Lethe: Practical Data Breach Detection with Zero Persistent Secret State

Antreas Dionysiou
University of Cyprus
dionysiou.antreas@ucy.ac.cy

Elias Athanasopoulos
University of Cyprus
athanasopoulos.elias@ucy.ac.cy

***Abstract*—Honeywords are false passwords associated with each user account. Using a honeyword to login sets off an alarm as a data breach has been detected. Existing approaches for detecting data breaches using honeywords suffer from the need of a trusted component to tell honeywords from the valid password. Once this trusted component is compromised, then honeywords can offer no assistance for mitigating or detecting a data breach. In this paper, we present Lethe, a honeywords-based data-breach detection system that requires no trusted components, other than a trusted bootstrap, and keeps limited transient state for verifying login attempts.**

Lethe is based on two fundamental principles. First, Lethe generates honeywords using a Machine Learning (ML) model, which constantly evolves. This means that an attacker that compromises the Honeyword Generation Technique (HGT) cannot reproduce the same set of honeywords, and thus cannot tell which password was used as the initial generator. In particular, Lethe is the first system that allows an attacker to fully compromise the HGT without affecting the security of already generated honeywords.

Second, Lethe is not aware of the valid password. In fact, for Lethe the only one that knows the actual password is the user that selected it in the first place. Lethe records login events, but without storing anywhere the password used. These login events can be further replayed in another server, which can check if, for a particular user, there were at least two different passwords used and therefore detect a data breach.

Lethe allows the detection of a data breach deterministically and not probabilistically as similar approaches do. Additionally, Lethe allows detecting data breaches that are associated with rarely used accounts. Lethe can signal an alarm even if a user account that has logged in *just once* with the system is compromised. This is in contrast to other efforts that require legitimate users to authenticate with the system, after the attacker has done so, for detecting the breach. To demonstrate the effectiveness of Lethe, we provide a fully functional prototype, along with the ML-based HGT, and assess the provided security with a set of diverse attackers.

***Index Terms*—honeyword, decoy password, password, data breach detection**

1. Introduction

Passwords firmly remain the most prevalent method for user authentication and are expected to keep their place in the foreseeable future, despite their notorious defects in both security and usability [1], [2]. For example, existing password-based authentication systems maintain a sensitive file F comprised of all registered users' hashed passwords; if F is successfully retrieved and reverted by cracking the hashes, an adversary can undetectably impersonate any user. Nowadays, it is no news to hear that even high-profile web services, such as Yahoo [3], Dropbox [4], LinkedIn [5] and Facebook [6], have been compromised and their passwords are leaked. These data breaches are often detected after several months or years since the attackers had exploited those services and posted, or even sold, their data online [1].

An interesting approach for timely detecting data breaches, initially proposed by Juels and Rivest, is to utilize *honeywords* [2]. Honeywords are false passwords associated with each user account. Even if an attacker has successfully retrieved and reverted the password file F, they must still decide about each user's real password from a set of k distinct *sweetwords* [1]. Note that for each user account only one of those k sweetwords is the real password. Using a honeyword to login sets off an alarm as a data breach has been reliably detected.

While potentially effective, honeywords suffer from two related shortcomings that have limited their use in practice [7]. First, honeywords are only useful if it is hard to differentiate them from the real password. In particular, Juels and Rivest proposed four Honeyword Generation Techniques (HGTs), which have been later shown to be ineffective to meet the expected security requirements [1]. Later, however, other HGTs followed, achieving close to the optimal robustness against state-of-the-art honeyword-distinguishing attacks [8].

Second, previous proposals that leverage honeywords require a trusted component to detect the entry of a honeyword. This component must retain a secret state even after the target has been breached [2], [8], [9]. Such a trusted component, however, is a *strong assumption* 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. One approach showing that honeywords can be used to detect a target's database breach with no persistent secret state at the target, was initially proposed by Wang et al. [7]. In particular, the authors propose Amnesia, a data-breach detection framework that enables the target to detect its own breach *probabilistically*.

Amnesia uses a marking scheme for noting possible

1. Each user's real password and their $k-1$ honeywords are called *sweetwords*. Juels and Rivest suggest to use 20 sweetwords per user account ($k = 20$) [2].

user-chosen passwords. This marking scheme ensures that the last used sweetword to access an account is always marked; the remaining sweetwords are marked independently with a certain probability. In case an attacker accesses an account using a different than the user-chosen password and the actual user-chosen password becomes unmarked, then, *when the legitimate user next accesses the account*, an alarm will be signalled since the supplied password is unmarked.

We provide a full description of the marking scheme, the complete list of limitations and a thorough comparison of this work with Amnesia in Sec. 2.3. In short, Amnesia can detect the breach *probabilistically* and only when legitimate users authenticate explicitly with their passwords, and not using established cookies, after the breach has actually happened. This is a rather strong assumption, since users tend to rarely authenticate with websites by explicitly entering their passwords. The most recent study we are aware of is back in 2016, which reports users having at least a single password-entry event per day [10], while past research, of 2007, reports that computer users undertake between 8 and 23 password-entry events every day [11]. We expect that this evident reduction in explicitly entering passwords to be further augmented today, especially with the prevalence of mobile specific-site apps, where the user logs in just once with the service.

In this paper, we provide the design and implementation of Lethe, which offers deterministic detection of a data breach, without relying on users to explicitly authenticate with the service *after* the breach has happened. Lethe is a *bounded-time* data-breach detection system, which does not require the external trusted entity, that detects the entry of a honeyword, to be trusted at *all* times. This is in contrast to the original honeywords schema [2]. Compared to Amnesia that assumes users often access their accounts, Lethe can detect a breach deterministically in a few hours, even when the user has performed *a single login in their entire lifetime*. Note that the average detection delay of a data breach ranges from 7 to 15 months [12], [13].

Lethe does *not* rely on any *persistent* secret state located in storage for determining the success of its users' login attempts. In fact, for Lethe the only one that knows the actual password is the user that *has selected* the password in the first place - hence the name Lethe [2]. Furthermore, Lethe can defend against attackers that can gain *full* access to the deployed HGT. Due to the *stochasticity* involved in the selected HGT, it is impossible to reproduce the same sweetwords for any given password. An attacker that wants to infer the real password by entering all passwords to the HGT will simply produce entirely new sets of sweetwords that signal zero information about the real password.

Lethe's operation involves two servers, namely the authentication server, S, and the checking server, C. Both servers initialize a cryptographically secure Random Number Generator (RNG), namely R_S and R_C, using the *same* seed, which is then discarded. Thus, S and C are essentially *synchronized* on producing random integers in the range $[1, 20]$.

Moreover, S records login events, stores them in a logins file L and blends the user-chosen password with sweetwords. The position of the password given by the user is decided by invoking R_S, randomizing the positions of the remaining sweetwords. As a result, an adversary that has access to the logins file L, has the same success rate as in the classic honeywords paradigm, which is $1/20$ if 20 sweetwords are used per user account.

Periodically, when the data-breach detection phase occurs, server C replays all login events by invoking its own RNG (R_C), and selecting the sweetword that was given by the user for each login attempt. If C identifies at least one case where *different* sweetwords have been provided as input for the *same* user account, C raises a data breach alarm.

Otherwise, if no breach is detected, C *simulates* a login attempt with the last used sweetword for each user account, by invoking S's RNG (R_S), and overwrites the logins file L. This action allows C to check if different sweetwords have been provided as input for the same user account between *different* data breach detection points. As a result, Lethe is capable of detecting a data breach even if a user has been logged in only once in their lifetime, during registration.

Our contributions can be summarized as follows.

