Using Amnesia to Detect Credential Database Breaches

Ke Coby Wang, *University of North Carolina at Chapel Hill;*
Michael K. Reiter, *Duke University*

https://www.usenix.org/conference/usenixsecurity21/presentation/wang-ke-coby

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Using Amnesia to Detect Credential Database Breaches

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

Michael K. Reiter
Duke University
michael.reiter@duke.edu

Abstract

Known approaches for using decoy passwords (honeywords) to detect credential database breaches suffer from the need for a trusted component to recognize decoys when entered in login attempts, and from an attacker's ability to test stolen passwords at *other* sites to identify user-chosen passwords based on their reuse at those sites. Amnesia is a framework that resolves these difficulties. Amnesia requires no secret state to detect the entry of honeywords and additionally allows a site to monitor for the entry of its decoy passwords elsewhere. We quantify the benefits of Amnesia using probabilistic model checking and the practicality of this framework through measurements of a working implementation.

1 Introduction

Credential database breaches have become a widespread security problem. Verizon confirmed 3950 database breaches globally between Nov. 2018 and Oct. 2019 inclusive; of those 1665 breaches for which they identified victims, 60% leaked credentials [43].[1] Credential database breaches are the largest source of compromised passwords used in credential stuffing campaigns [42], which themselves are the cause of the vast majority of account takeovers [41]. Unfortunately, there is usually a significant delay between the breach of a credential database and the discovery of that breach; estimates of the average delay range from 7 [23] to 15 [41] months. The resulting window of vulnerability gives attackers the opportunity to crack the passwords offline (if the stolen credential database stores only password hashes), to determine their value by probing accounts using them [41], and then to either use them directly to extract value or sell them through illicit forums for trafficking stolen credentials [41, 42].

Decoy passwords have been proposed in various forms to interfere with the attacker's use of a stolen credential database. In these proposals (see Sec. 2), a site (the target) stores decoy passwords alongside real passwords in its credential database, so that if the attacker breaches the database, the correct passwords are hidden among the decoys. The attacker's entry of a decoy password can alert the target to its breach; the term *honeywords* has been coined for decoys used in this way [25].

While potentially effective, honeywords suffer from two related shortcomings that, we believe, have limited their use in practice. First, previous proposals that leverage honeywords require a trusted component to detect the entry of a honeyword, i.e., a component that retains secret state even *after* the target has been breached. Such a trusted component is a strong assumption, however, and begs the question of whether one could have been relied upon to prevent the breach of the target's database in the first place. Second, the effectiveness of honeywords depends on the indistinguishability of the user-chosen password from the decoys when they are exposed to an attacker. However, because so many users reuse their chosen passwords across multiple accounts [11, 36, 44], an attacker can simply test (or *stuff*) all passwords for an account leaked from the target at accounts for the same user at other sites. Any password that works at another site is almost certainly the user-chosen password at the target.

In this paper, we resolve both of these difficulties and realize their solutions in a framework called Amnesia. First, we show that honeywords can be used to detect a target's database breach with no persistent secret state at the target, a surprising result in light of previous work. Specifically, we consider a threat model in which the target is breached passively but completely and potentially repeatedly. Without needing to keep secrets from the attacker, Amnesia nevertheless enables the target to detect its own breach probabilistically, with benefits that we quantify through probabilistic model checking. Our results show, for example, that Amnesia substantially reduces the time an attacker can use breached credentials to access accounts without alerting the target to its breach.

To address credential stuffing elsewhere to distinguish the user-chosen password from the honeywords, Amnesia enables the target to monitor for the entry of passwords stolen from it

[1] This number excludes 14 breaches of victims in Latin America and the Caribbean for which the rate of credential leakage was not reported.

at other sites, called monitors. Via this framework, incorrect passwords entered for the same user at monitors are treated (for the purposes of breach detection) as if they had been entered locally at the target. One innovation to accomplish this is a cryptographic protocol by which a monitor transfers the password attempted in an unsuccessful login there to the target, but only if the attempted password is one of the passwords (honey or user-chosen) for the same account at the target; otherwise, the target learns nothing. We refer to this protocol as a *private containment retrieval* (PCR) protocol, for which we detail a design and show it secure. Leveraging this PCR protocol, we show that Amnesia requires no trust in the monitors for the target to accept a breach notification. In other words, even if a monitor is malicious, it cannot convince an unbreached target that it has been breached.

We finally describe the performance of our Amnesia implementation. Our performance results suggest that the computation, communication and storage costs of distributed monitoring are minimal. For example, generating a monitoring response takes constant time and produces a constant-size result, as a function of the number of honeywords, and is practical (e.g., no more than 10ms and about 1KB, respectively).

To summarize, our contributions are as follows:

- We develop the first algorithm leveraging honeywords by which a target site can detect the breach of its password database, while relying on *no* secret persistent state. We evaluate this design using probabilistic model checking to quantify the security it provides.
- We extend this algorithm with a protocol to monitor accounts at monitors to detect the use of the target's honeywords there. Our algorithm is the first such proposal to ensure no false detections of a database breach, despite even malicious behavior by monitors.
- A core component of this algorithm is a new cryptographic protocol we term a *private containment retrieval* protocol, which we detail and prove correct.
- We describe the performance of our algorithm using an implementation and show that it is practical.

2 Related Work

Within research on decoy passwords, we are aware of only two proposals by which a target can detect its *own* breach using them. Juels and Rivest [25] coined the term *honeywords* for decoy passwords submitted in login attempts to signal to a site that it was breached by an attacker. In their proposal and works building on it (e.g., [14]), the target is augmented with a trusted *honeychecker* that stores which of the passwords listed with the account is the user-chosen one; login attempts with others alert the site to its breach. Almeshekah et al. [2] use a machine-dependent function (e.g., hardware security module) in the password hash at the target site to prevent offline cracking of its credential database if breached. Of more relevance here, an attacker who is unaware of this defense and so attempts to crack its database offline will produce plausible decoy passwords (*ersatzpasswords*) that, when submitted, alert the target site to its breach. The primary distinction between these proposals and ours is that ours permits a target to detect its own breach *without any secret persistent state*. In contrast, these proposals require a trusted component—the honeychecker or the machine-dependent function—whose state is assumed to remain secret even after the attacker breaches the site. In addition, we reiterate that ersatzpasswords are effective in alerting the target to its breach only if the attacker is unaware of the use of this scheme, as otherwise the attacker will know that passwords generated through offline cracking without access to the machine-dependent function are ersatzpasswords.

Other uses of decoy passwords leverage defenses at *other, unbreached* sites—either their online guessing defenses generically [5,28] or their cooperation to check for decoy passwords specifically [5, 48]—to defend accounts whose credentials have been stolen, whether by phishing [48], user device compromise [5], or the target site's database breach [28]. While we extend our design in Sec. 5 to monitor for a target's honeywords being submitted in login attempts at monitors, to our knowledge our design is the first to eliminate the need for the target to trust another site in order to accept that a detected breach actually occurred. Specifically, in our design a monitor, even if malicious, cannot convince an unbreached site that it has been breached.

Various other works have leveraged decoy *accounts* to detect credential database breaches, i.e., accounts with no owner that, if ever accessed, reveal the breach of the account's site or a site where a replica of the account was created (e.g., [14,21]). In Tripwire [13], each decoy account is registered with a distinct email address and password, for which the password at the email provider is the same. Any login to the email account (provided that the email provider is itself not compromised) suggests the breach of the website where that email address was used to register an account. Like the previously discussed proposals, this design places trust in the detecting party (the email provider or, in this case, the researchers working with it) to be truthful when reporting the breach of a target. Indeed, DeBlasio et al. report that sites' unwillingness to trust the evidence they provided of the sites' breaches was an obstacle to getting them to act.[2] Moreover, the utility of artificial accounts hinges critically on their indistinguishability from real ones, and if methods using them became effective in hindering attacker activity, ensuring the indistinguishability of these accounts would presumably become its own arms race. Our design is agnostic to whether it is deployed on real or decoy accounts, sidestepping the need for convincing decoy accounts but also demanding attention to the risks to real

[2]The paper concludes, "A major open question, however, is how much (probative, but not particularly illustrative) evidence produced by an external monitoring system like Tripwire is needed to convince operators to act, such as notifying their users and forcing a password reset" [13, Section 8].

accounts that it might introduce.

To be fair, generation of honeywords that are sufficiently indistinguishable from real ones is itself a topic of active investigation (e.g., [1, 14, 45]). Here we will simply assume that a site can generate honeywords in isolation to satisfy certain properties, detailed in Sec. 3. The development of methods to achieve these properties is a separate concern.

An alternative to decoy passwords or accounts for defending against a breach of a site's credential database is for the site to instead leverage a breach-hardening service. Even after having breached the target's credential database, the attacker must succeed in an *online* dictionary attack with the breach-hardening service per stolen credential he wishes to use, provided that the breach-hardening service is itself not simultaneously breached (e.g., [15, 30–32, 40]). While differing in their details, these schemes integrate the breach-hardening service tightly into the target's operation, in the sense that, e.g., the benign failure of a breach-hardening service would interfere with login attempts at the target. In contrast, while the benign failure of our monitors would render them useless for helping to detect the target's breach, the operation of the target would be otherwise unaffected.

3 Honeywords

We assume the existence of a randomized honeyword generator HoneyGen that, given an account identifier a, user-chosen password π_a, and integer k, produces a set Π_a containing π_a and k other strings and having the following properties. We use "$\leftarrow$" to denote assignment of the result of evaluating the expression on its right to the variable on its left, and "$\xleftarrow{\$}$" to denote sampling an element uniformly at random from the set on its right and assigning the result to the variable on its left.

First, the essential purpose of honeywords is to make it difficult for an adversary who breaches a credential database to determine which of the passwords listed for an account a is the user-chosen one. In other words, for any attacker algorithm A that is given the account identifier a and its set of passwords Π_a, we assume

$$\mathbb{P}\left(\pi = \pi_a \;\middle|\; \begin{array}{l} \Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k) \\ \pi \leftarrow A(a,\Pi_a) \end{array}\right) \approx \frac{1}{k+1} \qquad (1)$$

Second, because honeywords are intended to alert the target to a breach of its credential database, avoiding false alarms requires that an adversary be unable to generate a honeyword for an account without having actually breached the target. In particular, this property would ideally be achieved even if the user-chosen password π_a is known, e.g., because the user was phished or because she reused π_a as her password at another site that was compromised. While these place the user's account at the target at risk, neither equates to the target's wholesale breach and so should not suffice to induce a breach detection at the target. That is, for any attacker algorithm B that knows only the account identifier a and user-chosen password π_a, we assume:

$$\mathbb{P}\left(\pi \in \Pi_a \setminus \{\pi_a\} \;\middle|\; \begin{array}{l} \Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k) \\ \pi \leftarrow B(a,\pi_a) \end{array}\right) \approx 0 \qquad (2)$$

This assumption implies that any two invocations of $\mathsf{HoneyGen}(a,\pi_a,k)$ produce sets Π_a, Π'_a that intersect *only* in π_a with near certainty. Otherwise, an adversary $B(a,\pi_a)$ that invokes $\Pi'_a \leftarrow \mathsf{HoneyGen}(a,\pi_a,k)$ and returns a random $\pi \in \Pi'_a \setminus \{\pi_a\}$ would violate (2). In other words, (2) implies that the honeywords generated at two different sites for the same user's accounts are distinct, even if the user reuses the same password for both accounts.

