "Privacy-preserving set operations," in *25th CRYPTO*, ser. LNCS, vol. 3621, Aug. 2005.

[44] N. Koblitz and A. Menezes, "Another look at generic groups," *Advances in Mathematics of Communications*, vol. 1, no. 1, 2007.

[45] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical multi-party private set intersection from symmetric-key techniques," in *ACM CCS*, Oct. 2017.

[46] M. Kotadia, "Microsoft security guru: Jot down your passwords," https://www.cnet.com/news/microsoft-security-guru-jot-down-your-passwords/, May 24 2005.

[47] J. Manico and N. Mueller, "Credential stuffing," https://www.owasp.org/index.php/Credential_stuffing, Feb. 23 2015.

[48] U. Maurer, "Abstract models of computation in cryptography," in *10th IMA Cryptography and Coding*, ser. LNCS, vol. 3796, Dec. 2005.

[49] B. Menkus, "Understanding the use of passwords," *Computers and Security*, vol. 7, no. 2, 1988.

[50] T. Meskanen, J. Liu, S. Ramezanian, and V. Niemi, "Private membership test for Bloom filters," in *IEEE Trustcom/BigDataSE/ISPA*, Aug. 2015.

[51] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2005.

[52] D. Nield, "How to use the infinite number of email addresses Gmail gives you," https://fieldguide.gizmodo.com/how-to-use-the-infinite-number-of-email-addresses-gmail-1609458192, Jul. 7 2014.

[53] R. Nojima and Y. Kadobayashi, "Cryptographically secure Bloom-filters," *Trans. Data Privacy*, vol. 2, no. 2, Aug. 2009.

[54] 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 *24th ACM CCS*, Oct. 2017.

[55] G. Perng, M. K. Reiter, and C. Wang, "M2: Multicasting mixes for efficient and anonymous communication," in *26th ICDCS*, Jul. 2006.

[56] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on OT extension," *ACM TOPS*, vol. 21, no. 2, 2018.

[57] Ponemon Institute LLC, "The cost of credential stuffing," Ponemon Institute Research Report, Oct. 2017.

[58] S. Ramezanian, T. Meskanen, M. Naderpour, and V. Niemi, "Private membership test protocol with low communication complexity," in *11th NSS*, ser. LNCS, vol. 10394, Aug. 2017.

[59] S. Riley, "Password security: What users know and what they actually do," *Usability News*, vol. 8, no. 1, 2006.

[60] P. Rindal and M. Rosulek, "Improved private set intersection against malicious adversaries," in *EUROCRYPT*, ser. LNCS, vol. 10210, 2017.

[61] ——, "Malicious-secure private set intersection via dual execution," in *ACM CCS*, Oct. 2017.

[62] N. J. Rubenking, "The best password managers of 2018," https://www.pcmag.com/article2/0,2817,2407168,00.asp, Dec. 7 2017.

[63] S. Schechter, C. Herley, and M. Mitzenmacher, "Popularity *is* everything: A new approach to protecting passwords from statistical-guessing attacks," in *5th USENIX HotSec*, Aug. 2010.

[64] Shape Security, "2018 credential spill report," https://info.shapesecurity.com/rs/935-ZAM-778/images/Shape_Credential_Spill_Report_2018.pdf, 2018.

[65] 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 *SOUPS*, 2010.

[66] E. H. Spafford, "OPUS: Preventing weak password choices," *Computers & Security*, vol. 11, no. 3, 1992.

[67] E. Stobert and R. Biddle, "The password life cycle: User behaviour in managing passwords," in *SOUPS*, 2014.

[68] T. Takada, "Authentication shutter: Alternative countermeasure against password reuse attack by availability control," in *12th ARES*, Aug. 2017.

[69] S. Tamrakar, J. Liu, A. Paverd, J. Ekberg, B. Pinkas, and N. Asokan, "The circle game: Scalable private membership test using trusted hardware," in *ACM ASIACCS*, 2017.

[70] C. Wang, S. T. K. Jan, H. Hu, D. Bossart, and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *8th ACM CODASPY*, Mar. 2018.

[71] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *23rd ACM CCS*, 2016.

[72] K. C. Wang and M. K. Reiter, "How to end password reuse on the web," *arXiv preprint arXiv:1805.00566*, 2018.

[73] R. Wash, E. Rader, R. Berman, and Z. Wellmer, "Understanding password choices: How frequently entered passwords are re-used across websites," in *SOUPS*, 2016.

[74] W. Williamson, "What happens to stolen data after a breach?" http://www.securityweek.com/what-happens-stolen-data-after-breach, Mar. 17 2014.

[75] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: An algorithmic framework and empirical analysis," in *17th ACM CCS*, Oct. 2010.