1) We propose Lethe, a honeywords-based data-breach detection framework that allows the target to detect a breach deterministically. Lethe reduces the need for an external trusted entity, namely honeychecker, for detecting the entry of a honeyword. In particular, Lethe requires the external trusted entity to be trusted *only* during the data-breach detection phase, which happens *off-line*. This is in contrast to the original honeychecker, which is required to vet for *any* authentication attempt, and thus be trusted at *all* times. Compromising Lethe's trusted entity, at any time beyond checking, does not reveal the actual passwords.
2) We are the first to propose a deterministic data-breach detection framework that *guarantees* the detection of such incidents in a few hours, without being dependent on any persistent secret state for validating login attempts. Furthermore, in contrast to other approaches found in the literature, Lethe does not require specific steps to be made from the legitimate users for signalling an alarm. In other words, Lethe is capable of detecting a data breach incident irrespective to the legitimate users' actions.
3) Contrary to other papers that assume attackers who cannot infer the HGT, Lethe relaxes this assumption and allows for adversaries that can gain complete access to the HGT. Nonetheless, Lethe protects against such adversaries due to the stochasticity involved in the selected HGT, which makes it impossible to reproduce the same sweetwords for any given password. An attacker that wants to infer the real password by entering all passwords to the HGT will simply produce entirely new sets of sweetwords that signal zero information about the real password.
4) Lethe does *not* require backup authentication

2. Lethe, from the ancient Greek word "λήθη", which means forgetting. A fully functional prototype is available on Bitbucket (https://bitbucket.org/srecgrp/lethe-public/).

mechanisms to be deployed in case an attacker, who managed to breach into the system, triggers a password reset, in contrast to similar techniques found in the literature [7]. Instead, Lethe is by-design capable of handling such cases and timely signalling data-breach alarms.

The rest of this paper is organized as follows. Some preliminaries regarding the generation/operation of honeywords and decoy passwords, as well as Amnesia's assumptions/limitations and how Lethe deals with them, are discussed in Sec. 2. In Sec. 3, we provide the detailed threat model used in this work, and later, in Sec. 4, we provide the detailed architecture and operation of Lethe. In Sec. 5, we evaluate our framework and, in Sec. 6, we provide a detailed discussion in regards to its strengths and limitations. Finally, in Sec. 7, we outline related work, and in Sec. 8, we conclude this work.

2. Preliminaries

In this section, first, we provide some background knowledge in regards to honeywords (Sec. 2.1) and decoy passwords (Sec. 2.2), and second, we discuss Amnesia's limitations and how Lethe copes with them (Sec. 2.3).

2.1. Generating Honeywords

Honeywords are only useful if it is hard to differentiate them from the real password [1]. One can decide regarding whether or not a HGT produces high-quality honeywords using the metrics proposed by Wang et al. [1], namely *flatness* and *success-number* graphs.

- A flatness graph plots the probability of distinguishing the real password versus the number of allowed sweetword login attempts per user x ($x \leq 20$). A *perfect* HGT allows for a maximum of $x \times 1/20$ success rate for each allowed number of sweetword login attempts per user x.
- A success-number graph plots the total number of successful login attempts (logins with a real password), versus the total number of failed login attempts (logins with a honeyword). The success-number graph measures to what extent a method will produce vulnerable honeywords that could be easily distinguished. A *perfect* HGT produces 20 sweetwords per user with the same probability of being the real password.

For gathering the appropriate statistics and plotting the two aforementioned graphs Wang et al. deploy a honeyword distinguishing attacker, namely *Normalized Top-PW*. This adversary aims to find as many as possible real passwords before making T_1 failed login attempts per user and T_2 failed login attempts in total.

A Normalized Top-PW adversary tries each sweetword in decreasing order of *normalized* probability of being the real password. The probability of each sweetword $sw_{i,j}$ ($1 \leq i \leq n$ and $1 \leq j \leq 20$, where n is the total number of users) comes directly from a known probability distribution of a leaked password dataset D, such as RockYou [14], and is calculated as follows. For each sweetword that exists in D then $Pr(sw_{i,j}) = P_D(sw_{i,j})$ else $Pr(sw_{i,j}) = 0$. $\forall x \in D$, $P_D(x) = Count(x)/|D|$, where $Count(x)$ is the number of occurrences of x in D and $|D|$ is the size of the leaked password dataset D.

Moreover, a Normalized Top-PW adversary first attacks the user accounts for which their most probable honeyword is closest to 1. Thus, it normalizes each user's 20 sweetwords as follows. $\forall sw_{i,j} \in n \times 20$ sweetwords, $Pr(sw_{i,j}) = Pr(sw_{i,j})/\sum_{t=1}^{20} Pr(sw_{i,t})$. If $T_1 > 1$ and after a sweetword has been attempted, the probability of all the other unattempted sweetwords should be re-normalized.

Furthermore, any proposed HGT should ensure its *non-reversibility* property [8]. In other words, it has to be computationally inefficient or impossible to go from the enriched with honeywords password file to the initial password file containing only the real password for each user. This will ensure that any adversary that retrieves and reverts the password file F, cannot reproduce the actual model used for generating those honeywords.

One such honeywords generation framework, that is robust against a Normalized Top-PW adversary and ensures its non-reversibility property, is HoneyGen [8]. HoneyGen leverages representation learning techniques to learn useful and explanatory representations from each operator's password corpus for generating honeywords that are indistinguishable from real passwords.

HoneyGen leverages Machine Learning (ML) technologies on purpose since the intrinsic stochasticity of the utilized models ensure that reversing the algorithm is computationally hard. In particular, the authors propose a *hybrid HGT* that is split into two phases. First, they train a word embeddings model, namely `FastText`, on the operator's password corpus, which allows to learn the structure of the input and produce a word embedding for each password. This enables them to query the word embeddings model regarding the top-k nearest neighbours of a given password. Second, they issue a *chaffing-by-tweaking* technique for stochastically perturbing the returned passwords, thus adding an extra step of randomness.

HoneyGen outperforms the state-of-the-art HGTs and meets the expected security requirements in terms of flatness and success-number graphs. In this paper, we utilize HoneyGen to guarantee that the adversary cannot achieve more than the random guessing baseline attack success rate when having access to either the password file F or the actual model used for generating those honeywords.

2.2. Decoy Passwords

Decoy passwords were initially proposed to tackle the problem of password reuse, which can cause a system to fail if a user recycles the same password across vouching services [15]. Honeywords and decoy passwords have the same generation requirements. That is, they have to be indistinguishable from the real password. However, they differ in their core functionality. In particular, the main difference between decoy passwords and honeywords is that any of the decoys can successfully authenticate the user into the system, whereas the use of a honeyword sets off an alarm [2].

Lethe cannot tell the real password from the honeywords as it removes the honeychecker component. In

Lethe's context *only* the legitimate user knows the real password. Instead, what Lethe can tell is whether or not different sweetwords have been provided as input, for a particular user account. Thus, Lethe combines honeywords with decoys as, during the authentication phase, if any of the sweetwords for a particular user is given as input, Lethe approves access to the system. However, Lethe can detect cases where *different* sweetwords have been provided as input for a specific user account. If this is the case, Lethe signals a data breach alarm.

2.3. Amnesia

Amnesia marks users' sweetwords probabilistically with binary values. Marking ensures that the password last used to access the account is always marked, and thus its associated binary value is 1. For each successful login attempt, the user's set of sweetwords is remarked with probability p_{remark}, in which case the entered password is marked with probability 1 and each of the other sweetwords is marked independently with probability p_{mark}. As a result, if an attacker accesses an account using a honeyword, then the user-chosen password becomes unmarked with probability $p_{remark}(1 - p_{mark})$. In that case, the breach will be detected *when the legitimate user next accesses the account*, since the password they supply is unmarked.

Limitations. Amnesia offers only probabilistic detection guarantees, which might not be enough for certain cases. First, repeatedly observing the sweetwords left marked by legitimate user logins permits the attacker to narrow in on the user-chosen password as the one that is always marked. This means that legitimate logins should remark the passwords as rarely as possible (p_{remark} should be small) or that, when remarking occurs, already marked passwords stay that way (p_{mark} should be large).

Second, if an attacker accesses an account between two logins by the user, a remarking of the real password *must* occur if there is to be any hope of the second legitimate login triggering a detection. This means that p_{remark} should be large, which imposes a contradiction with the previous point.

Third, an attacker can repeatedly trigger many remarkings, between consecutive legitimate logins, and stop until the initial marking sequence is restored. Doing so, will ensure that the next legitimate login signals *no* data breach. This suggests that an attacker cannot trigger arbitrarily many remarkings on an account, i.e., p_{remark} should be small, or that when remarkings occur, significantly many passwords are left unmarked, i.e., p_{mark} should be small. This comes to direct contradiction with both points previously mentioned. Generally speaking, selecting the optimal values for p_{remark} and p_{mark} is not a trivial task, let alone the fact that in any case the adversary has a larger chance of evading detection compared to the chance provided by the classic honeywords schema [2].