4 Detecting Honeyword Entry Locally

The first contribution of this paper is in demonstrating how the target site can detect its own breach while relying on no secret persistent state. We detail the threat model for this section in Sec. 4.1 and provide the detection algorithm in Sec. 4.2. We demonstrate the efficacy of this algorithm in Sec. 4.3.

4.1 Threat Model

Our goal is to enable a site, called the target, to detect that its credential database has been stolen. We assume that the target uses standard password-based authentication, i.e., in which the password is submitted to the target under the protection of a cryptographic protocol such as TLS.

We allow for an attacker to breach the target *passively only*, in which case it captures all persistent storage at the site associated with validating or managing account logins. Throughout this paper, this persistent storage is denoted DB, and information associated specifically with account a is denoted DB_a. In particular, the information captured includes the passwords listed for each of the site's user accounts (DB_a.auths); if stored as salted hashes, the attacker can crack the passwords offline. The attacker also captures any long-term cryptographic keys of the site. As will become relevant below, we allow the attacker to capture the site's persistent storage multiple times, periodically.

We stress that the information captured by the attacker includes only information *stored persistently* at the site. Recall that the principle behind honeywords is to leverage their use in login attempts to alert the target that its credential database has been stolen. As such, we must assume that transient information that arrives in a login attempt but is not stored persistently at the site is unavailable to the attacker. Otherwise, the attacker would simply capture the correct password for an account once the legitimate owner of that account logs in. Since the site's breach leaks any long-term secrets, this

assumption implies that the cryptographic protocol protecting user logins provides perfect forward secrecy [20][3] or that the attacker simply cannot observe login traffic. Similarly, we assume that despite breaching the target site, the attacker cannot predict future randomness generated at the site.

We also highlight that, like in Juels and Rivest's honeyword design [25], we do not consider the active compromise of the target. In particular, the integrity of the target's persistent storage is maintained despite the attacker's breach, and the site always executes its prescribed algorithms. Without this assumption, having the target detect its own breach is not possible. We do, however, permit the attacker to submit login attempts to the target via its provided login interface.

Finally, while the adversary might steal passwords chosen by some legitimate users of the target (e.g., by phishing, keylogging, or social engineering) and be a user of the site himself, Amnesia leverages the activity of other account owners, each of whose chosen password is indistinguishable to the attacker in the set of passwords listed for her account. As such, when we refer to account owners below, we generally mean ones who have not been phished or otherwise compromised.

4.2 Algorithm

In this section we detail our algorithm for a target to leverage honeywords for each of its accounts to detect its own breach. Somewhat counterintuitively, in our design the honeywords the target site creates for each account are indistinguishable from the correct password, even to itself (and so to an attacker who breaches it)—hence the name *Amnesia*. However, the passwords for an account (i.e., both user-chosen and honey) are *marked* probabilistically with binary values. Marking ensures that the password last used to access the account is always marked (i.e., its associated binary value is 1). Specifically, upon each successful login to an account, the set of passwords is remarked with probability p_{remark}, in which case the entered password is marked (with probability 1.0) and each of the other passwords is marked independently with probability p_{mark}. As such, if an attacker accesses the account using a honeyword, then the user-chosen password becomes unmarked with probability $p_{\mathsf{remark}}(1 - p_{\mathsf{mark}})$. In that case, the breach will be detected when the user next accesses the account, since the password she supplies is unmarked.

More specifically, the algorithm for the target to detect its own breach works as follows. The algorithm is parameterized by probabilities p_{mark} and p_{remark}, and an integer $k > 0$. It leverages a procedure mark shown in Fig. 1, which marks the given element e with probability 1.0, marks other elements of DB_a.auths for the given account a with probability p_{mark}, and stores these markings in the credential database for account a as the function DB_a.marks.

[3] Cohn-Gordon et al. [9] observe that for a passive attacker, perfect forward secrecy implies protection not only against the future compromise of the long-term key but also its past compromise.

mark(a, e): /* Assumption: $e \in \mathsf{DB}_a$.auths */
- $\mathcal{X} \leftarrow \mathsf{DB}_a$.auths
- Choose $marked : \mathcal{X} \to \{0, 1\}$ subject to:
  - $marked(e) = 1$
  - $\forall e' \in \mathcal{X} \setminus \{e\} : marked(e') \sim \mathsf{Bernoulli}(p_{\mathsf{mark}})$
- DB_a.marks $\leftarrow marked$

Figure 1: Procedure mark, used in Secs. 4–5

Password registration: When the user sets (or resets) the password for her account a, she provides a user-chosen password π. The password registration system generates DB_a.auths $\leftarrow \mathsf{HoneyGen}(a, \pi, k)$ and then invokes mark(a, π).

Login: When a login is attempted to account a with password π, the outcome is determined as follows:
- If $\pi \notin \mathsf{DB}_a$.auths, then the login attempt is unsuccessful.
- If $\pi \in \mathsf{DB}_a$.auths and $\mathsf{DB}_a.\mathsf{marks}(\pi) = 0$, then the login attempt is unsuccessful *and a credential database breach is detected*.
- Otherwise (i.e., $\pi \in \mathsf{DB}_a$.auths and $\mathsf{DB}_a.\mathsf{marks}(\pi) = 1$) the login attempt is successful.[4] In this case, mark(a, π) is executed with probability p_{remark}.

This algorithm requires that a number of considerations be balanced if an attacker can breach the site repeatedly to capture its credential database many times. Consider that:
- Repeatedly observing the passwords left marked by user logins permits the attacker to narrow in on the user-chosen password as the one that is always marked. This suggests that legitimate logins should remark the passwords as rarely as possible (i.e., p_{remark} should be small) or that, when remarking occurs, doing so results in passwords already marked staying that way (i.e., p_{mark} should be large).
- If the attacker accesses an account between two logins by the user, a remarking *must* occur between the legitimate logins if there is to be any hope of the second legitimate login triggering a detection (i.e., p_{remark} should be large).
- If the attacker is permitted to trigger remarkings many times between consecutive legitimate logins, however, then it can do so repeatedly until markings are restored on most of the passwords that were marked when it first accessed the account. The attacker could thereby reduce the likelihood that the next legitimate login detects the breach. This suggests that it must be difficult for the attacker to trigger many remarkings on an account (i.e., p_{remark} should be small) or that when remarkings occur, significantly many passwords

[4] Or more precisely, the stage of the login pipeline dealing with the password is deemed successful. Additional steps, such as a second-factor authentication challenge, could still be required for the login to succeed.

are left unmarked (i.e., p_{mark} should be small).

All of this is complicated by the fact that the target site cannot distinguish between legitimate and attacker logins, of course. While an anomaly detection system (ADS) using features of each login attempt *other* than the password entered (e.g., [18]) could provide a noisy indication, unfortunately our threat model permits the attacker to learn *all persistent state* that the target site uses to manage logins; this would presumably include the ADS model for each account, thereby enabling the adversary to potentially evade it. For this reason, we eschew this possibility, instead settling for a probability p_{remark} of remarking passwords on a successful login and, if so, a probability p_{mark} with which each password is marked (independently), that together balance the above concerns. We explore such settings in Sec. 4.3.

4.3 Security

Methodology: To evaluate the security of our algorithm, we model an attack as a Markov decision process (MDP) consisting of a set of states and possible transitions among them. When the MDP is in a particular state, the attacker can choose from a set of available actions, which determines a probability distribution over the possible next states as a function of the current state and the action chosen. Using probabilistic model checking, we can evaluate the success of the adversary in achieving a certain goal (see below) under his *best* possible strategy for doing so. In our evaluations below, we use the Prism model checker [29].

The basic distributions for modeling our algorithm for a single account are straightforward. Let $\mathbb{N}_\ell$ denote the number of passwords that the attacker always observes as marked in ℓ breaches of the target, with each pair of breaches separated by at least one remarking in a legitimate-user login. (Breaches with no remarking between them will observe the same marks.) Then, $\mathbb{N}_\ell \sim \mathsf{binomial}\left(k, (p_{\mathsf{mark}})^\ell\right) + 1$, where the "+ 1" represents the user-chosen password, which remains marked across these ℓ remarkings. Now, letting $\mathbb{A}_n$ denote the number of these passwords that are marked after an adversary-induced remarking, conditioned on $\mathbb{N}_\ell = n+1$, we know $\mathbb{A}_n \sim \mathsf{binomial}(n, p_{\mathsf{mark}}) + 1$, where the "+ 1" represents the marked password that the adversary submitted to log into the account, which remains marked with certainty. If $\mathbb{A}_n = \alpha + 1$ after the adversary's login, then the probability of the target detecting its own breach upon the legitimate user's next login to this account is $1 - \frac{\alpha+1}{n+1}$.

To turn these distributions into a meaningful MDP, however, we need to specify some additional limits.

- The number of attacker breaches until it achieves ℓ that each follows a distinct remarking induced by a legitimate user login is dependent not only on p_{remark}, but also the rate of user logins. In our experiments, we model user logins as Poisson arrivals with an expected number $\lambda = 1$ login per time unit. We permit the attacker to breach the site and capture all stored state at the end of each time unit.
- Even with this limit on the rate of legitimate user logins, an attacker that breaches the site arbitrarily many times will eventually achieve $\mathbb{N}_\ell = 1$ and so will know the legitimate user's password. In practice, however, the attacker cannot wait arbitrarily long to access an account, since there is a risk that his breaches will be detected by other means (i.e., not by our algorithm). To model this limited window of vulnerability, we assume that the time unit in which the breach is discovered by other means (at the end of the time unit), and so the experiment *stops*, is represented as a random variable $\mathbb{S}$ distributed normally with mean μ_{stop} and relative standard deviation $\chi_{\mathsf{stop}} = 0.2$. For example, assuming a seven-month average breach discovery delay [23], an account whose user accesses it once per week on average, would have $\mu_{\mathsf{stop}} \approx 30$ time units (weeks).
- Once the attacker logs into the account with one of the $n+1$ passwords that it observed as always marked in its breaches, it can log in repeatedly (i.e., resample $\mathbb{A}_n$) to leave the account with marks that minimize its probability of detection on the next legitimate user login. If allowed an unbounded number of logins, it can drive its probability of detection to zero. Therefore, we assume that the site monitors accounts for an unusually high rate of *successful* logins, limiting the adversary to at most Λ per time unit.

Let random variable $\mathbb{L}$ denote the time unit at which the attacker logs into the account for the first time, and let random variable $\mathbb{D} \leq \mathbb{S}$ denote the time unit at which the attacker is detected. That is, $\mathbb{D} < \mathbb{S}$ means that our algorithm detected the attacker before he was detected by other means. Moreover, note that $\mathbb{L} < \mathbb{D}$, since our algorithm can detect the attacker only after he logs into the account. We define the benefit of our algorithm to be the expected number of time units that our algorithm deprives the attacker of undetectably accessing the account, expressed as a fraction of the number of time units it could have done so in the absence of our algorithm. In symbols:

$$\mathsf{benefit} = \frac{\mathbb{E}(\mathbb{S}-\mathbb{L}) - \mathbb{E}(\mathbb{D}-\mathbb{L})}{\mathbb{E}(\mathbb{S}-\mathbb{L})} = 1 - \frac{\mathbb{E}(\mathbb{D}-\mathbb{L})}{\mathbb{E}(\mathbb{S}-\mathbb{L})} \qquad (3)$$

When computing benefit, we do so for an attacker strategy maximizing $\mathbb{E}(\mathbb{D}-\mathbb{L})$, i.e., against an attacker that maximizes the time for which it accesses the account before it is detected.

