Thesis Dissertation

IMPLEMENTING DATA BREACH DETECTION WITH LIMITED SECRET STATE

Chrystalla Anastasiou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

June 2022

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Implementing Data Breach Detection with Limited Secret State

Chrystalla Anastasiou

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

June 2022

Acknowledgements

First and foremost, I would like to warmly thank Dr. Elias Athanasopoulos, my supervisor, for the support, guidance, and advice he provided throughout the development of this dissertation.

Furthermore, a huge thank you to my family that constantly encouraged me to keep going and not lose myself in the pressure and difficulties I phased in my way. Especially, I would like to thank my parents for the support they provided throughout my studies. Also, I would like to thank my friends, inside and outside of the University, that have been supporting me through all this time.

Also, I would like to thank my friend and classmate Katerina Erodotou for all the motivation she gave me during the countless hours we devoted together on our studies and, most of all, for all the beautiful moments we shared that I truly cherish.

Finally, I want to thank my best friend Max for lovingly watching over me.

Abstract

Despite all their known vulnerabilities, passwords remain the most common basis for user authentication. Data breaches often take some months until discovered. In this thesis we propose Lethe; a deterministic honeywords-based data breach detection framework that detects a database leakage timely.

Honeywords are essentially decoy passwords that are associated with each user account. Lethe utilizes a Honeyword Generation Technique (HGT) that leverages Machine Learning to produce honeywords. This implies that the HGT ensures its *non-reversibility* property. This means that even inserting the same password in the HGT will not result in the production of the same honeywords. Hence, the adversary is unable to revert the algorithm and tell which is the real password, even if fully compromising the HGT. In fact, Lethe does not assume of a trusted HGT, contrarily to all previous efforts.

Additionally, the only person that knows their genuine password (i.e., the one used during registration), is the user themselves. Our system is unaware of the real password of the user. Fundamentally what happens is that Lethe keeps a record of all logins that occur during a day and replays these logins in an auxiliary server which will detect if different honeywords were used under the same account. In that case, the server sets of an alarm since a data breach has been reliably detected.

Currently implemented approaches that utilize honeywords are based on the need of a trusted component to distinguish the real password out of the sweetwords. In contrast, Lethe keeps a zero persistent secret state and requires no trusted components, other than a trusted bootstrap. Similar efforts also require the need of a legitimate user action in order to signal an alarm. Opposed to these approaches, Lethe can detect a data breach even if a user has logged in just once during registration. In addition, Lethe is a deterministic approach instead of detecting a data breach probabilistically like previous similar attempts do.

In this thesis, we provide the first fully functionate prototype of Lethe. We explain Lethe's architecture and describe the system's implementation in great depth. We then further evaluate Lethe in terms of security and prove that Lethe is effective and ensures the detection of a data breach without the use of a trusted component, nor any actions taken by the legitimate users.

Contents

Chapter 1

Introduction

Traditional passwords still remain the most widespread technique used for user authentication since they are an inexpensive, uncomplicated, and convenient method to use. Even though passwords are considered to be an exceptionally poor form of security and are generally known for being vulnerable to attacks [9], [34], it seems likely that passwords will stay prevalent for some time, despite some suggestions to replace them [7]. Despite the fact that there is a plethora of counterattack mechanisms [5], [27], their use is still limited due to the complexity in terms of implementation.

The password-focused authentication systems that are currently used, store the cryptographic digest of the passwords of users in a database. The database containing this sensitive information is an extremely appealing target for attackers. If retrieved and restored in plain text, these passwords can be sold and posted online [32] with the adversary staying undetectable.

Note that there is usually a significant delay between the data breach and its discovery [18], [19], and it might take some months for the breach to be detected [2], [28] when the attackers have already exploited the passwords. Few of the biggest data breaches in recent history are those of Yahoo [17], Alibaba [25], LinkedIn [22], Facebook [8] and Target [30] which are some of the most popular web services out there. Thanks to these compromises, millions, even billions in the case of Yahoo and Alibaba, of passwords were leaked, leaving the companies with great damage in terms of revenue and brand reputation.

The fact that password leakages take such a long time to be detected, is the reason why Dionysiou, Vassiliadis and Athanasopoulos [13] propose Lethe; a system that will be able to detect a data breach timely, as early as at the end of the day. This approach makes use of honeywords, initially proposed by Juels and Rivest [21]. Honeywords are essentially false passwords that are associated with each user's account and are injected in a database for the sole purpose of detecting a data breach. Hence, when an attacker compromises the database that has the hashed passwords of users, even with cracking and recovering the

passwords, they cannot tell which of the sweetwords is the real password. The adversary must still decide between k different sweetwords, where only one of those is the real password. The attempted submission of a honeyword to login will be detected by an auxiliary server i.e., the checking server C. C will set off an alarm when a honeyword is used, as a data breach has been detected.

To address this major password leaks and their late detections, various methods have been implemented. Currently implemented approaches are based on the need of a trusted component to distinguish the real password out of the k sweetwords [11], [14], [21], [26], [29]. If, however, this trusted component is compromised, honeywords become useless in identifying a data breach.