Fourth, Amnesia assumes an optimal HGT that cannot be accessed or reverted as this would enable adversaries to distinguish the real passwords from the honeywords deterministically.

How Lethe Copes with Amnesia's Limitations. Lethe resolves all of the difficulties discussed in the previous paragraphs. In particular, in Lethe's context all sweetwords have an equal probability of being the real password as we do not use any sort of sweetwords' marking. As a result, Lethe tackles Amnesia's first limitation.

In addition, Lethe does not require any specific steps to be made by the legitimate users, which is widely deemed not desirable, for signalling a data breach, in contrast to Amnesia's approach. In particular, Lethe *guarantees* the detection of a data breach irrespective to the legitimate users' actions. In this way, Lethe tackles Amnesia's second and third limitations.

Furthermore, Lethe relaxes Amnesia's assumption that the adversaries cannot infer the deployed HGT and allows for attackers that can trigger the generation of honeywords anytime (see Sec. 5.2.3).

Finally, Lethe's deterministic nature improves Amnesia's approach for tackling password reuse as it eliminates its probabilistic component. Combining Lethe with Amnesia's approach for tackling password reuse guarantees the detection of a data breach incident in case it happens.

3. Threat Model

The threat model of Lethe follows *exactly* the threat model of Amnesia, i.e., "an attacker to breach the target *passively only* [3], in which case it captures all persistent storage at the site associated with validating or managing account logins" [7]. Additionally, as in Amnesia, the attacker cannot reveal the TLS private key, MiTM the connection and predict future randomness by exfiltrating the state of the RNGs. All these assumptions (secure cryptography and secure random numbers) are explicitly stated also in Amnesia. However, Lethe, in contrast to Amnesia, allows an attacker to interfere with the honeywords generation algorithm.

Lethe utilizes two servers, one that runs the authentication and validates logins (authentication server S) and one that checks for data breaches (checking server C). The two servers communicate with TLS and we assume that the attacker cannot break the cryptography of the communication. Nevertheless, we assume that both servers can be compromised, multiple times but for a short period (see below), and all storage –all vital information stored there and not just hashed passwords– can be leaked. We allow attackers to compromise the servers using the techniques recorded in real incidents [16].

Following the threat model of Amnesia, we assume that attackers are not able to replace the code, break the cryptography or predict future randomness of our system. These assumptions stem from the fact that attackers aim on exfiltrating massively user credentials without being detected. According to studied data breaches, attackers do not hold the compromise for a long period. They just target the storage information, and not the system. This is natural, since attackers may be benefited by further selling the credentials, or using the cracked credentials to other sites, rather than affecting the actual system that holds the credentials.

Attackers that have compromised any of the two servers may passively monitor the memory of the system.

3. Following this threat model, potential attackers cannot modify/edit or overwrite files F and L associated with users' logins and passwords.

In that case, attackers may reveal some of the users' supplied passwords. For instance, if the user logs in with the server while the latter is compromised, then the attacker may passively sniff the user-supplied password from memory. Lethe cannot protect against this, and this is the case for Amnesia.

However, we anticipate that an attacker risks being detected by holding the compromise for a long period to sniff passively in real-time passwords; users *rarely* authenticate by supplying passwords (most of the time they use established cookies), and the attacker essentially needs to exfiltrate a bulk amount of passwords and not just a few. We stress here that there is *no* system, to the best of our knowledge, that can protect against an attacker that has full access to the memory of the system for the entire period of the system's operation.

Finally, in contrast to other papers that assume attackers who cannot interfere with the honeywords generation algorithm, Lethe relaxes this assumption and allows for attackers that can trigger the generation of honeywords anytime. An attacker may supply any input to the HGT, observe the output and compare it with leaked credentials in order to *guess* the machine-generated ones. Lethe can protect against this (see Sec. 5.2.3).

4. Lethe

Lethe is based on two servers, the authentication server S, which is the typical web site that offers users with an authentication form, and the checking server C, which detects the data breach. Both servers can be compromised (see Sec. 3) after a first trusted bootstrap phase, where both servers initialize a cryptographically secure RNG. The generator is initialized in both servers with a seed that is then discarded. For details on how Lethe returns back to normal state in case the servers become de-synchronized see Sec. 4.7.

Once the two servers are initialized, then Lethe starts the authentication and data-breach detection algorithms. The authentication server S records login events from users, when entering their passwords, and the checking server C periodically checks if there was a data breach. A login event triggers the generator, and permutes the used password based on the random value released. Once the login has happened, the associated random value is discarded, and only the event is recorded, e.g., user Alice is logged in successfully in the system.

Both servers are *unaware* of the users' real passwords, so compromising any of them, or even both, at any time, reveals zero information about the real passwords stored along with the rest of the sweetwords. Detection happens periodically by replaying all login events and checking if there were different sweetwords used for one particular account. Since the two servers share a common RNG, C can replay all login events as appeared in S and detect if an account has been accessed with multiple sweetwords. Detection is deterministic even for compromised accounts that issued a single login attempt in their entire lifetime, during registration.

We assume that with Lethe in place attackers will focus their efforts to exploit the checking time where all passwords are exposed. Therefore, we offload this checking time to C, which is an external server with minimal functionality for checking for a data breach off-line. Note that this server needs to be protected only during checking. Compromising the memory of the server at any other time reveals nothing useful for the attacker. Compromising the storage of the server at *any* time, including checking, does not reveal the actual passwords since they are blended with honeywords. This is in contrast to the original honeychecker, which reveals all valid passwords and honeywords when either its memory or storage are compromised [2].

4.1. Lethe's Operation

Lethe's operation is split into four phases, namely *registration* (fig. 1(a)), *authentication* (fig. 1(b)), *data breach detection* (fig. 1(c)), and *password reset* (fig. 1(d)). Registration and authentication phases invoke the authentication server S, whereas the data breach detection, which occurs *once* per day and *off-line*, and password reset phases invoke both the authentication server S and the checking server C. S and C utilize two synchronized cryptographically secure RNGs, namely $R_S()$ and $R_C()$, which are initialized using the same seed at the start of Lethe's operation. Then, this seed is *discarded*. As a result, server S and C are essentially *synchronized* on producing random integers in the range of $[1, 20]$. The initialization process is done via the trusted bootstrap.

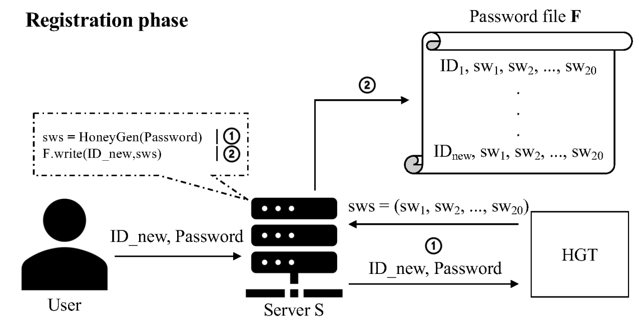
Note that the trusted bootstrap is a state where the system is *not* operational. That is, there are no inputs or incoming connections processed and it is performed in a fully isolated environment. During this phase, we assume *no* attacks. The entire process of the trusted bootstrap is fairly simplistic and involves only the synchronization of two RNGs. Thus, it can be carried out in a short time during the initialization of the system where no external inputs are processed.

4.2. Registration Phase

Fig. 1(a) shows Lethe's registration phase. As shown, Lethe uses a HGT to produce 19 honeywords for the password given by the user. Then, the list of 20 sweetwords (19 honeywords + the real password) is stored in F along with the user's ID. We utilize HoneyGen as our HGT for the reasons explained in Sec. 2.1. In short, we allow HoneyGen to be compromised, since future invocations of honeywords generation cannot reveal the original password; using a single password as input to HoneyGen twice results to entirely different generated sweetwords. After the registration process is completed Lethe uses the provided ID and password to log the user into the system.