Results: The computational cost of model-checking this MDP is such that we could complete it for only relatively small (but still meaningful) parameters. The results we achieved are reported in Figs. 2–4. To explore how increasing each of k, Λ, and μ_{stop} affects benefit, each of the tables in Fig. 2 corresponds to modifying one parameter from the baseline table shown in Fig. 2a, where $k = 48$, $\Lambda = 4$, and $\mu_{\mathsf{stop}} = 8$. Each number in each table is the benefit of a corresponding $\langle p_{\mathsf{remark}}, p_{\mathsf{mark}} \rangle$ parameter pair, where higher numbers are better. When k is increased from 48 to 64 (Fig. 2b), we can see

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.11	.11	.10	.09	.07	.06	.04	.03	.02
.30	.16	.15	.14	.12	.10	.08	.06	.04	.02
.40	.21	.21	.19	.16	.14	.11	.08	.05	.02
.50	.27	.26	.24	.20	.17	.13	.10	.07	.03
.60	.31	.30	.27	.23	.19	.15	.11	.07	.03
.70	.34	.35	.32	.27	.23	.18	.13	.09	.04
.80	.32	.38	.35	.30	.25	.19	.14	.09	.04
.90	.32	.41	.40	.34	.28	.22	.15	.11	.05
1.0	.33	.40	.42	.38	.31	.24	.17	.12	.05

(a) Baseline

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.12	.11	.10	.09	.07	.06	.04	.03	.02
.30	.17	.16	.15	.12	.10	.08	.06	.04	.02
.40	.23	.22	.19	.16	.14	.11	.08	.05	.03
.50	.29	.27	.24	.21	.17	.14	.10	.07	.03
.60	.33	.31	.28	.24	.20	.16	.11	.08	.03
.70	.37	.36	.33	.28	.23	.19	.13	.09	.04
.80	.35	.40	.36	.31	.26	.21	.15	.10	.04
.90	.34	.43	.41	.35	.29	.23	.16	.11	.05
1.0	.34	.42	.45	.38	.32	.26	.18	.12	.05

(b) $k = 64$

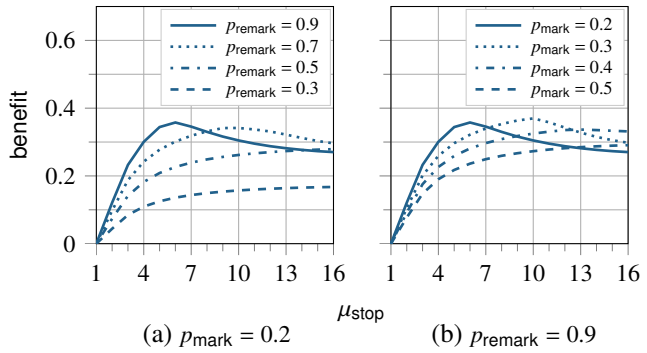
p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.05	.04	.04	.03	.02	.02	.01
.20	.11	.10	.10	.08	.07	.05	.04	.03	.01
.30	.15	.15	.14	.12	.10	.08	.05	.04	.02
.40	.20	.19	.18	.15	.12	.10	.07	.05	.02
.50	.25	.24	.22	.19	.15	.12	.08	.06	.03
.60	.29	.28	.26	.22	.18	.14	.10	.07	.03
.70	.29	.32	.30	.25	.21	.16	.11	.08	.03
.80	.29	.35	.33	.28	.23	.18	.12	.08	.03
.90	.28	.38	.37	.31	.25	.19	.13	.09	.04
1.0	.28	.36	.41	.35	.28	.22	.15	.10	.04

(c) $\Lambda = 8$

p_{remark} \ p_{mark}	.10	.20	.30	.40	.50	.60	.70	.80	.90
.10	.06	.06	.06	.05	.04	.03	.02	.02	.01
.20	.12	.12	.11	.10	.08	.06	.05	.03	.02
.30	.17	.17	.15	.13	.11	.09	.06	.04	.02
.40	.23	.22	.21	.18	.15	.11	.08	.06	.03
.50	.29	.28	.26	.22	.18	.14	.10	.07	.03
.60	.31	.32	.30	.25	.21	.16	.12	.08	.04
.70	.30	.36	.35	.30	.24	.19	.14	.09	.04
.80	.28	.34	.38	.33	.27	.21	.15	.10	.04
.90	.28	.34	.39	.37	.30	.23	.16	.11	.05
1.0	.31	.38	.40	.40	.33	.26	.18	.13	.05

(d) $\mu_{\text{stop}} = 12$

Figure 2: benefit of local detection, as k (b), Λ (c), and μ_{stop} (d) are increased individually from the "baseline" (a) of $k = 48$, $\Lambda = 4$, and $\mu_{\text{stop}} = 8$



(a) $p_{\text{mark}} = 0.2$ (b) $p_{\text{remark}} = 0.9$

Figure 3: benefit as a function of μ_{stop} with varying p_{remark} and varying p_{mark} ($k = 32, \Lambda = 4$)

a slight boost to the benefit. However, increasing Λ or μ_{stop}, shown in Fig. 2c and Fig. 2d respectively, causes benefit to drop slightly. The reasons behind these drops are that larger Λ (i.e., more repeated logins by the attacker) give him a better chance to leave with a reduced probability of detection, and a larger μ_{stop} allows the attacker to observe more user logins and so more remarkings (to minimize $\mathbb{N}_\ell$) before he is detected by other means.

This latter effect is illustrated in Fig. 3, which shows benefit as a function of μ_{stop}. When $\mu_{\text{stop}} \leq 7$, the settings $p_{\text{mark}} = 0.2$, $p_{\text{remark}} = 0.9$ yield the best benefit among the combinations pictured in Fig. 3. However, as μ_{stop} grows, the longer time (i.e., larger ℓ) the attacker can wait to access the account affords him a lower $\mathbb{N}_\ell$ and so a lower probability of being detected when the legitimate user subsequently logs in. This effect can be offset by decreasing p_{remark} (Fig. 3a), increasing p_{mark} (Fig. 3b), or both.

The impact of Λ is shown in Fig. 4, which plots benefit as a function of k for various Λ. Fig. 4 shows that even when the attacker logs in more frequently than the user by a factor of $\Lambda = 10$, our algorithm still remains effective with benefit ≈ 0.5 for moderately large k. That said, while Fig. 4 suggests that increasing k into the hundreds should suffice, we will see in Sec. 5 that an even larger k might be warranted when credential stuffing is considered.

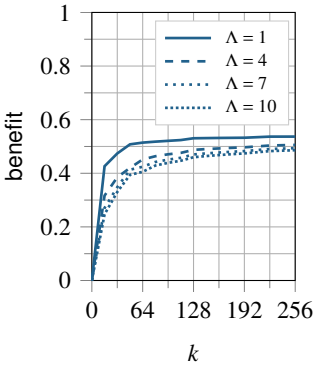


Figure 4: benefit as a function of k with varying Λ ($p_{\text{mark}} = 0.3$, $p_{\text{remark}} = 1.0$, $\mu_{\text{stop}} = 8$)

Interpreting benefit: As we define it, benefit is a conservative measure, in two senses. First, benefit is calculated (via probabilistic model checking) against the strongest attacker possible in our threat model. Second, benefit is computed only for one account, but detection on *any* account is enough to inform the target of its breach. For an attacker whose goal is to assume control of a large number of accounts at the target (vs. one account specifically), the detection power of our algorithm will be much higher.

That said, quantifying that detection power holistically for the target is not straightforward. Recall that benefit is defined in terms of time units wherein the legitimate user is expected to login $\lambda = 1$ time. As such, the real-time length of this unit for a frequently accessed account will be different than for an infrequently accessed one. And, since μ_{stop} is expressed in this time unit, μ_{stop} will be larger for a frequently accessed account than for an infrequently accessed one, even though the real-time interval that passes before a site detects its own breach by means other than Amnesia might be independent of the legitimate login rates to accounts. Thus, extrapolating the per-account benefit to the security improvement for a target holistically requires knowledge of the legitimate login rates across all the sites' accounts as a function of real time, adjusting μ_{stop} (and χ_{stop}) accordingly per account, and translating the per-account benefits back into a real-time measure.

5 Detecting Remotely Stuffed Honeywords

When a credential database is breached, it is common for attackers to submit the login credentials therein (i.e., usernames

and passwords) to *other* sites, in an effort to access accounts whose user set the same password as she did at the breached site. These attacks, called *credential stuffing*, are already the primary attack yielding account takeovers today [41]. But even worse for our purposes here, credential stuffing enables an attacker to circumvent the honeywords at a breached target site: If a user reused her password at another site, then stuffing the breached passwords there will reveal which is the user-chosen password, i.e., as the one that gains access. The attacker can then return to the target site with the correct password to access the user's account at the target.

The design in this section mitigates credential stuffing as a method to identify the user's chosen password, by ensuring that stuffing honeywords at other sites probabilistically still alerts the target site to its breach. At a high level, the target maintains a set of monitor sites and can choose to monitor an account at any of those monitors. To monitor the account at a monitor, the target sends the monitor a *private containment retrieval* (PCR) query for this account identifier, to which the monitor responds after any unsuccessful login attempt to this account (potentially even if the account does not exist at the monitor). In the abstract, a PCR query is a private (encrypted) representation of a set $\mathcal{X}$ of elements known to the target, and a response computed with element e reveals to the target the element e if $e \in \mathcal{X}$ and nothing otherwise. In this case, the target's set $\mathcal{X}$ contains the local password hashes for the user's account. If a monitor then sends a response computed using some $e \in \mathcal{X}$, the target can treat e as if it were attempted locally, permitting the detection of a breach just as in Sec. 4.

5.1 Threat Model

As in Sec. 4.1, we allow the adversary to breach the target passively, thereby learning all information persistently stored by the site for the purpose of determining the success of its users' login attempts. We highlight that in this section, the breached information includes a private key that is part of the target's stored state for managing login attempts in our algorithm. So, if the target is breached, then this private key is included in the data that the attacker learns.

We permit the attacker that breaches the target to also *actively* compromise monitors, in which case these monitors can behave arbitrarily maliciously. Malicious monitors can refuse to help the target detect its own breach via our design, e.g., by simply refusing to respond. However, our scheme must ensure that even malicious monitors cannot convince a target that it has been breached when it has not. Moreover, malicious monitors should not be able to leverage their participation in this protocol to attack passwords at a target that is never breached.

We do not permit the attacker to interfere with communication between a (breached or unbreached) target and an uncompromised monitor. Otherwise, the attacker could prevent the target from discovering its breach by simply refusing to let it communicate with uncompromised monitors.

Our design assumes that different sites can ascertain a common identifier a for the same user's accounts at their sites, at least as well as an attacker could. In practice, this would typically be the email address (or some canonical version thereof, see [46]) registered by the user for account identification or password-reset purposes.

5.2 Private Containment Retrieval

The main building block for our design is a private containment retrieval (PCR) protocol with the following algorithms.