One approach that does not require a secret state to detect the injection of a honeyword is Amnesia, proposed by Wang et al [33]. Amnesia is a framework that uses a probabilistic model verification. In summary, this framework utilizes a marking technique where the most recently used sweetword is marked. The rest of the sweetwords are marked separately with a specific probability. In the case where the adversary logs in an account using a honeyword (i.e., a sweetword that is not the user chosen password), then the real password of the user gets unmarked. Consequently, the next time the legitimate user logs in, giving a password that is not marked, an alarm sets off since a data breach has been reliably detected. However, previous research has shown that users seldom authenticate themselves by inputting their credentials explicitly [15], [35]. Thus, the fact that Amnesia identifies a breach only after users verify their credentials and assuming that they are not using established cookies is a strong assumption.

Furthermore, whilst Juels and Rivest [21] were the first to propose the use of honeywords, their approach's success relies on various assumptions based on the resulting honeywords. The effectiveness of honeywords depends on the indistinguishability of the real password [32]. If the attacker can easily determine the user chosen password out of the k distinct sweetwords, consequently, this method fails. This is the reason why Dionysiou, Vassiliadis and Athanasopoulos [13] highlight the need for a Honeyword Generation Technique (HGT) that can meet the expected security requirements [21], [32]. In particular, the authors introduce HoneyGen, a HGT that generates realistic looking honeywords using representation learning. Also, a HGT should ensure its *non-reversibil*ity property [32]; reversing the algorithm should be computationally hard. HoneyGen uses machine learning technologies that by nature ensure this property.

This research problem is important since, as we said, there is a need for methods that can timely detect a data breach, in order to protect companies and most importantly the sensitive information of users. Detecting password leakages before an attacker gets the opportunity to exploit the users' information provides safety to both users and web services. Therefore, implementing such systems will have a great impact in today's vulnerable world.

This thesis makes the following contributions.

1. We design and implement the first prototype of Lethe [12], a system that utilizes honeywords to detect data-breaches. Lethe keeps a zero persistent secret state and requires no trusted components, other than a trusted bootstrap. For the honeyword generation, we use HoneyGen [13] to guarantee that the attacker, when compromises the database and cracks the cryptographic digests, cannot distinguish the real password out of the k sweetwords. Hence, the adversary has no more than 1/k successful attack rate.
2. We further evaluate Lethe, by impersonating the adversary's behavior. We attack our system by simulating logins using random sweetwords on a database that contains 10000 users and their generated honeywords. We do a true verification of the system and discover that Lethe manages to detect 95% of all data breaches. Hence, only 5% of all data breaches are successful, revealing that the adversary cannot achieve more than the random choosing success rate.

The remainder of this thesis is structured as follows. In Chapter 2, we discuss about the preliminaries and the concepts that are utilized throughout this thesis. In Chapter 3, we analyze the architecture of Lethe in depth. Chapter 4 follows, in which we provide the detailed implementation of the first prototype of Lethe. In Chapter 5, we evaluate our prototype and in Chapter 6 and 7 we discuss about related and future work respectively. Lastly, the conclusion of this work can be found in Chapter 8.

Chapter 2

Background

Certain background knowledge is required in order to have a better understanding of this thesis. In this section, we first explain what password authentication is and how data breaches work. Next, we discuss about different Honeyword Generation Techniques (HGTs) and their limitations. Then, we analyze HoneyGen; the HGT used to implement Lethe.

2.1 Passwords

2.1.1 Password Based Authentication

Passwords are still the most widely used user authentication method since they are a cheap, non-complex, and simple technique to implement and use. Essentially, password-based authentication is the process of acquiring access and retrieving a user's entitled information by providing a username and password that are unique for each account.

The password-focused authentication systems that are currently used, store the cryptographic digest of the passwords of users in a database. Fundamentally what happens is that a user first provides their password during registration. Then, the hashed password is stored in the database and not the plain text of the password itself. A system maintains a sensitive database that contains the hashed passwords of all users that are registered in the system.

When logging in, a user provides their password. Then, the cryptographic digest is again calculated and compared with the one that is stored in the database. If the two match, then the user is gained access to the system, else access is denied.

2.1.2 Data Breaches

Traditional passwords still dominate the authentication landscape, despite their well-known security vulnerabilities. In fact, passwords are a major point of weakness in computer security [9], [34]. As we previously mentioned, a system maintains a sensitive database that contains the hashed passwords of all users that are registered in the system. The database containing this sensitive information is an extremely appealing target for attackers. If retrieved and restored in plain text, these passwords can be sold and posted online with the adversary staying undetectable.

Credential database breaches have become a widespread security problem. Following a successful database leak, the adversary now has the cryptographic digests of the passwords and their corresponding hashing function. These are enough for the attacker to attempt to crack the passwords in various ways [23], [34]. For example, they can do a brute-force or an offline dictionary attack [16], [27]. In the former, the attacker creates all the possible permutations in the given password space and see if it matches any of the hashes in the leaked dataset. If it does match, then that user password is revealed. In the latter, the attacker uses extensive lists of the most used passwords, popular pet names, fictional characters, or literally just words from a dictionary. Then, similarly as in the brute-force attack, the adversary computes a list of the aforementioned hashes and systematically checks if any word in the list matches any of the hashes in the leaked dataset.

2.2 Honeywords

We previously mentioned, that honeywords are false passwords associated with each user’s account [21]. Their effectiveness relies on the indistinguishability of the real password.