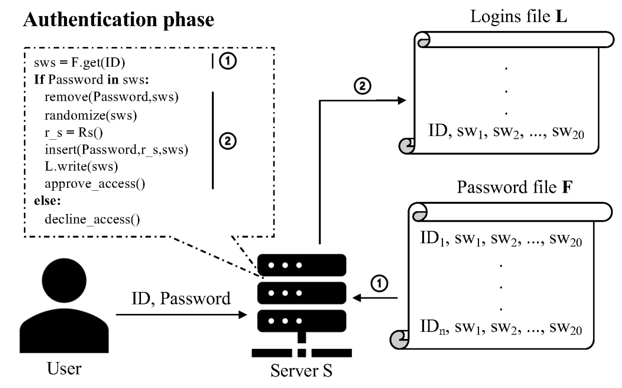
4.3. Authentication Phase

Fig. 1(b) shows Lethe's authentication phase. As shown, the server S uses the provided ID and password to retrieve the list of sweetwords sws stored in F for that particular user. If the given password exists as a sweetword in sws, Lethe creates a new list sws where the provided password is stored at a random position (1-20) specified



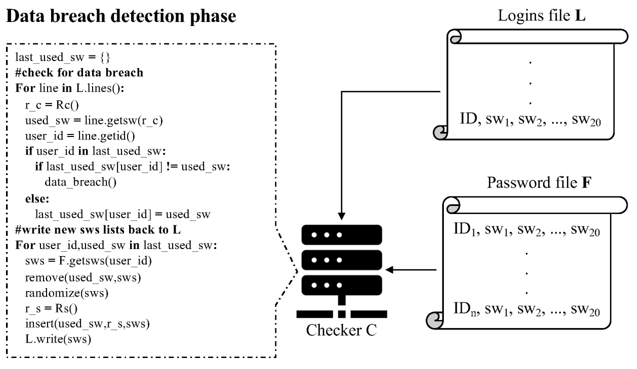
```
sws = HoneyGen(Password)  | ①
F.write(ID_new,sws)       | ②
```

(a) **Registration phase.** First, the server S feeds the user's real $password$ to the deployed HGT and receives, as a response, the list sws containing 20 sweetwords (1 of them being the real password). Second, the server S stores the list of sweetwords sws to the password file F along with the user's ID.



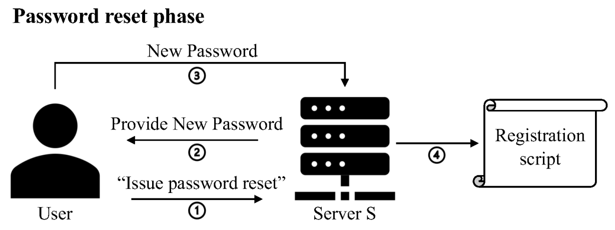
```
sws = F.get(ID)                 | ①
If Password in sws:
   remove(Password,sws)         |
   randomize(sws)               |
   r_s = Rs()                   | ②
   insert(Password,r_s,sws)     |
   L.write(sws)                 |
   approve_access()             |
else:
   decline_access()
```

(b) **Authentication phase.** First, the server S retrieves a copy of the list of sweetwords sws for the provided ID from the password file F. Second, it checks whether the provided $password$ exists as a sweetword in sws. If yes, S: (i) removes $password$ from sws, (ii) randomizes the positions of the remaining sweetwords, (iii) places the $password$ in the specific position derived from $R_S()$, (iv) stores the new list of sweetwords sws, along with the user's ID, in logins file L, and (v) approves access to the system. If not, the user is denied access to the system.



```
last_used_sw = {}
#check for data breach
For line in L.lines():
   r_c = Rc()
   used_sw = line.getsw(r_c)
   user_id = line.getid()
   if user_id in last_used_sw:
      if last_used_sw[user_id] != used_sw:
         data_breach()
   else:
      last_used_sw[user_id] = used_sw
#write new sws lists back to L
For user_id,used_sw in last_used_sw:
   sws = F.getsws(user_id)
   remove(used_sw,sws)
   randomize(sws)
   r_s = Rs()
   insert(used_sw,r_s,sws)
   L.write(sws)
```

(c) **Data breach detection phase.** First, the checker C iterates over all login attempts (records in L) and extracts each user's selected sweetword by invoking $R_C()$. For each login attempt the checker C checks if the user's selected sweetword is the same as the one stored in $last\_used\_sw$ dictionary. If yes, it proceeds to the next record, else it sounds an alarm. If C encounters a user for which an entry in $last\_used\_sw$ does not exist, it creates a new entry for that user and stores the used sweetword. Finally, if no data breach has been detected, the checker C: (i) overwrites L, and (ii) iterates over all $(user\_id, used\_sw)$ pairs in $last\_used\_sw$ and for each one of them it creates a new sweetwords list sws, where each user's last used sweetword is stored at the position derived from $R_S()$, randomizing the positions of the remaining sweetwords.



(d) **Password reset phase.** In order for this phase to kick in, the authentication phase, see fig. 1(b), has to precede. First, the user issues the password reset option. Second, the server S asks from the user to provide the new $password$. After the user provides the new $password$ (step 3) and hits submit, the server S initiates the registration phase (step 4, see fig. 1(a)). A potential data breach will be detected at the end of the day when the data breach detection phase will kick in.

Figure 1. Lethe's operation phases, namely registration (fig. 1(a)), authentication (fig. 1(b)), data breach detection (fig. 1(c)), and password reset (fig. 1(d)). Lethe does not work with plain passwords but with hashed ones. In some visualizations we omit the $H()$ function for better understanding.

by $R_S()$ and randomizes the positions of the remaining sweetwords. Afterwards, S stores the reordered list of sweetwords and the ID of the user into logins file L, and authenticates the user into the system. The system declines access if the given password does not exist in sws. It is worth mentioning that for a system with millions of passwords the overhead for this phase is practically zero.

Note that the server S cannot tell if the user has entered the real password or not. The only fact that S can check is if the user has entered a valid sweetword. For S, during authentication *all* sweetwords are valid passwords. S assumes that a legitimate login uses the real password, but a malicious login uses one of the other sweetwords and that can be detected in the short future. Furthermore, note that S does not record which password was actually used during a successful login. Instead, S permutes the sweetwords based on a random token.

Only one that has a synchronized random generator can replay the login attempts and see which passwords were actually used. Otherwise, even S cannot go back in time and re-use the released random tokens to infer which passwords were used, unless the random tokens are stored, which is not the case.

4.4. Data Breach Detection Phase

Fig. 1(c) shows Lethe's data-breach detection phase. We refer to the interval between two successive data-breach detection points as an *epoch*. During this phase, which happens at the end of each epoch, the server C iterates over all login attempts in L, and checks whether or not *different* sweetwords have been provided as passwords for a particular user account. If this is the case, Lethe sets off an alarm as a data breach has been detected. For doing so, C invokes $R_C()$, for selecting the sweetword that was given by the user for each login attempt.

The checking server maintains a dictionary, namely $last\_used\_sw$, containing the last used sweetword for each user account. This dictionary is discarded after checking is completed. By using this dictionary, C can infer if a login was made with a provided password that is different than the one stored in $last\_used\_sw$ for that particular user. The computational complexity of data breach detection phase is $O(n)$, where n represents the total number of lines in logins file L.

Note that C releases random tokens, as it happened during the actual authentication in S, and *replays* all login events. This replay can be done only by C because it has access to the synchronized generator. Also, this replay can happen *once*, since all random tokens are discarded and *going back* to the generator's sequence is not possible. This replay is not possible even in S, which recorded all login events in the first place.

Detection can happen since C in replaying all login events can infer if a particular user authenticated with more than one sweetwords in the system. However, we expect that during an epoch most of the users will have logged in once or even zero times in the system. For this, after the checking phase has been completed and no data breach has been detected, C overwrites the logins file L. In particular, the checking server C adds one record for each user account placing the last used sweetword at the position derived from the server S's RNG, randomizing the positions of the remaining sweetwords.

This step essentially simulates a user login with the last sweetword used. This simulated login is propagated from epoch to epoch. For instance, a user that performed a single login attempt, during registration, will have their single successful login attempt propagated in the following epochs; it is, thus, sufficient for an attacker to log in just once as the particular user but with a different sweetword for signalling the detection of a data breach.