- pcrQueryGen is an algorithm that, on input a public key pk and a set $\mathcal{X}$, generates a PCR query $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$.
- pcrRespGen is an algorithm that, on input a public key pk, an element e, and a query $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$, outputs a PCR response $Z \leftarrow \mathsf{pcrRespGen}_{pk}(e, Y)$.
- pcrReveal is an algorithm that on input the private key sk corresponding to pk, an element $e' \in \mathcal{X}$, and a response $Z \leftarrow \mathsf{pcrRespGen}_{pk}(e, Y)$ where $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathcal{X})$, outputs a Boolean $z \leftarrow \mathsf{pcrReveal}_{sk}(e', Z)$ where $z = \mathit{true}$ iff $e' = e$.

Informally, this protocol ensures that Y reveals nothing about $\mathcal{X}$ (except its size) to anyone not holding sk; that Z computed on $e \notin \mathcal{X}$ reveals nothing about e (except $e \notin \mathcal{X}$); and that if $\mathsf{pcrReveal}_{sk}(e', Z) = \mathit{true}$, then the party that computed Z knows e'. We make these properties more precise and provide an implementation in Sec. 6.

5.3 Algorithm

We first provide greater detail about how the target maintains its credential database. Whereas in Sec. 4 we left hashing of the honey and user-chosen passwords in DB_a.auths implicit, in this section we need to expose this hashing explicitly for functional purposes. Consistent with current best practices, the target represents DB_a.auths as a set of hashes salted with a random κ-bit salt DB_a.salt, including one hash $f(s, \pi)$ of the user-chosen password π where $s \leftarrow \mathsf{DB}_a$.salt and a salted hash $f(s, \pi')$ for each of k honeywords π'. Then, testing whether π is either a honey or user-chosen password amounts to testing $f(s, \pi) \in \mathsf{DB}_a$.auths. In addition to these refinements, for this algorithm the target is also initialized with a public-key/private-key pair $\langle pk, sk \rangle$ for use in the PCR protocol, and a set $\mathcal{S}$ of possible monitors (URLs). If the target R is breached, then all of DB, $\mathcal{S}$, and $\langle pk, sk \rangle$ are captured by the attacker.

The algorithm below treats local logins at the target R similar to how they were treated in Sec. 4, with the exception of exposing the hashing explicitly. In addition, the algorithm permits R to ask monitor S to monitor a. To do so, R sends a PCR query Y to S computed on DB_a.auths. Upon receiving this request, S simply saves it for use on each incorrect login to a at S, to generate a PCR response to R. The hash

encoded in this response is then treated at R (for the purposes of detecting a breach) as if it has been entered in a local login attempt. In sum, the protocol works as described below.

Password registration at R: When the user (re)sets the password for her account a at the target site R, she provides her chosen password π. The password registration system at R executes:

- $\Pi_a \leftarrow \mathsf{HoneyGen}(a,\pi,k)$
- $\mathsf{DB}_a.\mathsf{salt} \xleftarrow{\$} \{0,1\}^\kappa$
- $\mathsf{DB}_a.\mathsf{auths} \leftarrow \{f(\mathsf{DB}_a.\mathsf{salt},\pi')\}_{\pi' \in \Pi_a}$
- $\mathsf{mark}(a, f(\mathsf{DB}_a.\mathsf{salt},\pi))$

Login attempt at R: For a login attempted to account a with password π at R, the outcome is determined as follows, where $h \leftarrow f(\mathsf{DB}_a.\mathsf{salt},\pi)$:

- If $h \notin \mathsf{DB}_a.\mathsf{auths}$, the login attempt is unsuccessful.
- If $h \in \mathsf{DB}_a.\mathsf{auths}$ and $\mathsf{DB}_a.\mathsf{marks} = 0$, then the login attempt is unsuccessful *and a credential database breach is detected*.
- Otherwise (i.e., $h \in \mathsf{DB}_a.\mathsf{auths}$ and $\mathsf{DB}_a.\mathsf{marks} = 1$), the login attempt is successful and R executes $\mathsf{mark}(a,h)$ with probability p_{remark}.

R **monitors** a **at** S: At an arbitrary time, R can ask $S \in \mathcal{S}$ to monitor account a by generating $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathsf{DB}_a.\mathsf{auths})$ and sending $\langle a, \mathsf{DB}_a.\mathsf{salt}, pk, Y\rangle$ to S.

S **receives a monitoring request** $\langle a,s,pk,Y\rangle$ **from** R: S saves $\langle R,a,s,pk,Y\rangle$ locally.

Login attempt at S: For an *unsuccessful* login attempt to an account a using (incorrect) password π, if S holds a monitoring request $\langle R,a,s,pk,Y\rangle$, then it computes $Z \leftarrow \mathsf{pcrRespGen}_{pk}(f(s,\pi),Y)$ and sends $\langle a,Z\rangle$ to R.

R **receives a monitoring response** $\langle a,Z\rangle$: If $\mathsf{pcrReveal}_{sk}(h,Z)$ is *false* for all $h \in \mathsf{DB}_a.\mathsf{auths}$, then R discards $\langle a,Z\rangle$ and returns. Otherwise, let $h \in \mathsf{DB}_a.\mathsf{auths}$ be some hash for which $\mathsf{pcrReveal}_{sk}(h,Z)$ is *true*. R *detects a breach* if $\mathsf{DB}_a.\mathsf{marks}(h) = 0$ and otherwise executes $\mathsf{mark}(a,h)$ with probability p_{remark}.

In the above protocol, the only items received by the monitor S in $\langle a,s,pk,Y\rangle$ are all available to an attacker who breaches R. In this sense, a malicious S gains nothing that an attacker who breaches the target R does not also gain, and in fact gains less, since it learns none of sk, $\mathsf{DB}_a.\mathsf{auths}$, or $\mathcal{S}$. Indeed, the *only* advantage an attacker gains by compromising S in attacking passwords at R is learning the salt $s = \mathsf{DB}_a.\mathsf{salt}$, with which it can precompute information (e.g., rainbow tables [35]) to accelerate its offline attack on $\mathsf{DB}_a.\mathsf{auths}$ if it eventually breaches R. If this possibility is deemed too risky, R can refuse to send s to S in its request but instead permit S to compute $f(s,\pi')$ when needed by interacting with R, i.e., with f being implemented as an oblivious pseudo-random function (OPRF) [17] keyed with s, for which there are efficient implementations (e.g., the DH-OPRF implementation leveraged by OPAQUE [24]). This approach would require extra interaction between S and R per response from S, however, and so we do not consider this alternative further here.

S should authenticate a request $\langle a,s,pk,Y\rangle$ as coming from R, e.g., by requiring that R digitally sign it. Presuming that this digital signing key (different from sk) is vulnerable to capture when R is breached, S should echo each monitoring request back to R upon receiving it. If R receives an echoed request bearing its own signature but that it did not create, it can again detect its own breach. (Recall that we cannot permit the attacker to interfere with communications between R and an uncompromised S and still have R detect its breach.)

In practice, a monitor will not retain a monitoring record forever, as its list of monitoring records—and the resulting cost incurred due to generating responses to them—would only grow. Moreover, it cannot count on R to withdraw its monitoring requests, since R does not retain records of where it has deposited what requests, lest these records be captured when it is breached and the attacker simply avoid monitored accounts. Therefore, presumably a monitor should unilaterally expire each monitoring record after a period of time or in a randomized fashion. We do not investigate specific expiration strategies here, nor do we explore particular strategies for a target to issue monitoring requests over time.

5.4 Security

Several security properties are supported directly by the PCR protocol, which will be detailed in Sec. 6. Here we leverage those properties to argue the security of our design.

No breach detected by unbreached target: If the target R has not been breached, then the PCR protocol will ensure that S must know h for it to generate a Z for which $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* at R. Assuming S cannot guess a $h \in \mathsf{DB}_a.\mathsf{auths}$ without guessing a password π such that $h = f(s,\pi)$ and that (ignoring collisions in f) guessing such a π is infeasible (see (2)), generating such a Z is infeasible for S unless the user provides such a π to S herself. Since the only such π she knows is the one she chose during password registration at R, π is the user-chosen password at a. And, since R has not been breached, the hash of π will still be marked there. As such, R will not detect its own breach.

No risk to security of account at unbreached target: If the target R has not been breached, then the PCR request Y reveals nothing about $\mathsf{DB}_a.\mathsf{auths}$ (except its size) to S. As such, sending a monitoring request poses no risk to the target's account.

No risk to security of account at uncompromised monitor: We now consider the security of the password π for account a at the monitor S (if this account exists at S). First recall

that S generates PCR responses only for *incorrect* passwords attempted in local login attempts for account a; the correct password at S will not be used to generate a response. Moreover, S could even refuse to generate responses for passwords very close to the correct password for a, e.g., the correct password with typos [7]. Second, the PCR protocol ensures that the target R learns nothing about the attempted (and again, incorrect) password π if S is not compromised, unless R included $h = f(s,\pi)$ in the set from which it generated its PCR query Y. In this case, $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* but, again, R already guessed it.

Detection of the target's breach: We now consider the ability of R to detect its own breach by monitoring an account a at an uncompromised monitor S, which is the most nuanced aspect of our protocol's security. Specifically, an attacker who can both repeatedly breach R and simultaneously submit login attempts at an uncompromised S poses the following challenge: Because this attacker can see what hashes for a are presently marked at R, it can be sure to submit to S a password for one of the marked hashes at R, so that the induced PCR response Z will not cause R to detect its own breach. Moreover, if the user reused her password at both R and S, then the attacker will know when it submits this password to S, since S will accept the login attempt.

As such, for R to detect its own breach in these (admittedly extreme) circumstances, the attacker must be unable to submit enough stolen passwords for a to S to submit the user-chosen one with high probability, in the time during which it can repeatedly breach R and before the next legitimate login to a at R or S. To slow the attacker somewhat, R can reduce p_{mark} and p_{remark} to limit the pace of remarkings and, when remarkings occur, the number of hashes that are marked (which are the ones that the attacker can then submit to S).

Two other defenses will likely be necessary, however. First, R can greatly increase the attacker's workload by increasing the number of honeywords per account, say to the thousands or tens of thousands (cf., [28]). Second, since honeywords from R submitted to S will be incorrect for the account a at S, online guessing defenses (account lockout or rate limiting) at S can (and should) be used to slow the attacker's submissions at S. In particular, NIST recommends that a site "limit consecutive failed authentication attempts on a single account to no more than 100" [19, Section 5.2.2], in which case an attacker would be able to eliminate, say, at most 2% of the honeywords for an account with 5000 honeywords stolen from R by submitting them in login attempts at S. Our design shares the need for these defenses with most other methods for using decoy passwords [5, 14, 25, 28, 48]. In particular, if the user reused her password at other sites that permit the attacker to submit passwords stolen from the target without limitation, then the attacker discovering the user's reuse of that password is simply a matter of time, after which the attacker can undetectably take over the account.

5.5 Alternative Designs

The algorithm presented above is the result of numerous iterations, in which we considered and discarded other algorithm variants for remote detection of stuffed honeywords. Here we briefly describe several variants and why we rejected them.