Honeywords fail if the adversary can clearly find out the user chosen password out of the k different sweetwords [32].

2.2.1 Honeyword Generation Techniques

To evaluate the effectiveness of a HGT, Wang et al. [32], proposed two new metrics: *flatness graph* and *success-number graph*. These two measures assess a technique's resilience against the adversary's ability to distinguish the real password from the rest of the sweetwords in both the average and the worst-case scenarios.

Note that, Juels and Rivest [21] recommend using 20 sweetwords for each account hence, k = 20.

A *flatness graph* illustrates the likelihood of recognizing the real password against the number of login attempts using a sweetword, inferring that a point *(x, y)* in this graph demonstrates that the genuine password is found in the first *x* attempts with a probability *y*. Ideally, a perfect method does not allow more than *x* / 20 success rate.

A *success-number graph* plots the total number of logins using a real password against the number of logins using any of the rest of the honeywords. Therefore, a point *(x, y)* in this graph indicates that an attacker could successfully detect *y* real passwords before the x^{th} honeyword login attempt. In the best-case scenario, a HGT produces k sweetwords, where each one has the same likelihood (i.e., 1 / 20) of being the real password.

In addition, the *non-reversibility property* should be guaranteed by any HGT, i.e., reversing the algorithm should be computationally inefficient or impossible.

2.2.2 HoneyGen: Lethe's HGT

For the honeywords generation we utilize HoneyGen, proposed by Dionysiou, Vassiliadis and Athanasopoulos [13]. HoneyGen satisfies the aforementioned security requirements (*flatness* and *success-number graphs*) [21], [32].

Particularly, HoneyGen is a highly robust honeywords generation framework, that leverages representation learning to generate high-quality honeywords resulting in the adversary unable achieve more than the random choosing success rate. Also, HoneyGen uses machine learning technologies that automatically ensure the *non-reversibility property*.

Essentially, HoneyGen is a hybrid HGT and has two phases. The first phase trains the ML model on a real-world dataset of passwords. This allows the model to get familiar with the input's form and generate a word embedding for each password. Then, they leverage this ML model to get the top-k nearest neighbors of a password. At the second phase, they generate out-of-vocabulary honeywords using a chaffing-by-tweaking technique.

In the paper, the authors include HoneyGen's *flatness* and *success-number graphs* for different numbers of sweetwords per user. By observing these results, we can see that HoneyGen is very close to the random guessing baseline, which is the optimal solution. From further research, the authors concluded that HoneyGen has a very high resistance against sophisticated attackers, since a big percentage of the respondents could not distinguish the real password out of the generated honeywords. All these indicate that this HGT meets the expected requirements [21], [32].

Chapter 3

Architecture

In this section we outline the architecture of Lethe [12]. Lethe makes use of two servers; the authentication server S and the checking server C, which are responsible for the authentication and validation of logins, and the data breach detections respectively. The two servers communicate with the TLS Protocol. Compromising either server does not reveal any information about the real user passwords since neither server is aware of the real passwords of the users.

3.1 Lethe

At the start of Lethe's operation, the two servers initialize a cryptographically secure random number generator (CSRNG). Both servers initialize a CSRNG with a same random seed which is later discarded. That makes the two CSRNGs synchronized. We call that step the trusted bootstrap phase.

After the trusted bootstrap phase, Lethe begins to perform the authentication and the data breach detection phases.

During the authentication phase, the authentication server S keeps track of and saves all the user logins. The authentication phase is triggered every time a user logs into the system, and each time, it sets off the CSRNG of S (*Rs*), which gives the new position of

the used password. The rest of the sweetwords are then shuffled. The random value released is discarded after it's been used, and server S adds to the record that a user has logged in the system (their user ID along with the position of the password used, before the permutation).

At the end of each epoch, the Data Breach Detection phase occurs, where the checking server C checks if there has been a data breach. Since server S has recorded all the login events, and the two servers have synchronized CSRNGs, server C is able to replay these events and check whether a user used the same sweetword for all their logins. C does that using its own CSRNG (*Rc*) that has been initialized with the same seed as *Rs* during the trusted bootstrap. If different sweetwords have been used under a single account, then there has been a data breach.

We expand each phase in more depth below.

3.2 Lethe's Phases

As we said, there are four main phases; the registration phase, the authentication phase, the data breach detection phase, and the password reset. The authentication server S is responsible for the registration and authentication phases and checking server C does the data breach detection phase at the end of each epoch. Lethe's operation requires a trusted bootstrap preceding those phases.

3.2.1 Trusted Bootstrap

The trusted bootstrap initializes the system. During this phase, each server initializes a cryptographically secure random number generator (CSRNG). Both servers use the same sensitive seed to initialize their CSRNGs and the seed is then discarded. Consequently, the two CSRNGs (i.e., *Rs* and *Rc*) are synchronized.

3.2.2 Registration Phase

The registration phase occurs every time a user registers in the system. During this phase, (as shown in Figure 3.1) a Honeyword Generation Technique (*HGT*) is used. The *HGT* is fed with the real password given by the user, and results in the production of 19 honeywords based on it. The 20 sweetwords (19 honeywords plus the real password) are then stored to the authentication server S's database (*DBs*) along with the user's ID. Furthermore, the user ID and the slots "a" through "t" are stored in the checking server

C's database (*DBc*). By using slots instead of the actual sweetwords, we emphasize once again that the checking server does not even know what the real password is, neither what the sweetwords of the user are.