4.5. Handling Password Reset

In Amnesia the breach detection happens when the legitimate user logs into their account *after* the attacker has done so. Thus, in case that an attacker triggers a password reset the legitimate user is locked out of their account, without a data breach being detected. For this reason, Amnesia requires the target to utilize a backup authentication method, before enabling password reset.

Contrary, Lethe is capable of signalling a data breach alarm *without* the need of any backup authentication mechanism (see Fig. 1(d)). In particular, for the attacker to issue the password reset routine, they have to first log into the system, and thus the authentication phase (see Fig. 1(b)) has to be executed with the given ID and password [4]. Note that during the authentication phase, the adversary must guess the correct password from the list of 20 sweetwords. Otherwise, the next data breach detection phase occurrence, at the end of the epoch, will signal an alarm if *different* sweetwords are provided as password for the same user account. Afterwards, when the attacker enters the new password and clicks submit, Lethe initiates the registration phase (see Fig. 1(a)).

Whenever a password reset is issued, the event is logged in L. Lethe can differentiate between a password reset issued by the legitimate user and by an attacker, assuming the attacker has used the wrong password to log in for triggering the reset. At the end of the epoch, while checking all authentication events, Lethe will signal that a data breach has happened, before reaching the password-reset event, since the attacker's login with the wrong password is going to be processed first. Now, assuming there is no data breach detected and a password-reset event is processed, then a new login, with the new password, is forwarded from C to S, and the new password is now used from all future epochs.

4.6. Example of Operation

In order to better communicate Lethe's operation we provide a step-by-step example with two epochs, showing how data breach detection works, assuming one registered user and an authentication system that uses ten sweetwords per user account (see Fig. 2).

Initially, at epoch 1, the user registers to the system using the password PW_1. Lethe derives the list of sweetwords for the given password, by issuing HoneyGen, and next authenticates the user into the system. The authentication involves receiving a random integer from $R_S()$, i.e., 1, which will be used for storing the given password at the specific position in the list of sweetwords, randomizing the positions of the remaining sweetwords. Note that Lethe cannot differentiate the user-chosen password from the sweetwords; in Fig. 2, we use the notation PW to indicate the sweetword that is used by the user during authentication.

Next, during the same epoch, the user logs into the system once more, giving the password PW_1 as input. Lethe invokes $R_S()$ and creates another record in logins file L placing PW_1 at the position 8. Next, the user logs into the system one more time, during the same epoch, using again PW_1. The same procedure is followed, and a new entry is created in L placing PW_1 at position 2.

Afterwards, the end of epoch occurs and the data breach detection phase kicks in. During this phase, the server C replays all login events. In particular, for each record in L, the checking server C invokes $R_C()$ and selects the respective sweetword located at that position. As shown, C observes the same password, that is, PW_1, for all user's login attempts, and thus no breach is detected. Next, C initiates the start of the new epoch, by creating a new record in L and storing the last used sweetword (PW_1) at position 3 that was received by invoking $R_S()$.

4. Note that the password reset phase does not apply on resetting *forgotten* passwords, where standard practices can be followed, but only on the intentional change of the password. This is the case for both Amnesia and Lethe.

Epoch	Event	Action	File L									
1	User registers with password $\mathbf{PW_1}$	$R_S \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
1	User authenticates with password $\mathbf{PW_1}$	$R_S \leftarrow 8$	SW_2	SW_4	SW_8	SW_3	SW_7	SW_9	SW_5	$\mathbf{PW_1}$	SW_4	SW_6
1	User authenticates with password $\mathbf{PW_1}$	$R_S \leftarrow 2$	SW_2	$\mathbf{PW_1}$	SW_3	SW_8	SW_6	SW_9	SW_7	SW_4	SW_8	SW_5
Data breach detection phase												
1	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
1	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 8$	SW_2	SW_4	SW_8	SW_3	SW_7	SW_9	SW_5	$\mathbf{PW_1}$	SW_4	SW_6
1	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 2$	SW_2	$\mathbf{PW_1}$	SW_3	SW_8	SW_6	SW_9	SW_7	SW_4	SW_8	SW_5
No data breach detected.												
Next epoch												
2	Add last used_sweetword to L	$R_S \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
2	User authenticates with password $\mathbf{PW_2}$	$R_S \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_2}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
Data breach detection phase												
2	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
2	used_sweetword = $\mathbf{PW_2} \neq \mathbf{PW_1}$	$R_C \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_2}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
Signal a data breach alarm!												

Figure 2. A timeline of Lethe's operation assuming 1 registered user and an authentication system that uses 10 sweetwords per user account. As shown, for the 1st epoch, Lethe detects the entry of a single password (PW_1) for all user's login attempts, thus, not signalling a data breach. Then, server C starts the new epoch and propagates the last used sweetword by adding it back to L. For the 2nd epoch, the checking server C observes a login attempt using a different, than the one given before, password (PW_2). As a result, C signals an alarm since a data breach has been detected.

During the 2nd epoch the attacker logs into the system providing a different password, PW_2. Lethe invokes $R_S()$ and creates a new record in L placing the given password (PW_2) at position 4. Finally, the data breach detection phase occurs once more and the server C replays all login events using $R_C()$. Invoking $R_C()$ once returns 3 and C selects PW_1 as the last used sweetword. C invokes $R_C()$ once more and receives 4. Then, C selects the sweetword located at position 4 (PW_2) and compares it with the previous one (PW_1). C realizes that $PW_1 \neq PW_2$ and immediately signals an alarm.

Finally, Figs. 3 and 4 show two more example timelines, where the user logs in once, during registration, and then logs in again, at a significantly later epoch e, using either the *same* (Fig. 3) or a *different* (Fig. 4) password. In both cases, we can see that the user's initial successful login attempt is propagated to *all* subsequent epochs. This propagation mechanism makes Lethe capable of detecting a data breach irrespective to the user's login attempts.

4.7. De-synchronization of S and C

Intuitively, RNGs can be de-synchronized in cases where users are involved. For example, when a user accidentally issues more random tokens than needed. This is not the case with Lethe, where machines release random tokens following a prescribed algorithm. Nonetheless, a de-synchronization event between $R_S()$ and $R_C()$ can potentially happen in case: (a) one of the servers crashes/loses its state of the RNG, and (b) an attacker manages to explicitly make a call to either $R_S()$ or $R_C()$ in order to trigger a de-synchronization. This would lead in a data breach being detected while there was not one. In this section, we provide details on how Lethe can be re-synchronized in case of such situations.

When a de-synchronization event happens, there are two parts to be addressed: (a) detection and (b) re-synchronization. For (a), Lethe instructs S and C to issue a random number and check it over TLS at the start of each epoch. If the numbers are different then it means that there was a de-synchronization event in the previous epoch. In that case, Lethe is not able to detect if a data breach has happened in the last epoch, since it might be a false positive or an actual breach. We assume that (i) the data breach, if there was one, will be detected in a future epoch and (ii) several de-synchronization events is an indicator that the system is experiencing abnormal behavior.

For (b), which is already implemented in our system, we use the trusted bootstrap. Lethe needs no state to run again correctly, since the state is actually shared by the users that know their own valid passwords. Moreover, Lethe is able to detect a data breach in the next epoch. Note that in such case, files L and F are both reset to a clean state.

5. Evaluation

In this section we evaluate Lethe. Compared to other systems, Lethe involves the network communication of two servers, S and C, for detecting a data breach. This communication, although not frequent, can introduce some overhead, which is moderate. We explore this overhead below. Moreover, we provide a discussion of how Lethe can detect a breach when different attackers attempt to leak the passwords later in this section.

5.1. Network Overhead

In order for Lethe to operate as expected, servers S and C must maintain a copy of files F and L. S updates F every time a new user is registered and L for each new login attempt, assuming the provided password is valid.

When the data breach detection phase occurs, C has to update its own instances of files F and L. For updating F, C needs only the new registered users along with their sweetword lists. These updated records can be sent to C either in their plain text format or in a compressed form. For example, if $500k$ new users are registered the size of the updates is 102 MB and 46.5 MB for the plain or compressed form [5], respectively.