- The target could exclude the known (entered at password reset) or likely (entered in a successful login) user-chosen password π from the monitor request, i.e., $Y \leftarrow \mathsf{pcrQueryGen}_{pk}(\mathsf{DB}_a.\mathsf{auths} \setminus \{f(s,\pi)\})$. In this case, *any* "non-empty" PCR response Z (i.e., $\mathsf{pcrReveal}_{sk}(h,Z)$ returns *true* for some $h \in \mathsf{DB}_a.\mathsf{auths}$) would indicate a breach. However, combining the data breached at the target with Y at a malicious monitor would reveal the password not included in Y as the likely user-chosen one.
- Since a monitor returns a PCR response only for an *incorrect* password attempted locally, the target could plausibly treat any non-empty PCR response as indicating its breach. That is, if the user reused her password, it would not be used to generate a response anyway, and so the response would seemingly have to represent a honeyword attempt. However, if the user did *not* reuse her target password at the monitor, then her mistakenly entering it at the monitor would cause the target to falsely detect its own breach.
- The monitor could return a PCR response for *any* login attempt, correct or not, potentially hastening the target detecting its own breach. However, a PCR request would then present an opportunity for a malicious target to guess $k+1$ passwords for the account at the monitor, and be informed if the user enters one there.
- Any two PCR responses for which $\mathsf{pcrReveal}_{sk}$ returns true with distinct $h, h' \in \mathsf{DB}_a.\mathsf{auths}$ is a reliable breach indicator; one must represent a honeyword. This suggests processing responses in batches, batched either at the monitor or target. However, ensuring that the attacker cannot artificially "fill" batches with repeated password attempts can be complex; batching can delay detection; and batching risks disclosure of a user-chosen password if one might be included in a response and responses are saved in persistent storage (to implement batching).

6 Private Containment Retrieval

Recall that in the algorithm of Sec. 5, upon receiving a monitoring request for an account a from a target, a monitor stores the request locally and uses it to generate a PCR response per failed login attempt to a. Since a response is generated per failed login attempt, it is essential that pcrRespGen be efficient and that the response Z be small. Moreover, considering that a database breach is an uncommon event for a site, we expect that most of the time, the response would be generated using a password that is not in the set used by the target to generate the monitoring request. (Indeed, barring a database breach at the target, this should never happen unless

the user enters at the monitor her password for her account at the target.) So, in designing a PCR, we place a premium on ensuring that pcrReveal is very efficient in this case.

6.1 Comparison to Related Protocols

Since the monitor's input to pcrRespGen is a singleton set (i.e., a hash), a natural way to achieve the functionality of a private containment retrieval is to leverage existing private set intersection (PSI) protocols, especially *unbalanced* PSIs that are designed for the use case where two parties have sets of significantly different sizes [8, 26, 27, 39, 42]. Among these protocols, those based on oblivious pseudo-random functions (OPRFs) [26, 27, 39, 42] require both parties to obliviously agree on a privacy-preserving but deterministic way of representing their input sets so at least one party can compare and output elements in the intersection, if any. To achieve this, both parties participate in at least one round of interaction (each of at least two messages) during an online phase, and so would require more interaction in our context than our framework as defined in Sec. 5. Chen et al. [8] proposed a PSI protocol with reduced communication, but at the expense of leveraging fully homomorphic encryption. And, interestingly, these unbalanced PSI protocols, as well as private membership tests (e.g., [34, 38, 46, 47]), are all designed for the case where the target has the smaller set and the monitor has the larger one, which is the opposite of our use case.

Among other PSI protocols that require no more than one round of interaction, that of Davidson and Cid [12] almost meets the requirements of our framework on the monitor side: its monitor's computation complexity and response message size are manageable and, more importantly, constant in the target's set size. However, in their design, the query message size depends on the false-positive probability (of the containment test) due to their use of Bloom filters and bit-by-bit encryption, while ours is also constant in the false-positive probability. If applied in our context, their design would generate a significantly larger query and so significantly greater storage overhead at the monitor than ours, especially when a relatively low false-positive probability is enforced. For example, to achieve a 2^{-96} false-positive probability, their query message would include $\approx 131\times$ more ciphertexts than ours.

Our PCR protocol, on the other hand, is designed specifically for the needs of our framework, where the target has a relatively large set and the monitor's set is smaller (in fact, of size 1) that keeps changing over time. Our protocol requires only one message from the monitor to the target. In addition, the response message computation time and output size is constant in the target's set size. We also constructed our algorithm so that determining that $\mathsf{pcrReveal}_{sk}(h, Z)$ is *false* for all $h \in \mathsf{DB}_a.\mathsf{auths}$, which should be the common case, costs much less time than finding the $h \in \mathsf{DB}_a.\mathsf{auths}$ for which $\mathsf{pcrReveal}_{sk}(h, Z)$ is *true*. We demonstrate these properties empirically in Sec. 6.5. While our protocol leverages tools (e.g., partially homomorphic encryption, cuckoo filters) utilized in other protocols (e.g., [47]), ours does so in a novel way and with an eye toward our specific goals here.

6.2 Building Blocks

Partially homomorphic encryption: Our protocol builds on a partially homomorphic encryption scheme $\mathcal{E}$ consisting of algorithms Gen, Enc, isEq, and $+_{[\cdot]}$.

- 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 determines a prime r for which the plaintext space for encrypting with pk is the finite field $\langle \mathbb{Z}_r, +, \times\rangle$ where $+$ and $\times$ are addition and multiplication modulo r, respectively. For clarity below, we denote the additive identity by $\mathbf{0}$, the multiplicative identity by $\mathbf{1}$, and the additive inverse of $m \in \mathbb{Z}_r$ by $-m$. The value of pk also determines a ciphertext space $C_{pk} = \bigcup_m C_{pk}(m)$, where $C_{pk}(m)$ denotes the ciphertexts for plaintext m.
- Enc is a randomized algorithm that on input public key pk and a plaintext m, outputs a ciphertext $c \leftarrow \mathsf{Enc}_{pk}(m)$ chosen uniformly at random from $C_{pk}(m)$.
- isEq is a deterministic algorithm that on input a private key sk, plaintext m, and ciphertext $c \in C_{pk}$, outputs a Boolean $z \leftarrow \mathsf{isEq}_{sk}(m, c)$ where $z = true$ iff $c \in C_{pk}(m)$.
- $+_{[\cdot]}$ is a randomized algorithm that, on input a public key pk and ciphertexts $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, outputs a ciphertext $c \leftarrow c_1 +_{pk} c_2$ chosen uniformly at random from $C_{pk}(m_1 + m_2)$.

Note that our protocol does not require an efficient decryption capability. Nor does the encryption scheme on which we base our empirical evaluation in Sec. 6.5, namely "exponential ElGamal" (e.g., [10]), support one. It does, however, support an efficient isEq calculation.

Given this functionality, it will be convenient to define a few additional operators involving ciphertexts. Below, "$\mathbf{Y} \stackrel{d}{=} \mathbf{Y}'$" denotes that random variables $\mathbf{Y}$ and $\mathbf{Y}'$ are distributed identically; "$\mathbf{Z} \in (\mathcal{X})^{\alpha\times\alpha'}$" means that $\mathbf{Z}$ is an α-row, α'-column matrix of elements in the set $\mathcal{X}$; and "$(\mathbf{Z})_{i,j}$" denotes the row-i, column-j element of the matrix $\mathbf{Z}$.

- $\sum_{pk}$ denotes summing a sequence using $+_{pk}$, i.e.,

$$\sum_{k=1}^{z}{}_{pk}\ c_k \stackrel{d}{=} c_1 +_{pk} c_2 +_{pk} \ldots +_{pk} c_z$$

- If $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ and $\mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{C} +_{pk} \mathbf{C}' \in (C_{pk})^{\alpha\times\alpha'}$ is the result of component-wise addition using $+_{pk}$, i.e., so that

$$\left(\mathbf{C} +_{pk} \mathbf{C}'\right)_{i,j} \stackrel{d}{=} (\mathbf{C})_{i,j} +_{pk} (\mathbf{C}')_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha\times\alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$, then $\mathbf{M} \circ_{pk} \mathbf{C} \in (C_{pk})^{\alpha\times\alpha'}$ is the result of Hadamard (i.e., component-wise) "scalar

multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} \circ_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{(\mathbf{M})_{i,j}}{}_{pk} \; (\mathbf{C})_{i,j}$$

- If $\mathbf{M} \in (\mathbb{Z}_r)^{\alpha \times \alpha'}$ and $\mathbf{C} \in (C_{pk})^{\alpha' \times \alpha''}$, then $\mathbf{M} *_{pk} \mathbf{C} \in (C_{pk})^{\alpha \times \alpha''}$ is the result of standard matrix multiplication using $+_{pk}$ and "scalar multiplication" using repeated application of $+_{pk}$, i.e., so that

$$\left(\mathbf{M} *_{pk} \mathbf{C}\right)_{i,j} \stackrel{d}{=} \sum_{k=1}^{\alpha'}{}_{pk} \; \sum_{k'=1}^{(\mathbf{M})_{i,k}}{}_{pk} \; (\mathbf{C})_{k,j}$$

Cuckoo filters: A cuckoo filter [16] is a set representation that supports insertion and deletion of elements, as well as testing membership. The cuckoo filter uses a "fingerprint" function $\mathsf{fp} : \{0,1\}^* \to F$ and a hash function $\mathsf{hash} : \{0,1\}^* \to [\beta]$, where for an integer z, the notation "$[z]$" denotes $\{1,\ldots,z\}$, and where β is a number of "buckets". We require that $F \subseteq \mathbb{Z}_r \setminus \{\mathbf{0}\}$ for any r determined by $\langle pk, sk\rangle \leftarrow \mathsf{Gen}(1^\kappa)$. For an integer bucket "capacity" χ, the cuckoo filter data structure is a β-row, χ-column matrix $\mathbf{X}$ of elements in $\mathbb{Z}_r$, i.e., $\mathbf{X} \in (\mathbb{Z}_r)^{\beta \times \chi}$. Then, the membership test $e \stackrel{?}{\in} \mathbf{X}$ returns *true* if and only if there exists $j \in [\chi]$ such that either

$$(\mathbf{X})_{\mathsf{hash}(e),j} = \mathsf{fp}(e) \qquad \text{or} \qquad (4)$$

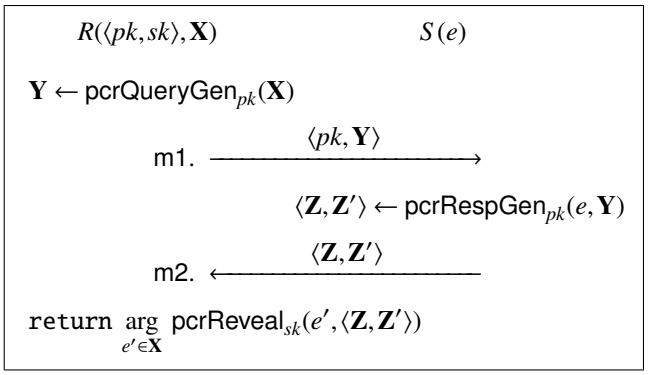
$$(\mathbf{X})_{\mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e)),j} = \mathsf{fp}(e) \qquad (5)$$

Cuckoo filters permit false positives (membership tests that return *true* for elements not previously added or already removed) with a probability that, for fixed χ, can be decreased by increasing the size of F [16].