After the registration phase, the user's ID and given password are used to log the user into the system and that triggers a login event. As we mentioned, all login events set off the authentication phase, which is explained thoroughly below.

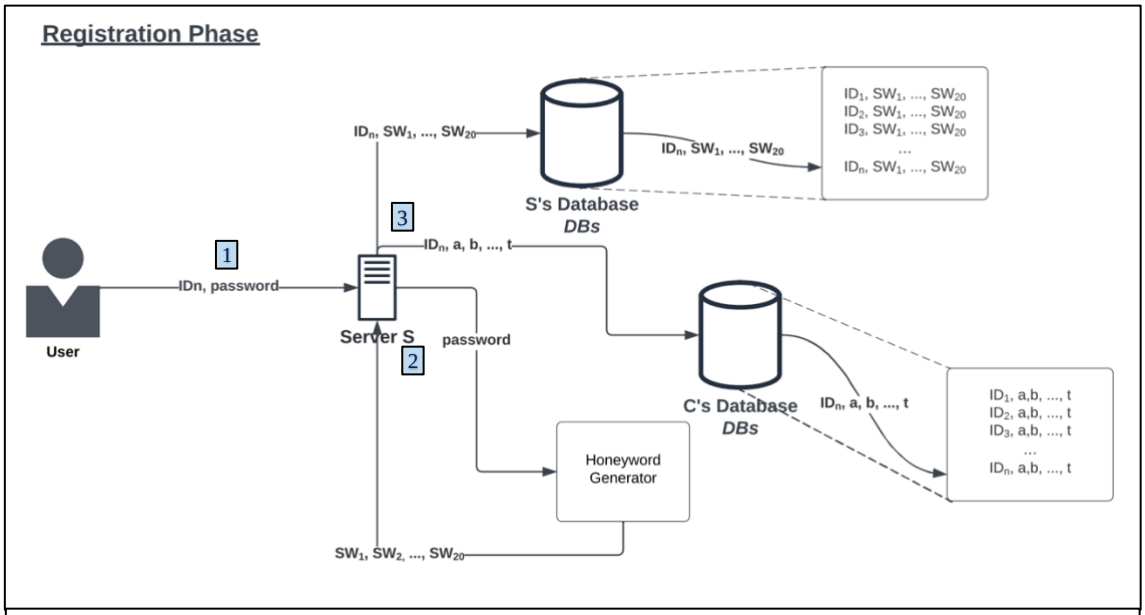

Figure 3.1 Registration phase

1. The user registers in the system by giving their password
2. *S* gives the password to a Honeyword Generator and gets 20 sweetwords as a response (19 honeywords and the 1 real password)
3. The new user is saved to *DBs* along with the 20 sweetwords, and to *DBc* along with the slots "a" through "t"

3.2.3 Authentication Phase

The authentication phase is invoked anytime a login event occurs. During this phase, which is shown in Figure 3.2, a user is attempting a login by giving their password, thus, the authentication server S receives a user ID and the password given by the user. Server S then uses that ID to retrieve the list of sweetwords (*sws*) associated with that user from its database. If the *sws* received does not contain the password given by the user, then the user is declined access to the system. However, if the given user password is one of the sweetwords in *sws*, then, Lethe removes the sweetword used from *sws*, shuffles *sws* and places the sweetword used to the position received by the authentication server S's CSRNG (i.e., *Rs*). Finally, the user is approved access to the system. Once the login has

completed, the random value received by *Rs* is discarded and the event along with the initial position of the password used is recorded in logins file L and the action of this event (e.g., “user 1 logged in the system using password in position 3” would be “1, 3, login”).