For updating L, C needs the records for the new login attempts made to S. Similar to F's updates, the new login attempts can be either transferred in their plain text format or in a compressed form with similar file sizes as the

5. We use WinZip, however, other libraries that compress text files more efficiently, such as 7-zip, do exist.

Epoch	Event	Action	File L									
1	User registers with password $\mathbf{PW_1}$	$R_S \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
Data breach detection phase												
1	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
No data breach detected.												
Next epoch												
2	Add last used_sweetword to L	$R_S \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
Data breach detection phase												
2	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
No data breach detected.												
⋮					⋮							⋮
Next epoch												
e	Add last used_sweetword to L	$R_S \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_1}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
e	User authenticates with password $\mathbf{PW_1}$	$R_S \leftarrow 9$	SW_2	SW_6	SW_1	SW_9	SW_3	SW_4	SW_7	SW_5	$\mathbf{PW_1}$	SW_8
Data breach detection phase												
e	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_1}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
e	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 9$	SW_2	SW_6	SW_1	SW_9	SW_3	SW_4	SW_7	SW_5	$\mathbf{PW_1}$	SW_8
No data breach detected.												
Next epoch												
$e+1$	Add last used_sweetword to L	$R_S \leftarrow 5$	SW_7	SW_5	SW_8	SW_1	$\mathbf{PW_1}$	SW_6	SW_9	SW_4	SW_3	SW_2
⋮					⋮							⋮

Figure 3. A timeline of Lethe's operation, where the registered user logins once (at registration) and then logins at a very later epoch e, using the *same* password (PW_1). As shown, the data breach detection phase, that occurs at epoch e, does *not* detect a data breach, and thus does not signal an alarm.

Epoch	Event	Action	File L									
1	User registers with password $\mathbf{PW_1}$	$R_S \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
Data breach detection phase												
1	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 1$	$\mathbf{PW_1}$	SW_1	SW_2	SW_3	SW_4	SW_5	SW_6	SW_7	SW_8	SW_9
No data breach detected.												
Next epoch												
2	Add last used_sweetword to L	$R_S \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
Data breach detection phase												
2	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 3$	SW_8	SW_3	$\mathbf{PW_1}$	SW_5	SW_6	SW_9	SW_7	SW_4	SW_2	SW_1
No data breach detected.												
⋮					⋮							⋮
Next epoch												
e	Add last used_sweetword to L	$R_S \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_1}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
e	User authenticates with password $\mathbf{PW_2}$	$R_S \leftarrow 9$	SW_2	SW_6	SW_1	SW_9	SW_3	SW_4	SW_7	SW_5	$\mathbf{PW_2}$	SW_8
Data breach detection phase												
e	used_sweetword = $\mathbf{PW_1}$	$R_C \leftarrow 4$	SW_2	SW_1	SW_9	$\mathbf{PW_1}$	SW_3	SW_5	SW_4	SW_7	SW_6	SW_8
e	used_sweetword = $\mathbf{PW_2} \neq \mathbf{PW_1}$	$R_C \leftarrow 9$	SW_2	SW_6	SW_1	SW_9	SW_3	SW_4	SW_7	SW_5	$\mathbf{PW_2}$	SW_8
Signal a data breach alarm!												

Figure 4. A timeline of Lethe's operation, where the registered user logins once (at registration) and then logins at a very later epoch e, using a *different* password (PW_2). As shown, the data breach detection phase, that occurs at epoch e, detects this incident, due to the propagated last used sweetword (PW_1), and thus signals a data breach alarm.

ones mentioned above. However, when the data breach detection phase is completed and C needs to propagate the sweetwords used for each user account to the next epoch, it is enough to ask S for a list containing n random tokens, where n is the number of registered users, from its RNG. These tokens are essentially integers in the range $[1, 20]$ and if we have 1 million registered users we only need to transfer a file of size 3.3 MB (or 737 KB if we compress it).

5.2. Security Analysis

Lethe can be attacked in numerous ways. We discuss here all possible attacks and how Lethe behaves. The relevant threat model (see Sec. 3) for Lethe is similar to the one used in Amnesia with some relaxations to allow for even stronger adversaries.

5.2.1. Attacking the Authentication Server. An attacker that compromises the storage of S cannot reveal any of the users' real passwords, since they are all blended with honeywords. The attacker can use any of the sweetwords associated with a particular account to successfully authenticate with the system. However, they risk being detected if they use a sweetword that is not the user's real password. Thus, the attacker has a probability of $1/20$ for being not detected, when logging to an account of a particular user and for a system with 19 honeywords in place.

The attacker can compromise S multiple times and observe the passwords stored in the disk, which constantly change during user logins. However, the attacker cannot tell real passwords from honeywords, since: (a) the attacker has no access to the RNG that is used to permute the honeywords and (b) we assume that future random values cannot be compromised –this is aligned with the assumptions in Amnesia.

Finally, the attacker can compromise S and sniff passwords, or random numbers, from memory. This cannot be protected by Lethe, but this is also out of scope for similar efforts [7]. As discussed in Section 3, we anticipate that attackers aim at compromising a vast amount of passwords, and not just a few, that can sniff from memory when users authenticate with the server. In fact, a fully compromised server that allows users to authenticate resembles more the threat model of phishing [17] and not the one of a data breach.

5.2.2. Attacking the Checking Server. An attacker that compromises the storage of C cannot reveal any of the users' real passwords, since they are all blended with honeywords. The storage of C does not change frequently, but only during checking. Again, the attacker can compromise the storage of C multiple times, but since there is no access to the RNG or to future values of it, the attacker cannot tell which password is the real one.

In contrast to S, C is a really minimal system. In fact, most of the time C can be off-line since C uses the network *only* for receiving login updates once every epoch. Even during checking for a data breach, C can be off-line. This means that practically C can minimize the window of opportunity for becoming compromised. The only interaction of C with the network is for receiving the login events from S through a TLS connection, which we assume secure, that is only open for a limited time and very sporadically.

Finally, compromising the memory of C during checking is out of scope. Again, having a full compromise and inspecting the memory of the system is out of scope for similar works [7]. But, note that C can be easily protected since there is limited interaction of C with the rest of the system. Later, in Section 6.4, we provide some directions for improving Lethe's robustness against adversaries that can compromise server C during the checking time.

5.2.3. Attacking HoneyGen. Contrary to Amnesia, Lethe relaxes the assumption that the adversaries cannot infer the HGT used to produce the honeywords for each user's password and allows for attackers that can trigger the generation of honeywords anytime. This is important as in the case that adversaries manage to invert the HGT, they can go from the enriched with honeywords password file back to the initial password file having only the real password for each user account.

For attacking the HGT an adversary can either: (a) gain access to, or (b) reproduce, the actual honeywords generation model stored in server S. Lethe protects from both cases. For the first scenario, even if the adversaries gain access to the honeywords generation model they cannot use it to extract the real password from the honeywords, by supplying any sweetword that exists in F and comparing the retrieved list of sweetwords with the one stored in F for each user. This is due to the added step of randomness in HoneyGen's hybrid HGT, which makes it impossible to respond with the same set of sweetwords (see Table 1), and the continuous batch retraining of its weights, every $1{,}000$ newly registered users, which dramatically changes its previous operation mechanics [8].

For the second scenario, HoneyGen offers protection by design as it is impossible to reproduce the same honeywords generation model used to generate the honeywords included in password file F for the same aforementioned reasons.

5.2.4. Lethe's Bounded Data Breach Detection Time. One of the main concerns for data breach detection frameworks is to minimize the time interval between the actual data breach incident and its detection from the system [1], [7], [8]. The optimal case scenario is the classic honeywords schema, where a data breach is detected as soon as the adversary attempts to log in with a honeyword [2]. However, this is hard to be achieved in practice when removing the honeychecker component.

Amnesia requires the legitimate user to login, after the attacker accesses the account using a honeyword, for signalling a data breach alarm. Thus, Amnesia assumes that users *often* access their accounts using their login credentials, something which is not the case [6]. Even if this is the case, Amnesia may *miss* detecting a data breach incident due to its probabilistic nature, i.e., stochasticity when marking the users' sweetwords. In contrast, Lethe can detect a data breach *deterministically* in a few hours, even when the user has performed a single login in their entire lifetime, during registration.