6.3 Protocol Description

Our PCR protocol is detailed in Fig. 5. Fig. 5a shows the message flow, which conforms with the protocol's use in our algorithm of Sec. 5, and Fig. 5b shows the procedures. In this protocol, the target R has a public-key pair $\langle pk, sk\rangle$ for the encryption scheme defined in Sec. 6.2 and a cuckoo filter $\mathbf{X}$. In the context of Sec. 5, $\mathbf{X}$ holds the password hashes (for k honeywords and one user-chosen password) for an account. $\mathsf{pcrQueryGen}_{pk}$ simply encrypts each element of the cuckoo filter individually and returns this matrix $\mathbf{Y}$ as the PCR query. R sends pk and $\mathbf{Y}$ to the monitor S in message m1.

S has an input e—which is the hash of a password entered in a failed login attempt, in the algorithm of Sec. 5—and invokes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$ to produce a response $\langle \mathbf{Z}, \mathbf{Z}'\rangle$. pcrRespGen first generates a $2 \times \beta$ matrix $\mathbf{Q}$ with $\mathbf{1}$ at the indices i_1 and i_2 in the first and second rows, respectively (lines s2–s4), and $\mathbf{0}$ elsewhere, and a $2 \times \chi$ matrix $\mathbf{F}$ that contains encryptions of $-\mathsf{fp}(e)$ (lines s5–s6). Referring to line s8, the operation $\mathbf{Q} *_{pk} \mathbf{Y}$ thus produces the two buckets (rows) of $\mathbf{Y}$ that could include a ciphertext of $\mathsf{fp}(e)$ (ignoring collisions in fp), and $\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}$ produces a matrix where



(a) Message flow

$\mathsf{pcrQueryGen}_{pk}(\mathbf{X})$:
r1. abort if $\mathbf{X} \notin (\mathbb{Z}_r)^{\beta \times \chi}$
r2. $\forall i \in [\beta], j \in [\chi] : (\mathbf{Y})_{i,j} \leftarrow \mathsf{Enc}_{pk}((\mathbf{X})_{i,j})$
r3. return $\mathbf{Y}$

$\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$:
s1. abort if $\mathbf{Y} \notin (C_{pk})^{\beta \times \chi}$
s2. $i_1 \leftarrow \mathsf{hash}(e)$
s3. $i_2 \leftarrow \mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e))$
s4. $\forall i \in [2], j \in [\beta] :$
$$(\mathbf{Q})_{i,j} \leftarrow \begin{cases} \mathbf{1} & \text{if } \langle i, j\rangle \in \{\langle 1, i_1\rangle, \langle 2, i_2\rangle\} \\ \mathbf{0} & \text{otherwise} \end{cases}$$
s5. $f \leftarrow \mathsf{Enc}_{pk}(-\mathsf{fp}(e))$
s6. $\forall i \in [2], j \in [\chi] : (\mathbf{F})_{i,j} \leftarrow f$
s7. $\forall i \in [2], j \in [\chi] : (\mathbf{M})_{i,j} \stackrel{\$}{\leftarrow} \mathbb{Z}_r \setminus \{\mathbf{0}\}$
s8. $\mathbf{Z} \leftarrow \mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)$
s9. $f' \leftarrow \mathsf{Enc}_{pk}(\mathsf{fp}'(e))$
s10. $\forall i \in [2], j \in [\chi] : (\mathbf{F}')_{i,j} \leftarrow f'$
s11. $\forall i \in [2], j \in [\chi] : (\mathbf{M}')_{i,j} \stackrel{\$}{\leftarrow} \mathbb{Z}_r$
s12. $\mathbf{Z}' \leftarrow \left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}'$
s13. return $\langle \mathbf{Z}, \mathbf{Z}'\rangle$

$\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}'\rangle)$:
r4. return *false* if $\mathbf{Z} \notin (C_{pk})^{2 \times \chi} \vee \mathbf{Z}' \notin (C_{pk})^{2 \times \chi}$
r5. $\langle \hat{i}, \hat{j}\rangle \leftarrow \arg\limits_{\langle i,j\rangle} \mathsf{isEq}_{sk}(\mathbf{0}, (\mathbf{Z})_{i,j})$
r6. return *false* if $\langle \hat{i}, \hat{j}\rangle = \langle \perp, \perp\rangle$
r7. return $\mathsf{isEq}_{sk}(\mathsf{fp}'(e'), (\mathbf{Z}')_{\hat{i},\hat{j}})$

(b) Procedures

Figure 5: Private Containment Retrieval protocol, with matrices $\mathbf{X} \in (\mathbb{Z}_r)^{\beta \times \chi}$; $\mathbf{Y} \in (C_{pk})^{\beta \times \chi}$; $\mathbf{Q} \in (\mathbb{Z}_r)^{2 \times \beta}$; $\mathbf{M}, \mathbf{M}' \in (\mathbb{Z}_r)^{2 \times \chi}$; $\mathbf{F}, \mathbf{F}', \mathbf{Z}, \mathbf{Z}' \in (C_{pk})^{2 \times \chi}$.

that ciphertext (if any) has been changed to a ciphertext of $\mathbf{0}$. This ciphertext of $\mathbf{0}$ remains after multiplying this matrix component-wise by the random matrix $\mathbf{M}$ to produce $\mathbf{Z}$. The

remaining steps (lines s9–s12) simply rerandomize $\mathbf{Z}$ and transform this ciphertext of $\mathbf{0}$ to a ciphertext of $\mathsf{fp}'(e)$ in $\mathbf{Z}'$, for a fingerprint function $\mathsf{fp}' : \{0,1\}^* \to F$ that is "unrelated" to fp. (We will model fp' as a random oracle [4] for the security argument in Sec. 6.4.) Rerandomization using $\mathbf{M}'$ in the creation of $\mathbf{Z}'$ is essential to protect the privacy of e if $e \notin \mathbf{X}$, since without rerandomizing, the component-wise differences of the plaintexts of $\mathbf{Z}$ and $\mathbf{Z}'$ would reveal $\mathsf{fp}'(e)$ to R.

For (an artificially small) example, suppose $\beta = 3, \chi = 1$, and that the monitor S invokes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$ where $i_1 = \mathsf{hash}(e) = 3$ and $i_2 = \mathsf{hash}(e) \oplus \mathsf{hash}(\mathsf{fp}(e)) = 2$. Furthermore, suppose that $(\mathbf{X})_{i_1,1} \stackrel{d}{=} \mathsf{Enc}_{pk}(e)$. Then,

$$\mathbf{Q} *_{pk} \mathbf{Y} \stackrel{d}{=} \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix} *_{pk} \begin{bmatrix} c_1 \\ c_2 \\ \mathsf{Enc}_{pk}(e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(e) \\ c_2 \end{bmatrix}$$

and so

$$\begin{aligned} &\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F} \\ &\stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(e) \\ c_2 \end{bmatrix} +_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(-e) \\ \mathsf{Enc}_{pk}(-e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_2 - e) \end{bmatrix} \end{aligned}$$

where $c_2 \in C_{pk}(m_2)$. Assuming $m_2 \neq e$, we then have

$$\begin{aligned} \mathbf{Z} &\stackrel{d}{=} \mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right) \\ &\stackrel{d}{=} \begin{bmatrix} m_3 \\ m_4 \end{bmatrix} \circ_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_2 - e) \end{bmatrix} \stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_5) \end{bmatrix} \end{aligned}$$

where $m_3, m_4 \stackrel{\$}{\leftarrow} \mathbb{Z}_r \setminus \{\mathbf{0}\}$ and so $m_5 \neq \mathbf{0}$. Finally,

$$\begin{aligned} \mathbf{Z}' &\stackrel{d}{=} \left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}' \\ &\stackrel{d}{=} \left(\begin{bmatrix} m_6 \\ m_7 \end{bmatrix} \circ_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathbf{0}) \\ \mathsf{Enc}_{pk}(m_5) \end{bmatrix}\right) +_{pk} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \\ \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \end{bmatrix} \\ &\stackrel{d}{=} \begin{bmatrix} \mathsf{Enc}_{pk}(\mathsf{fp}'(e)) \\ \mathsf{Enc}_{pk}(m_8) \end{bmatrix} \end{aligned}$$

where $m_6, m_7 \stackrel{\$}{\leftarrow} \mathbb{Z}_r$ and so m_8 is uniformly random in $\mathbb{Z}_r$.

Given this structure of $\langle \mathbf{Z}, \mathbf{Z}' \rangle$, $\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ must simply find the location $\langle \hat{i}, \hat{j} \rangle$ where $\mathbf{Z}$ holds a ciphertext of $\mathbf{0}$ (line r5) and, unless there is none (line r6), return whether the corresponding location in $\mathbf{Z}'$ is a ciphertext of $\mathsf{fp}'(e')$ (line r7).

6.4 Security

The use of this protocol to achieve the security arguments of Sec. 5.4 depends on the PCR protocol achieving certain key properties. We present these properties below.

Security against a malicious monitor: When the target R is not breached, our primary goals are twofold. First, we need to show that monitoring requests do not weaken the security of R's accounts or, in other words, that the request $\mathbf{Y}$ does not leak information about $\mathbf{X}$ (except its size). This is straightforward, however, since in this protocol S observes only ciphertexts $\mathbf{Y}$ and the public key pk with which these ciphertexts were created. (The target R need not, and should not, divulge the result of the protocol to the monitor S.) As such, the privacy of $\mathbf{X}$ reduces trivially to the IND-CPA security [3] of the encryption scheme.

The second property that we require of this protocol is that a malicious monitor be unable to induce the target to evaluate $\mathsf{pcrReveal}_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ to *true* for any $e' \in \mathbf{X}$ unless the monitor knows e'. That is, in the context of Sec. 5, we want to ensure that the monitor must have received (a password that hashes to) e' in a login attempt, as otherwise the monitor might cause the target to falsely detect its own breach. This is straightforward to argue in the random oracle model [4], however, since if fp' is modeled as a random oracle, then to create a ciphertext $(\mathbf{Z}')_{i,j} \in C_{pk}(\mathsf{fp}'(e'))$ with non-negligible probability in the output length of fp', S must invoke the fp' oracle with e' and so must "know" it.

Security against a malicious target: Though our threat model in Sec. 5.1 does not permit a malicious target for the purposes of designing an algorithm for it to detect its own breach, a monitor will participate in this protocol only if doing so does not impinge on the security of its own accounts, even in the case where the target is malicious. The security of the monitor's account a is preserved since if the monitor correctly computes $\mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$, then the output $\langle \mathbf{Z}, \mathbf{Z}' \rangle$ carries information about e only if some $(\mathbf{Y})_{i,j} \in C_{pk}(\mathsf{fp}(e))$, i.e., only if the target already enumerated this password among the $k+1$ in $\mathbf{Y}$ (ignoring collisions in fp). That is, even a malicious target learns nothing about e from the response computed by an honest monitor unless the target already guessed e (or more precisely, $\mathsf{fp}(e)$).

This reasoning requires that pk is a valid public key for the cryptosystem, and so implicit in the algorithm description in Fig. 5 is that the monitor verifies this. This verification is trivial for the cryptosystem with which we instantiate this protocol in Sec. 6.5.