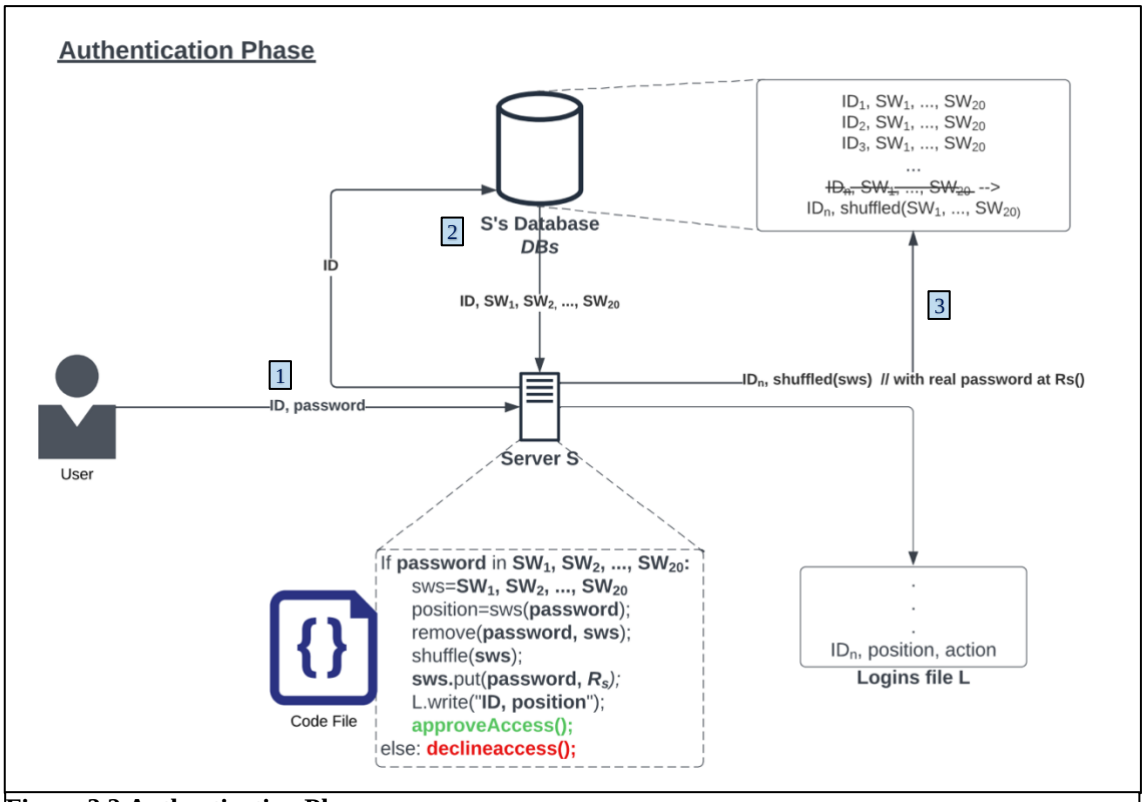

Figure 3.2 Authentication Phase

1. The user tries to log in the system giving their password
2. S retrieves the user’s sweetwords from *DBs* using the user’s ID.
   If the password given is not one of the sweetwords, then the user is declined access to the system. Else, the password used is placed at the position *Rs* as derived from S’s CSRNG and the rest of the *sws* is shuffled.
3. Then, *DBs* is updated with the shuffled *sws*, and the login event is recorded in Logins file *L*.

3.2.4 Data Breach Detection Phase

This phase happens at the end of each epoch. As you can see in Figure 3.3, during this phase, the checking server C replays all the logins contained in the logins file L and checks if more than one sweetword was used under a single account. If yes, then a data breach has been detected. Notice, that C has access to the S’s database, which is shuffled at this point, but, also, has its own database that has stayed untouched since the previous data breach detection occurred. Furthermore, C sustains a dictionary *last_used_sw* where it stores the last used sweetword/slot for each user (e.g., {“1”: “c”})

More thoroughly, checker C extracts the events one by one from the logins file L. The logins file contains a log with all the users (IDs) that logged in the system during this epoch along with the initial position of the sweetword used. For each login in the file, C replays the login on its own database. Firstly, C retrieves the user's slots from its database (*DBc*) and gets the slot at the said position (*P*) from the logins file. Considering that *DBc* is untouched since the last data breach detection, then the slot at the position *P*, represents the sweetword used. The next step is to check whether the user and their last slot used exists in the *last_used_sw*. If they do not, then we add them to the dictionary along with their sweetword/slot used. If they do exist in *last_used_sw*, then we just check if the slot used in this current login is the same as the last used slot. If the two slots are identical, then the user did not use two different sweetwords and therefore we proceed to the next login, else an alarm is triggered since a data breach has been detected. Provided that no attack has been detected, then before moving on to the next login, we simulate the current login to *DBc*. Thus, we remove the slot used from the retrieved slots, shuffle them, and place the slot used to the position received by the checking server C's CSRNG (i.e., *Rc*). Substantially, what happens is that the two CSRNGs stay synchronized because C replays all the events that happened in S during an epoch. And this can occur because the two CSRNGs (i.e., *Rs* and *Rc*) are initialized with the same seed. Therefore, only the server C can replay those logins.

At the end of the data breach detection phase, what also happens is an overwrite of the L file with a simulation of one login for all users in this epoch using *Rs*. This implies that, even with just one login in the system, a user along with their real password is propagated in each data breach detection phase.

Notice that, as we said in *3.2.2*, *3.2.3* and above, during registration, the authentication phase is triggered, and the authentication phase keeps the log of all logins in file L. C then takes the L file and replays all the logins during the data breach detection phase. That along with the fact that we simulate logins at the end of the data breach detection phase, means that even if when a user has logged in during registration phase, then that password used, basically the real password, is passed on in all the rest of the epochs.

Furthermore, in case of a password reset ("pReset" action), we reset the slots under that user.