6. Discussion

This section provides a detailed discussion in regards to: (a) the way that Lethe improves the classic honeywords schema (Sec. 6.1), (b) how Lethe can be used with Amnesia's monitors for efficiently detecting data breaches in case of password reuse (Sec. 6.2), and (c) Lethe's limitations (Sec. 6.3).

6.1. Improving the Classic Honeywords Paradigm

Lethe improves the classic honeywords schema in two ways. First, Lethe removes the need for the external trusted entity to be trusted at all times, and thus minimizes the window of opportunity for becoming compromised. In Lethe's context, C needs to be trusted only during the data breach detection phase, which happens off-line.

Second, Lethe does *not* rely on any persistent secret state for validating login attempts; in Lethe's context, an adversary that accesses all persistent storage at the site associated with validating or managing account logins *cannot* reveal *any* of the users' real passwords since they are blended with honeywords.

6.2. Tackling Password Reuse

Users tend to select easy-to-remember passwords rather than uniformly distributed [8]. Not only that, but they also reuse their chosen passwords across multiple accounts [18], [19], [20]. As a result, in the case where

6. There is a trend in users entering their passwords less frequently, which makes detection schemes, based on explicit user logins, less efficient [7].

TABLE 1. THREE EXAMPLE HONEYWORD LISTS RECEIVED FROM HONEYGEN, WHEN GIVEN THE PASSWORD "*jeremiah523*" AS INPUT. AS SHOWN, THE RETRIEVED LISTS ARE DIFFERENT, THUS, MAKING IT IMPOSSIBLE FOR POTENTIAL ATTACKERS TO REPRODUCE THE SAME SET OF SWEETWORDS AS THE ONE STORED IN PASSWORD FILE F FOR EACH USER ACCOUNT, EVEN WHEN HAVING ACCESS TO THE ACTUAL HONEYWORDS GENERATION MODEL ITSELF.

$\mathbf{Hw_1}$	$\mathbf{Hw_2}$	$\mathbf{Hw_3}$	$\mathbf{Hw_4}$	$\mathbf{Hw_5}$	$\mathbf{Hw_6}$	$\mathbf{Hw_7}$	$\mathbf{Hw_8}$	$\mathbf{Hw_9}$	$\mathbf{Hw_i}$	$\mathbf{Hw_{19}}$
jeremiah23	jeremiah0623	jeremy823	jeremie03	jeremiaH1	jeremiah91	jeremiebarabin	jeremy_23	jeremiah29vs11	...	jeremiej1
Jeremiah23	jeremiah51	jeremiah29v11	jeremieh1	jeremiah1	$jeremiah$1	jeremias13	jeremie007	jeremy423	...	jeremiah29:11
jeremiah41	jeremias03	jeremey1	jeremiah29	jeremiah61	jeremylee1	jeremmi1	1jeremiah1	jeremy2006	...	jeremie03

an adversary compromises two targets tar_1 and tar_2 (employing honeywords), for which a specific user maintains accounts on both of them, the user-chosen password will likely be the one contained in the intersection of the two sweetword lists. Even if the second target tar_2 is not compromised, the attacker can reveal the user-chosen password by testing the leaked sweetwords, from the compromised target tar_1, at tar_2 one by one, which is also known as a credential stuffing attack [21].

For addressing the credential stuffing attacks, Amnesia enables the target to monitor for the entry of passwords stolen from it at other sites, called monitors. In particular, incorrect passwords entered for the same user at monitors are treated as if they had been entered locally at the target. For doing so, Amnesia introduces a cryptographic protocol, namely Private Containment Retrieval (PCR), by which a monitor transfers the password attempted in an unsuccessful login to the target, but only if the attempted password is one of the sweetwords for the same account at the target.

Lethe's approach is orthogonal with Amnesia detection and can be in principle used with Amnesia's monitors. If that happens, Lethe is faster in signalling a data breach alarm because detection is not based on users entering their passwords.

6.3. Lethe's Limitations

Lethe's servers S and C employ two RNGs, which are synchronized on producing random integers in the range $[1, 20]$. These two generators are initialized during the trusted bootstrap phase using a sensitive seed, which is immediately discarded. However, in case an adversary gains access to the seed before it is discarded, they can compute positions of the used sweetwords in the logins file L. The most common sweetword for each user will be most probably the real password.

Furthermore, although Lethe bounds the time interval between the actual data breach incident and its detection by the system, it still allows adversaries to act undetected for a certain amount of time (a few hours). This is due to Lethe's agnostic nature regarding the users' real passwords. In particular, Lethe authenticates login attempts that utilize any of the sweetwords for a particular user and later, during the data breach detection phase, checks whether or not different sweetwords have been provided as input for a specific user account. This duration, however, may be enough to cause significant damage to the target system in case the password file F is leaked.

Attackers that have compromised any of the two servers and passively monitor the *memory* of the system for a long period may reveal some of the users' supplied passwords. Similar to other honeywords-based detection frameworks, such as Amnesia, Lethe cannot protect against such cases.

However, we anticipate that adversaries risk being detected by holding the compromise for a long period to sniff, passively and in real-time passwords, let alone the fact that users rarely authenticate by supplying their credentials. Usually, attackers need to exfiltrate a bulk amount of passwords and not just a few. Note that for the time being, *no* system that can protect against adversaries that have *full* access to the memory of the system for its entire period of operation exists, at least to our knowledge. We have stressed all this discussion in Section 5.

6.4. Improving Lethe

An attacker that monitors, for a long period, the memory of the system can reveal some of the users' supplied passwords. More importantly, however, an attacker that compromises the memory of server C during the checking time can reveal the real passwords for all user accounts. There are two possible directions for mitigating this problem: (a) employing an RNG per user account, and (b) utilizing trusted computing.

A Single RNG per User Account. In its current form, Lethe utilizes a single RNG to encode and subsequently validate login attempts. The use of a single generator forces C to examine all user accounts, even the ones that are inactive, every epoch. An alternative option is to assign each user with an RNG. A unique generator per user can facilitate in checking only the active accounts each time. In that case, an attacker that has fully compromised C and can inspect its memory during checking can leak *only* the passwords of specific accounts.

Trusted Computing. In Lethe's current schema, C realizes an *isolated* data-breach detection environment. This is because the checking for a data breach happens off-line, and thus the window of opportunity for compromising C is minimized. In principle, this operation could be realized in the main server using trusted computing, such as SGX [22] or similar upcoming technologies [23], and thus removing the need of maintaining two servers. Notice, that the RNG of C generates all sensitive random tokens fast, during checking, and therefore needs protection. On the other hand, the RNG of S generates random tokens slowly, whenever a user authenticates with their password.

Isolating the RNG of C enables Lethe to utilize a single server for performing both authenticating a user into the system and checking for a data breach. By doing so, the network communication overhead discussed in Section 5.1, is eliminated since there is no need for Lethe to transfer updates of files F and L to an external server.

7. Related Work

Several techniques for making off-line password guessing harder exist. However, all of them impose significant performance limitations [8]. In particular, machine-dependent functions have poor scalability [24], distributed cryptography techniques require client-side system changes which is not user friendly [25], and external password-hardening services are subject to a single point of failure [26].

Contrary, an interesting direction, initially proposed by Juels and Rivest, is to deploy honeywords, which are false passwords associated with each user's account, for detecting password leakage [2]. When using honeywords, even if an attacker steals and reverts the password file F, containing the users' hashed passwords, they must still decide about the real password from a set of 20 distinct sweetwords. Using a honeyword to login sets off an alarm as a data breach incident has been reliably detected. For generating those honeywords, Juels and Rivest proposed four random replacement-based HGTs, which have been later shown to be ineffective to meet the expected security requirements [1].

Bojinov et al. [27] suggested to hide the real password file F amongst others, which are similar to F but decoy ones. For constructing these decoy password files, they proposed a syntax-based HGT in which honeywords are generated using the same syntax as the real passwords. In particular, the authors parse each password into a series of tokens containing consecutive characters of letters, digits, or special characters, and then, replace each character with a randomly selected one that matches the token's type.