Proposition 1. *Given* $\langle pk, \mathbf{Y} \rangle$ *and* e *where* $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ *for each* $i \in [\beta]$, $j \in [\chi]$, *if the* monitor *correctly computes* $\langle \mathbf{Z}, \mathbf{Z}' \rangle \leftarrow \mathsf{pcrRespGen}_{pk}(e, \mathbf{Y})$, *then*

$$\mathbb{P}\left((\mathbf{Z})_{i,j} \in C_{pk}(m) \wedge (\mathbf{Z}')_{i,j} \in C_{pk}(m')\right) = \frac{1}{r(r-1)}$$

for any $i \in [2]$, $j \in [\chi]$, $m \in \mathbb{Z}_r \setminus \{\mathbf{0}\}$, *and* $m' \in \mathbb{Z}_r$.

Proof. Since each $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ by assumption, the constructions of $\mathbf{Q}$ and $\mathbf{F}$ imply that $\left(\mathbf{Q} *_{pk} \mathbf{Y}\right)_{i,j} \notin C_{pk}(\mathsf{fp}(e))$ and so $\left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)_{i,j} \notin C_{pk}(\mathbf{0})$ for any $i \in [2]$, $j \in [\chi]$. Then, since $(\mathbf{M})_{i,j}$ is independently and uniformly distributed in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$, it follows that $(\mathbf{Z})_{i,j} = \left(\mathbf{M} \circ_{pk} \left(\left(\mathbf{Q} *_{pk} \mathbf{Y}\right) +_{pk} \mathbf{F}\right)\right)_{i,j} \in C_{pk}(m)$ for m distributed uniformly in $\mathbb{Z}_r \setminus \{\mathbf{0}\}$, as well. Finally, since $(\mathbf{M}')_{i,j}$ is independently and uniformly distributed

in $\mathbb{Z}_r$, we know that $\left(\left(\mathbf{M}' \circ_{pk} \mathbf{Z}\right) +_{pk} \mathbf{F}'\right)_{i,j} \in C_{pk}(m')$ for m' distributed uniformly in $\mathbb{Z}_r$. □

The proposition above shows that the *plaintexts* in the response are uniformly distributed if $(\mathbf{Y})_{i,j} \notin C_{pk}(\mathsf{fp}(e))$. The following proposition also points out that the *ciphertexts* are uniformly distributed.

Proposition 2. *If the* monitor *follows the protocol, then*

$$\mathbb{P}((\mathbf{Z})_{i,j} = c \mid (\mathbf{Z})_{i,j} \in C_{pk}(m)) = \frac{1}{|C_{pk}(m)|}$$

$$\mathbb{P}((\mathbf{Z}')_{i,j} = c \mid (\mathbf{Z}')_{i,j} \in C_{pk}(m)) = \frac{1}{|C_{pk}(m)|}$$

for any $i \in [2]$, $j \in [\chi]$, $m \in \mathbb{Z}_r$, *and* $c \in C_{pk}(m)$.

Proof. This is immediate since $+_{pk}$ ensures that for $c_1 \in C_{pk}(m_1)$ and $c_2 \in C_{pk}(m_2)$, $c_1 +_{pk} c_2$ outputs a ciphertext c chosen uniformly at random from $C_{pk}(m_1 + m_2)$. □

6.5 Performance

We implemented the protocol of Fig. 5 to empirically evaluate its computation and communication costs. The implementation is available at https://github.com/k3coby/pcr-go.

Parameters: In our implementation, we instantiated the underlying cuckoo filter with bucket size $\chi = 4$, as recommended by Fan et al. [16]. We chose fingerprints of length 224 bits to achieve a low false-positive probability, i.e., about 2^{-221}. For the underlying partially homomorphic encryption scheme, we chose exponential ElGamal (e.g., see [10]) implemented in the elliptic-curve group secp256r1 [6] to balance performance and security (roughly equivalent to 3072-bit RSA security or 128-bit symmetric security).

Experiment setup: Our prototype including cuckoo filters and cryptography, were implemented in Go. We ran the experiments reported below on two machines with the same operating system and hardware specification: Ubuntu 20.04.1, AMD 8-core processor (2.67GHz), and 72GiB RAM. These machines played the role of the target and the monitor. We report all results as the means of 50 runs of each experiment and report relative standard deviations (rsd) in the figure captions.

Results: We report the computation time of pcrQueryGen, pcrRespGen, and pcrReveal in Fig. 6. As shown in Fig. 6a, the computation time of pcrQueryGen is linear in the target's set size (i.e., $k+1$). One takeaway here is that even if the number of honeywords is relatively large, e.g., $k = 1000$, it only takes the target about 100ms to generate a query with four logical CPU cores. Moreover, since a query is generated only when choosing to monitor an account at a monitor, the target can choose when to incur this cost. Fig. 6b shows that the computational cost of PCR response generation is essentially

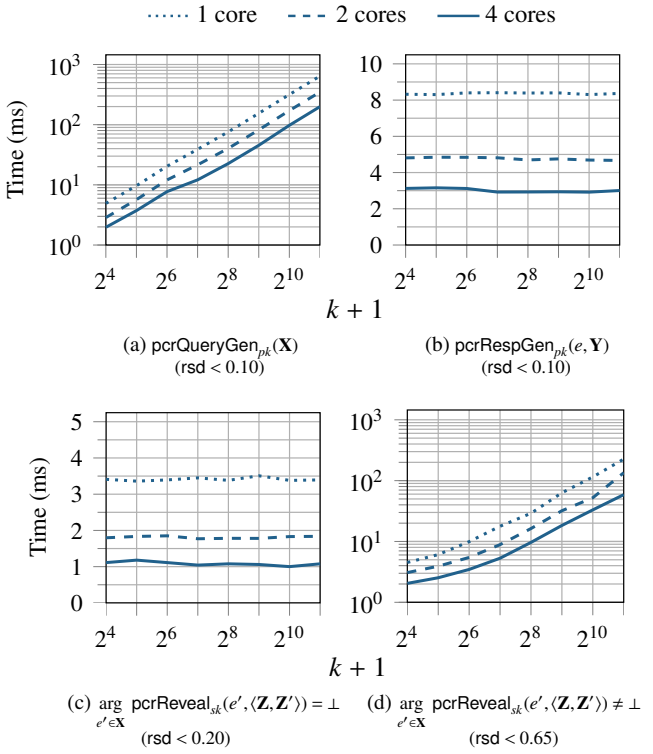


Figure 6: Runtimes of pcrQueryGen$_{pk}(\mathbf{X})$, pcrRespGen$_{pk}(e, \mathbf{Y})$, and $\arg_{e' \in \mathbf{X}}$ pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ when $= \perp$ and when $\neq \perp$, as functions of $k+1$ with varying numbers of logical CPU cores.

unchanged regardless of k. This is important so that the computational burden on the monitors does not increase even if the target grows its number of honeywords per account. Another observation from Fig. 6b is that it only takes less than 9ms for the monitor, with even a single logical core, to produce a response when a failed login attempt occurs.

The computation time of $\arg_{e' \in \mathbf{X}}$ pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle)$ is shown in Figs. 6c–d in two separate cases: when for all $e' \in \mathbf{X}$ is pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{false}$ (and so the result $= \perp$, Fig. 6c) and when for some $e' \in \mathbf{X}$, pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{true}$ (i.e., the result $\neq \perp$, Fig. 6d). We report these cases separately since they have significantly different performance characteristics. Again, we expect the former to be the common case. This operation takes constant time in the former case, since the target needs only to test if any of the 2χ ciphertexts (e.g., 8 ciphertexts with $\chi = 4$) are encryptions of zeros. In our experiments for Fig. 6d, the element e' for which pcrReveal$_{sk}(e', \langle \mathbf{Z}, \mathbf{Z}' \rangle) = \mathit{true}$ was randomly picked from $\mathbf{X}$, and the target immediately returned once e' was identified. So the position of e' in $\mathbf{X}$ has a large impact on the computation time for each run, yielding an increased relative standard deviation. Since the target on average performs approximately $\frac{k+1}{2}$ isEq operations to identify e' in this case, the cost is linear in the target's set size, as shown in Fig. 6d.

As shown in Fig. 7, the query (message m1) is of size linear in the target's set size, while the response (m2) size is constant (≈ 1KB). These communication and storage costs are quite manageable. For example, even 100,000 monitoring requests would require only about 32GB of storage at the monitor when $k+1 = 4096$.

Performance example: To put these performance results in context, consider the STRONTIUM credential harvesting attacks launched against over 200 organizations from September 2019 to June 2020. Microsoft [33] reported that their most aggressive attacks averaged 335 login attempts per hour per account for hours or days at a time, and that organizations targeted in these attacks saw login attempts on an average of 20% of their total accounts. So, if all of a target's monitors had been attacked simultaneously by STRONTIUM, then 20% of the target's monitoring requests would have been triggered to generate responses to the target. Suppose that in the steady state, the target had maintained a total of x active monitoring requests across all of its monitors.

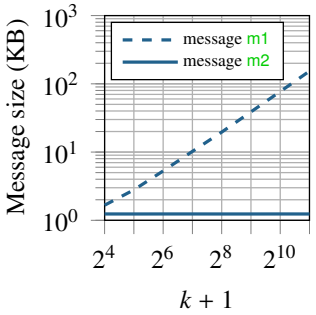


Figure 7: Message size as a function of R's password set size for a (rsd < 0.01)

We now consider two scenarios. First, if monitors would not have limited the number of incorrect logins per account that induced monitoring responses, then each triggered monitoring request would have induced an average of 335 monitoring responses per hour. As such, the target would have averaged $(20\%)(335)(x) = 67x$ monitoring responses per hour, or $\frac{67}{3600}x$ monitoring responses per second. Since in our experiments, processing each monitoring response averaged ≈ 0.002s on a 2-core computer (Fig. 6c), this computer could have sustained the processing load that would have been induced on the target provided that $x < \frac{3600}{(0.002)(67)} \approx 26{,}865$ monitoring requests. Even if all x monitoring requests had been active at the same monitor, this monitor (using the same type of computer) could have sustained generating responses as long as $x < \frac{3600}{(0.005)(67)} \approx 10{,}746$, since generating responses on a 2-core computer averaged ≈ 0.005s (Fig. 6b). If the x monitoring requests had been spread across even only three monitors, however, the bottleneck would have been the target.

The second scenario we consider is one in which monitors would have limited the number of incorrect logins per account that induced a monitoring response, as recommended in Sec. 5.4. If each monitor would have limited the number of consecutive incorrect logins (and so monitoring responses) to 100 per account [19, Section 5.2.2], then the target would have averaged $(20\%)(100)(x) = 20x$ monitoring responses per hour and, using reasoning similar to that above, could have absorbed the induced processing load provided that $x < \frac{3600}{(0.002)(20)} = 90{,}000$ monitoring requests. And, in the extreme case that the same monitor held all x monitoring requests, the monitor (using the same type of computer) could have sustained generating responses for $x < \frac{3600}{(0.005)(20)} = 36{,}000$ monitoring requests.

7 Discussion

In this section we discuss various risks associated with Amnesia. The first is a general risk associated with Amnesia, and the others are specific to the distributed defenses against credential stuffing proposed in Sec. 5.

Password reset: Because detection happens in Amnesia when the legitimate user logs into her account at the target after the attacker has, the attacker can try to interfere with breach detection by changing the account password upon gaining access to the account. The legitimate user will be locked out of her account and so will presumably be forced to reset her password, but this will not serve as unequivocal evidence of the breach; after all, users reset their passwords all the time, due to simply forgetting them [22]. As such, target sites should utilize a backup authentication method (e.g., a code sent to a contact email or phone for the account) before enabling password reset.