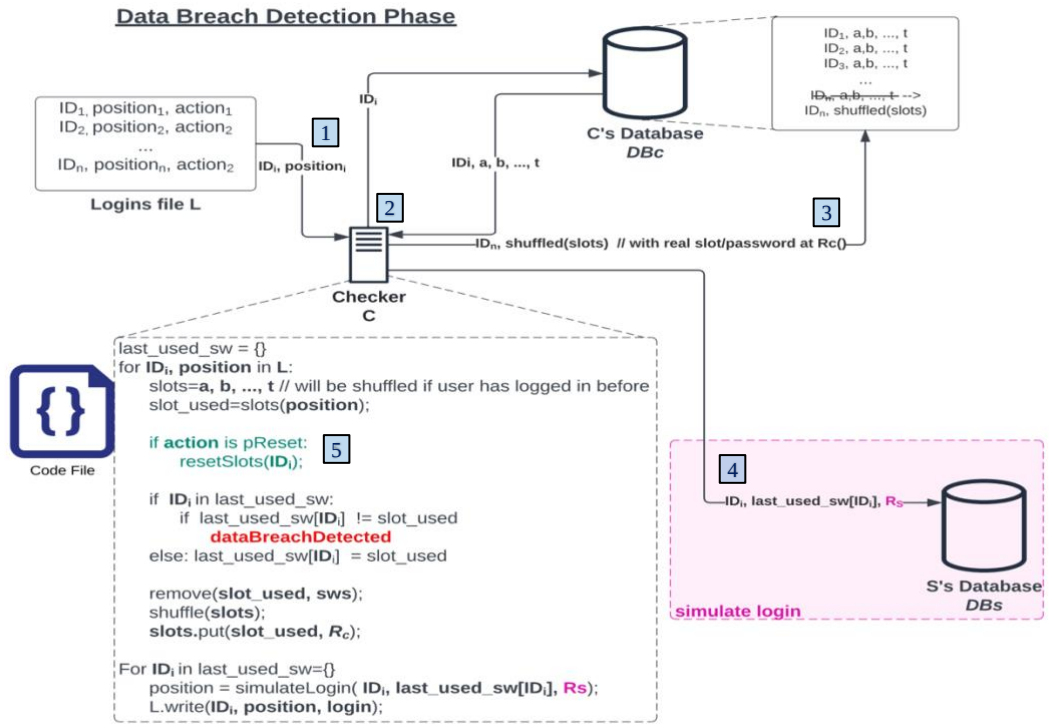

Figure 3.3 Data Breach Detection Phase

1. First, the checking server *C* takes the logins events (recorded during the authentication phase) from Logins file *L*, one by one
2. Then, for each login, *C* retrieves the user's slots from *DBc* using the user's ID. If the slot given is not one of the saved in *last_used_sw*, then we save it. Else, if the current slot used is not the last used slot, then a data breach has been detected else the slot used is placed at the position *Rc* as derived from C's CSRNG and the rest of the slots are shuffled.
3. Then, *DBc* is updated with the shuffled slots
4. After all the logins have been through the data breach detection phase, we simulate a login for every user in *last_used_sw*
5. Notice that, in case of a "*pReset*" action, the slots of the user are rebooted

3.2.5 Password Reset

The Password Reset action essentially does the registration phase again for the user, by resetting their sweetwords and slots in both *DBs* and *DBc.*

Remember that we said in *3.2.3*; "the event along with the initial position of the password used is recorded in logins file L along with the action of this event". In case of a password reset action the action recorded in L is "*pReset*". First, of course, the system will provoke a registration phase, therefore will feed the new password to a *HGT,* and produce new 19 sweetwords for the user. Assuming a "*pReset"* action, the Checker will receive a line in this format in its *L* file; "user ID, position of new password, pReset". Then, during the

data breach detection phase, when the Checker detects that the action is *pReset*, it resets the slots of the user.

However, naturally, for any password reset event to occur, one must first login into the system. In case an attacker has gotten access to the sweetwords and uses one of them to log into the system, if the sweetword used is not the real password, then, at the end of the epoch when the data breach detection occurs, a data breach will be reliably detected. So, even if the attacker changes the password, the data breach will be detected during the following data breach detection. But in case the attacker uses the real password of the user, and that is a 1/20 = 5% chance, then that would be a successful attack.