Erguler suggested an alternative HGT that selects the honeywords from existing user passwords [28]. However, the effectiveness of Erguler's HGT is yet to be verified using the relevant metrics, i.e., flatness and success-number graphs, while also suffering from significant shortcomings, such as critical deployment issues, low robustness against the "peeling-onions style" distinguishing attack, and limited honeywords generation spectrum [1].

Dionysiou et al. [8] introduced HoneyGen, a practical and non-reversible HGT that causes state-of-the-art distinguishing attackers to fail. HoneyGen leverages representation learning techniques to generate realistic looking honeywords, and takes advantage of the stochasticity of the utilized ML models to ensure its non-reversibility property. In doing so, HoneyGen meets the expected security requirements in terms of flatness and success-number graphs. Finally, HoneyGen supports the generation of honeywords with arbitrary length and structure.

Almeshekah et al. [24] proposed a machine-dependent function, namely a hardware security module, to be deployed in the authentication server S (target) for producing the hashes of the given passwords. Thus, an attacker who is unaware of this defence mechanism and tries to crack its database off-line will produce plausible decoy passwords that, when submitted, alert the target site to its breach.

Kontaxis et al. [15] proposed SAuth, a protocol that employs authentication synergy among different services. In this context, the authors suggest the use of decoy passwords to tackle the problem of password reuse, which can cause a system to fail if a user recycles the same password across all vouching services [15]. The key difference between decoy passwords and honeywords is the fact that any of the decoys can successfully authenticate the user to the service, whereas the use of a honeyword sets off an alarm as an attack has been detected [2].

Various papers have suggested the use of decoy accounts, which are false accounts with no owner, that if ever accessed, signal a data breach alarm [29]. For example, DeBlasio et al. [30] proposed to register a decoy email, at a separate email provider, for each decoy account using the same email address and password. Thus, if a successful login attempt at the uncompromised email provider is observed for any of those decoy emails, then the system sets off an alarm. However, similar to all other works in the field, except from Amnesia, this schema places trust to a third party, which in this case is the email provider, for detecting data breaches.

The aforementioned papers require the use of an extra trusted component, such as a honeychecker [2], [8], [28], a machine-dependent function [24], or an external secure email provider [30], whose state is assumed to remain secret even after the attacker breaches the target.

The first approach which is free of this, rather strong, assumption, is the framework proposed by Wang et al. [7], namely Amnesia. Amnesia is a honeywords-based breach detection framework that allows the target to *probabilistically* detect its own data breach. Furthermore, Amnesia enables the target to monitor for the entry of passwords stolen from it at other sites, called monitors. Using their set-up, 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. The authors introduce a cryptographic protocol, namely PCR, with which a monitor transfers the password attempted in an unsuccessful login to the target, but only if the attempted password is one of the sweetwords for the same account at the target.

8. Conclusion

In this paper, we proposed Lethe, a honeywords-based data-breach detection framework that is not dependent on an always trusted external entity for detecting the entry of a honeyword and on any persistent secret state for validating login attempts. Lethe guarantees the detection of a data breach incident with the same probability as in the classic honeywords paradigm, that is, $19/20$ in case 20 sweetwords are used for each user account.

Furthermore, Lethe bounds the time an attacker can use breached credentials to access accounts without alerting the target to its data breach. In Lethe's context, if an adversary logs into the system using a sweetword that is different from the one given by the legitimate user during the registration phase, a data breach is guaranteed to be detected in a few hours. Finally, Lethe allows for adversaries that can gain full access to the honeywords generation model. Such adversaries, however, cannot harm Lethe since the deployed honeywords generation model makes it impossible to reproduce the same set of sweetwords for any given password.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank the shepherd for helping us to improve the final version of this paper. In addition, we thank Chrystalla Anastasiou for her assistance in the implementation of the prototype. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 830929 (CyberSec4Europe) and the Marie Skłodowska-Curie grant agreement No. 101007673.

References

[1] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A security analysis of honeywords," in *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018, pp. 1–16.

[2] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013, p. 145–160.

[3] R. Hackett, "Yahoo raises breach estimate to full 3 billion accounts, by far biggest known," 2017. [Online]. Available: https://fortune.com/2017/10/03/yahoo-breach-mail/

[4] P. Heim, "Resetting passwords to keep your files safe," August 2016. [Online]. Available: https://blog.dropbox.com/topics/company/resetting-passwords-to-keep-your-files-safe

[5] P.-H. Kamp, P. Godefroid, M. Levin, D. Molnar, P. McKenzie, R. Stapleton-Gray, B. Woodcock, and G. Neville-Neil, "LinkedIn Password Leak: Salt Their Hide," *ACM Queue*, vol. 10, no. 6, p. 20, 2012.

[6] C. Cadwalladr and E. Graham-Harrison, "Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach," *The Guardian*, vol. 17, p. 22, 2018.

[7] K. C. Wang and M. K. Reiter, "Using amnesia to detect credential database breaches," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 839–855.

[8] A. Dionysiou, V. Vassiliades, and E. Athanasopoulos, "Honeygen: Generating honeywords using representation learning," in *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security*, 2021, pp. 265–279.

[9] C. Shi and H. Sun, "Honeyhash: Honeyword generation based on transformed hashes," in *Proceedings of the 25th Nordic Conference on Secure IT Systems*, 2020, pp. 161–173.

[10] R. Wash, E. Rader, R. Berman, and Z. Wellmer, "Understanding password choices: How frequently entered passwords are re-used across websites," in *Proceedings of the 12th Symposium on Usable Privacy and Security*, 2016, pp. 175–188.

[11] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th International Conference on World Wide Web*, 2007, p. 657–666.

[12] "IBM registration form." [Online]. Available: https://www.ibm.com/account/reg/us-en/signup?formid=urx-46542

[13] "2018 Credential Spill Report." [Online]. Available: https://www.coursehero.com/file/107589941/Shape-Credential-Spill-Report-2018pdf/

[14] W. J. Burns, "Rockyou password leak," January 2019. [Online]. Available: https://www.kaggle.com/wjburns/common-password-list-rockyoutxt

[15] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis, "Sauth: Protecting user accounts from password database leaks," in *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013, pp. 187–198.

[16] H. Saleem and M. Naveed, "Sok: Anatomy of data breaches," in *Proceedings of the 20th Conference on Privacy Enhancing Technologies*, 2020, pp. 153–174.

[17] R. Dhamija, J. D. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems*, 2006, pp. 581–590.

[18] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *Proceedings of the 21st Network and Distributed System Security Symposium*, vol. 14, no. 2014, 2014, pp. 23–26.

[19] S. Pearman, J. Thomas, P. E. Naeini, H. Habib, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, and A. Forget, "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017, pp. 295–310.

[20] C. Wang, S. T. Jan, H. Hu, D. Bossart, and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, 2018, pp. 196–203.

[21] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki *et al.*, "Data breaches, phishing, or malware? Understanding the risks of stolen credentials," in *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017, pp. 1421–1434.

[22] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 86, pp. 1–118, 2016.

[23] "12th Generation Intel Core^{TM} Processors, Datasheet, Volume 1 of 2," January 2022, https://cdrdv2.intel.com/v1/dl/getContent/655258.

[24] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, "Ersatzpasswords: Ending password cracking and detecting password leakage," in *Proceedings of the 31st Computer Security Applications Conference*, 2015, pp. 311–320.

[25] J. Camenisch, A. Lehmann, and G. Neven, "Optimal distributed password verification," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 182–194.

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

[27] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *Proceedings of the 15th European Symposium on Research in Computer Security*, 2010, pp. 286–302.

[28] I. Erguler, "Achieving flatness: Selecting the honeywords from existing user passwords," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 2, pp. 284–295, 2015.

[29] I. Mokube and M. Adams, "Honeypots: concepts, approaches, and challenges," in *Proceedings of the 45th ACM Southeast Regional Conference*, 2007, pp. 321–326.

[30] J. DeBlasio, S. Savage, G. M. Voelker, and A. C. Snoeren, "Tripwire: Inferring internet site compromise," in *Proceedings of the 17th Internet Measurement Conference*, 2017, pp. 341–354.