Denial-of-service attacks: There are mainly two potential ways of launching denial-of-service (DoS) attacks against a target: one in which the attacker submits login attempts at a high rate to a benign monitor to induce monitor responses to the target, and one in which a malicious monitor directly sends responses to the target at a high rate. The former DoS should be difficult for an attacker to perform effectively, since it requires the attacker to know or predict where the target will send monitoring requests and for what accounts. While we have not prescribed a specific strategy by which a target deploys monitor requests, such a strategy would need to be unpredictable; otherwise, rather than using this knowledge to conduct DoS, the attacker could instead use it to sidestep the accounts at sites while they are monitored, to avoid alerting the target to its breach. Another reason the former DoS will likely be ineffective is that, as discussed in Sec. 5.4, a target that can be breached repeatedly must rely on monitors to slow stuffing attacks to identify a user's reused password. These defenses will correspondingly help defend the target from this type of DoS. The latter DoS against a target, i.e., by a malicious monitor, would alert the target that this monitor is either conducting DoS or not implementing these slowing defenses. In either case, the target can remove this monitor from its list of monitors and drop responses from it.

As any site, a monitor should deploy state-of-the-art defenses against online guessing attacks which, in turn, can benefit targets as discussed above and in Sec. 5.4. The primary DoS risk introduced by Amnesia to monitors is the stor-

age overhead of monitoring requests, though as discussed in Sec. 6.5, this need not be substantial. Moreover, the monitor has discretion to expire or discard monitoring requests as needed, and so can manage these costs accordingly.

User privacy: Privacy risks associated with remote monitoring of a user account include revealing to monitors the targets at which a user has an account and revealing to a target when a user attempts to log into a monitor. To obscure the former information, a target could send (ineffective) monitoring requests for accounts that have not been registered locally, e.g., using inputs $\mathcal{X}$ to pcrQueryGen consisting of uniformly random values. The latter information will likely be naturally obscured since failed login attempts to an account at a monitor due to automated attacks (online guessing, credential stuffing, etc.) would trigger PCR responses even if the account does not exist at the monitor and can outnumber failed login attempts by a legitimate user even if it does [41]. In addition, a monitor could further obscure user login activity on accounts for which it holds monitoring requests by generating monitoring responses at arbitrary times using uniformly random passwords.

Incentives to monitor accounts: Given the overheads that monitoring requests induce on monitors, it is natural to question whether monitors have adequate incentives to perform monitoring for targets and, if so, at what rates. Moreover, these questions are complicated by site-specific factors.

On the one hand, large disparities in the numbers of accounts at various sites that might participate in a monitoring ecosystem could result in massive imbalances in the monitoring loads induced on sites. For example, issuing monitoring requests at a rate to induce expected steady-state monitoring of, say, even 10% of Gmail users' accounts, each at only a single monitor, would impose ≈ 180 million monitoring requests across monitors on an ongoing basis [37]. This could easily induce more load on monitors than they would find "worth it" for participating in this ecosystem.

On the other hand, dependencies among sites might justify substantial monitoring investment by the web community as a whole. For example, the benefit to internet security in the large for detecting a breach of Google's credential database quickly is considerable: as one of the world's largest email providers, it is trusted for backup authentication and account recovery (via email challenges) for numerous accounts at other sites. Indeed, as discussed above, some form of backup authentication *needs* to be a gatekeeper to resetting account passwords at a site who wishes to itself participate as target in our design, to ensure it will detect its own breach reliably. Such a site might thus be willing to participate as a monitor for numerous accounts of a target site on which many of its accounts depend for backup authentication.

Balancing these considerations to produce a viable monitoring ecosystem is a topic of ongoing research. We recognize, however, that establishing and sustaining such an ecosystem might benefit from additional inducements, e.g., monetary payments from targets to monitors or savings in the form of reduced insurance premiums for sites that agree to monitor for one another.

8 Conclusion

In this paper, we have proposed Amnesia, a methodology for using honeywords to detect the breach of a site *without relying on any secret persistent state*. Our algorithm remains effective to detect breaches even against attackers who repeatedly access the target site's persistent storage, including any long-term cryptographic keys. We extended this algorithm to allow the target site to detect breaches when the attacker tries to differentiate a (potentially reused) real password from honeywords by stuffing them at other sites. We realized this remote detection capability using a new private containment retrieval protocol with rounds, computation, communication, and storage costs that work well for our algorithm. We expect that, if deployed, Amnesia could effectively shorten the time between credential database breaches and their discovery.

Acknowledgments

We are grateful to our shepherd, Patrick Traynor, and to the anonymous reviewers for their constructive feedback. This research was supported in part by grant numbers 2040675 from the National Science Foundation and W911NF-17-1-0370 from the Army Research Office. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, Army Research Office, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notices herein.

References

[1] Akshima, D. Chang, A. Goel, S. Mishra, and S. K. Sanadhya, "Generation of secure and reliable honeywords, preventing false detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 5, pp. 757–769, 2019.

[2] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, "ErsatzPasswords: Ending password cracking and detecting password leakage," in *31st Annual Computer Security Applications Conference*, Dec. 2015, pp. 311–320.

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology – CRYPTO 1998*, ser. Lecture Notes in Computer Science, vol. 1462, Aug. 1998.

[4] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *1st ACM Conference on Computer and Communications Security*, Nov. 1993.

[5] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 6345, Sep. 2010.

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

[7] R. Chatterjee, A. Athayle, D. Akhawe, A. Juels, and T. Ristenpart, "pASSWORD tYPOS and how to correct them securely," in *37th IEEE Symposium on Security and Privacy*, May 2016, pp. 799–818.

[8] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *24nd ACM Conference on Computer and Communications Security*, Oct. 2017.

[9] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *29th IEEE Computer Security Foundations Symposium*, Jun. 2016.

[10] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," in *Advances in Cryptology – EUROCRYPT '97*, ser. Lecture Notes in Computer Science, vol. 1233, 1997, pp. 103–118.

[11] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *ISOC Network and Distributed System Security Symposium*, 2014.

[12] A. Davidson and C. Cid, "An efficient toolkit for computing private set operations," in *22nd Australasian Conference on Information Security and Privacy*, ser. Lecture Notes in Computer Science, vol. 10343, Jul. 2017.

[13] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren, "Tripwire: Inferring internet site compromise," in *17th Internet Measurement Conference*, Nov. 2017.

[14] I. Erguler, "Achieving flatness: Selecting the honeywords from existing user passwords," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, 2016.

[15] A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, and T. Ristenpart, "The Pythia PRF service," in *24th USENIX Security Symposium*, Aug. 2015, pp. 547–562.

[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *10th ACM Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[17] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *2nd Theory of Cryptography Conference*, ser. Lecture Notes in Computer Science, vol. 3378, Feb. 2005.

[18] D. Freeman, S. Jain, M. Dürmuth, B. Biggio, and G. Giacinto, "Who are you? A statistical approach to measuring user authenticity," in *23rd ISOC Network and Distributed System Security Symposium*, Feb. 2016.

[19] 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.

[20] C. G. Günther, "An identity-based key-exchange protocol," in *Advances in Cryptology – EUROCRYPT '89*, ser. Lecture Notes in Computer Science, vol. 434, Apr. 1989, pp. 29–37.

[21] C. Herley and D. Florêncio, "Protecting financial institutions from brute-force attacks," in *23rd International Conference on Information Security*, ser. IFIP Advances in Information and Communication Technology, vol. 278, Sep. 2008, pp. 681–685.

[22] HYPR, "New password study by HYPR finds 78% of people had to reset a password they forgot in past 90 days," https://www.hypr.com/hypr-password-study-findings/, Dec. 2019.

[23] IBM Security, "Cost of a data breach report 2020," https://www.ibm.com/security/digital-assets/cost-data-breach-report/, 2020.

[24] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks," in *Advances in Cryptology – EUROCRYPT 2018*, ser. Lecture Notes in Computer Science, vol. 10822, 2018, pp. 456–486.

[25] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *20th ACM Conference on Computer and Communications Security*, Nov. 2013.

[26] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *28th USENIX Security Symposium*, Aug. 2019.

[27] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, "Private set intersection for unequal set sizes with mobile applications," *17th Privacy Enhancing Technologies Symposium*, vol. 2017, no. 4, pp. 177–197, 2017.

[28] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis, "SAuth: Protecting user accounts from password database leaks," in *20th ACM Conference on Computer and Communications Security*, Nov. 2013.

[29] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 6806, 2011.

[30] R. W. F. Lai, C. Egger, D. Schröder, and S. S. M. Chow, "Phoenix: Rebirth of a cryptographic password-hardening service," in *26th USENIX Security Symposium*, Aug. 2017, pp. 899–916.

[31] P. MacKenzie and M. K. Reiter, "Delegation of cryptographic servers for capture-resilient devices," *Distributed Computing*, vol. 16, no. 4, pp. 307–327, Dec. 2003.

[32] ——, "Networked cryptographic devices resilient to capture," *International Journal on Information Security*, vol. 2, no. 1, pp. 1–20, Nov. 2003.

[33] Microsoft Threat Intelligence Center, "STRONTIUM: Detecting new patterns in credential harvesting," https://www.microsoft.com/security/blog/2020/09/10/strontium-detecting-new-patters-credential-harvesting/, 10 Sep. 2020.

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

[35] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology – CRYPTO 2003*, ser. Lecture Notes in Computer Science, vol. 2729, 2003, pp. 617–630.

[36] 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 Conference on Computer and Communications Security*, Oct. 2017.

[37] C. Petrov, "50 Gmail statistics to show how big it is in 2020," https://techjury.net/blog/gmail-statistics/, 30 Jun. 2020.

[38] S. Ramezanian, T. Meskanen, M. Naderpour, and V. Niemi, "Private membership test protocol with low communication complexity," in *11th International Conference on Network and System Security*, ser. Lecture Notes in Computer Science, vol. 10394, Aug. 2017.

[39] A. C. D. Resende and D. F. Aranha, "Faster unbalanced private set intersection," in *22nd International Conference on Financial Cryptography and Data Security*, 2018, pp. 203–221.

[40] J. Schneider, N. Fleischhacker, D. Schröder, and M. Backes, "Efficient cryptographic password hardening services from partially oblivious commitments," in *23rd ACM Conference on Computer and Communications Security*, Oct. 2016, pp. 1192–1203.

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

[42] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein, "Data breaches, phishing, or malware? Understanding the risks of stolen credentials," in *24th ACM Conference on Computer and Communications Security*, 2017.

[43] Verizon, "2020 data breach investigations report," https://enterprise.verizon.com/resources/reports/dbir/, 2020.

[44] 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 Conference on Data and Application Security and Privacy*, Mar. 2018.

[45] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A security analysis of honeywords," in *25th ISOC Network and Distributed System Security Symposium*, Feb. 2018.

[46] K. C. Wang and M. K. Reiter, "How to end password reuse on the web," in *26th ISOC Network and Distributed System Security Symposium*, Feb. 2019.

[47] ——, "Detecting stuffing of a user's credentials at her own accounts," in *29th USENIX Security Symposium*, Aug. 2020.

[48] C. Yue and H. Wang, "BogusBiter: A transparent protection against phishing attacks," *ACM Transactions on Internet Technology*, vol. 10, no. 2, May 2010.