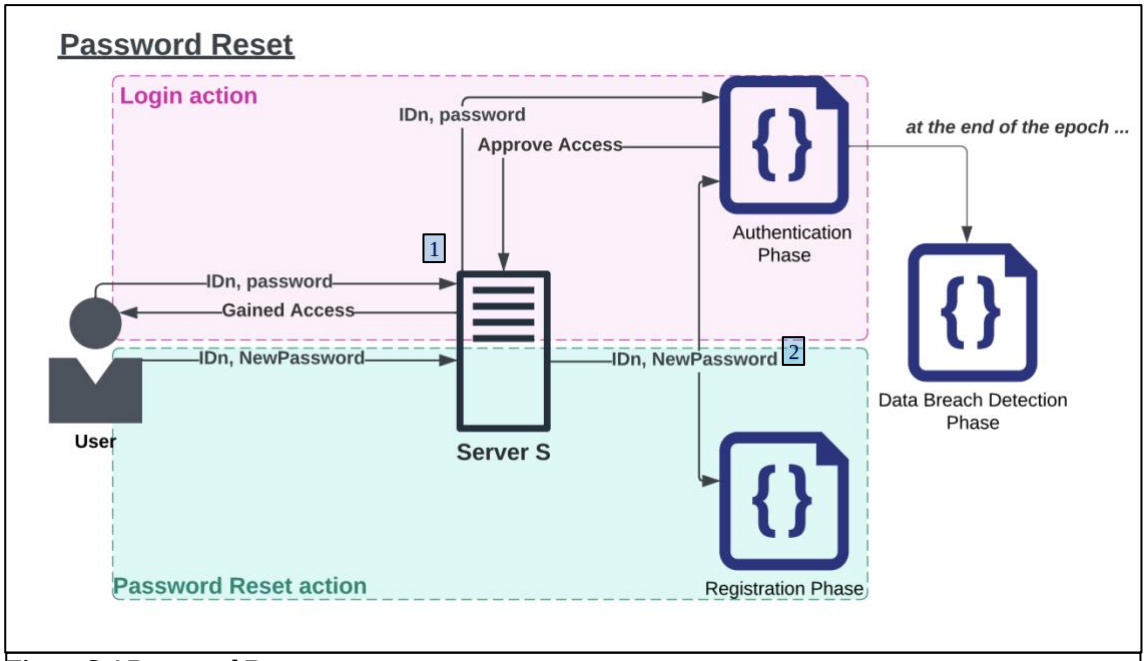

Figure 3.4 Password Reset

1. Firstly, a user must log into the system to be able to invoke password reset event
2. After successfully logging in, when a user requests a password reset, they shall provide the new password. This will trigger the registration phase (thus, changing the *DBs*), which will also set off the authentication phase (since it will commit a *pReset* event) and the action will also reset the slots in *DBc* during the next data breach detection phase

Chapter 4

Implementation

To implement Lethe, as described in Chapter 3, we wrote a python system that supports multiple functions that when combined, built the Lethe system. We show and explain the implementation of each function that is a part of this system below.

4.1 RC4 as the Cryptographically Secure Random Number Generator

As we mentioned, both the Server S and the Checker C have their own Cryptographically Secure Random Number Generator (CSRNG). S's and C's CSRNGs must be always synchronized.

We choose to implement RC4 as the used CSRNG, because it is significantly easy to use, therefore, it makes the implementation less complicated. Furthermore, because of its simplicity, RC4 is a faster and more efficient cipher [3].

Since S's and C's RC4 implementation are identical, we only show the implementation for S, i.e., *Rs*.

The steps of RC4, in general, are to first prepare a key array and then hand that key array to a Key Scheduling Algorithm (KSA). As shown in Listing 4.1, our RC4 implementation also creates the key array, consequently creating the Unicode of the seed given. Then, the key array is given to KSA, which returns the initial permutation of S. In KSA, we shuffle S according to the key, therefore we produce a random permutation of S (of all numbers between 0 and 255).

```
1   def preparing_key_array(s):
2       return [ord(c) for c in s]
3
4   def KSA(key):
5       keylength = len(key)
6
7       # Initially we have one list S that has all bytes from 0 to 255
8       S = list(range(256))
9
10      j = 0
11      # Initial permutation of S
12      for i in range(256):
13          j = (j + S[i] + key[i % keylength]) % 256
14          S[i], S[j] = S[j], S[i]  # swap
15
16      return S
```

Listing 4.1 preparing_key_array and KSA functions

Finally, we implement the Pseudo-random generation algorithm (PRGA) to get the wanted keystream. The PRGA is a loop that runs n times, where n is the fixed number of keys wanted. Each time, we get a byte from S. As shown in Listing 4.2, each time when calculating the "i" and "j", we swap them in S. Consequently, each time we get something from S, we alter it, and it is never the same again.

For our purposes, in our implementation (see Listing 4.2) we slightly transform RC4, to return each key one by one instead of giving a fixed number n for the number of keys wanted. Since we do not actually know how many users will be doing a login in the system at each epoch, we needed a function that would not return a fixed number of keys but give each key one by one anytime it is requested. The first step, which is equivalent to the first time a key is asked, is to give the seed to the RC4 algorithm. Then, each time a key is needed, we calculate and return a single key, the first key of a key stream the normal RC4 algorithm would return.

For example, the original RC4 algorithm would be given a length *n* and return a key stream containing *n* keys, let's say [*key_1, key_2, …, key_n*]. Our algorithm will return *key_1* the first time that it is called, and any afterwards requests for a key will return *key_2, key_3* and so forth.

```
# get one by one key
def PRGA_one_by_one():
    global i, j
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    # swap
    S[i], S[j] = S[j], S[i]

    # the next byte
    K = S[(S[i] + S[j]) % 256]
    return K

# retrieve the first key, given the seed (for server S)
def rs(seed):
    global S
    seed = str(seed)
    seed = preparing_key_array(seed)
    S = KSA(seed)

    key = PRGA_one_by_one()
    return key

# retrieve the rest of keys (for server S)
def rs_rest():
    return PRGA_one_by_one()
```

Listing 4.2 RC4 Implementation to return keys one-by-one

4.2 Trusted Bootstrap

As we previously mentioned, the trusted bootstrap initializes the system. During this phase, each server initializes a cryptographically secure random number generator (CSRNG). The CSRNG implemented is the RC4 as explained above.

Both servers use the same random seed to initialize their CSRNGs which is then discarded. As shown in Listing 4.3 our seed is set as the current datetime, thus it is a value that is depended on current time. Therefore, it is a true random value taken from the environment, and not a pseudo random value (e.g., the result of *random()*). We use that as our true random seed to initialize both the Authentication Server (*Rs*) and the Checking Server (*Rc*).

Since both CSRNGs (i.e., *Rs* and *Rc*) are initialized with the same seed, consequently, they will be producing the same random values.

```
# This function initializes the system
# Each server initializes a cryptographically secure random number sequence
# Both servers use the same seed to initialize their CSRNGs and the seed is then
Discarded
# Consequently, the two CSRNGs (i.e., Rs and Rc) are synchronized.
def trustedBootstrap():
    # Calculate a true random seed based on the date
    seed = datetime.now()
    # Send seed to the checking server
    Checker.getSeed(seed)
    # Send seed to the authenticating server
    Authenticator.getSeed(seed)
    # Discard the seed
    del seed
    gc.collect()
```

Listing 4.3 Trusted Bootstrap

4.3 Registration Phase

As previously discussed in Chapter 3, the registration phase takes place each time a user signs up in the system. During this phase (see Listing 4.4 for pseudocode), a *HGT* (Honeyword Generation Technique) is used to produce 19 honeywords based on the real password the user has given for their registration. These, 20 now, sweetwords are stored in *DBs* (authentication server S's database) together with the user ID. Also, the user ID and the slots "a" through "t" are saved in *DBc* (checking server C's database). Then, the user is logged into the system with the user ID and the password given during the registration. This activates a login event, which will then consequently trigger the authentication phase.

Basically, what will happen is that when the authentication phase occurs, this first login/registration of the user is recorded in the logins file L, as previously mentioned in Chapter 3, Subsection 3.2.4. When the next data breach detection happens, the real password used to register/log in the user for the first time will be documented and passed on forever in all the following epochs (since, as we said, we simulate logins at the end of each data breach detection phase, see Subsection 4.5).

```
def registrationPhase(userId, password):
    sws = HGT(password);
    insertDBs(userId, sws);

    slots = "a, b, ..., t";
    insertDBc(userId, slots);

    authenticate(userId, password);
```

Listing 4.4 Registration Phase – pseudocode

4.4 Authentication Phase

We show the implementation for the authentication phase in Listing 4.5 and Listing 4.7. As previously stated in Chapter 3, Subsection 3.2.3, the authentication phase occurs each time a login event happens.

As shown in Listing 4.5, the authentication phase is given a user ID and a password. Then, that user ID is used by the function to retrieve the user's sweetwords (*sws*, produced and stored in *DBs* during the Registration Phase) if the user is not included in the *blacklist*. The *blacklist* is a list that contains all the users under which a data breach has been reliably detected. Hence, if the user is not in the *blacklist*, we retrieve their *sws* from *DBs* and check whether the provided password is one of the sweetwords contained in *sws*. If the password given is not contained in *sws*, subsequently the access in the system is declined. But, in the case where *sws* contains the given password, then the user is approved access to the system and consequently the system executes the functions in Listing 4.6 to update the row in *DBs* associated with the specific user.

```
1   # This function does the authentication phase for the given user IDS and passwords.
2   For each user, the function checks if the password provided is one of the sweetwords
3   in the database for that user, and if it is then it approves access, else it denies
4   it. Every EPOCHS logins, the function calls the Data Breach Detection phase.
5   def authenticationPhase(user_ids, passwords):
6     # a list where the accounts under which a data breach has been detected are stored
7       blacklist = []
8       # get the next random value from the CSRNG
9       rs = Rs.rs_rest() % Rs.LIMIT
10      # open the L.txt to store all the logins, for the Data Breach detection
11      f = open("L.txt", "a")
12
13      # For each login
14      for i in range(len(user_ids)):
15          if user_ids[i] in blacklist:
16              Continue
17          else:
18              # retrieve the user's sweetwords from S's database
19              sws = getSws(user_ids[i]);
20
21              # if password given is one of the sweetwords
22              if passwords[i] in sws:
23                 Listing 4.6 Authentication Phase - Update row of current user
24                 Listing 4.7 Authentication Phase – Data Breach Detection call at the
    end of the epoch
25
26              # get next Rs and approve access to system since pass is in sweetwords
27                  rs = Rs.rs_rest() % Rs.LIMIT
28                  approveAccess()
29          else:
30              # decline access to system since pass given is not one of the sweetwords
31                declineAccess() and exit
32      f.close()
```

Listing 4.5 Authentication Phase – Approve or Decline Access

If the given user password is one of the sweetwords in sws, then, as shown in Listing 4.6, Lethe first records the login in logins file L. After that, the sweetword used is removed from *sws*, *sws* is shuffled and the sweetword used is then placed to the position that was retrieved by *Rs*, which is the authentication server S's CSRNG as we stated earlier. Finally, the user is approved access to the system. (Listing 4.5)

Note that, each time we get the next value of Rs (e.g., line 27 in Listing 4.5), we mod the value with "*LIMIT*" so that we get a number between 0 and "*LIMIT - 1*". "*LIMIT*" is the number of sweetwords used for any user, thus, in our case 20, but this can change with every implementation by simply changing *LIMIT*.

```
1               action = "login"
2               # before shuffling row in database and placing the pass used at Rs
3               # position, write the user ID and the index of the password used in L.txt
4                    f.write(str(user_ids[i]), ',', str(sws.index(passwords[i]))
5                    f.write(',', action, "\n")
6                    # remove pass used, shuffle rest, insert pass in place given by Rs
7                    sws.remove(passwords[i])
8                    random.shuffle(sws)
9                    sws.insert(rs, passwords[i])
10
11                   # update row with new shuffled row
12                   updateSws(sws, user_ids[i])
```

Listing 4.6 Authentication Phase - Update row of current user

As above stated, and demonstrated in Listing 4.7, at the end of an epoch, the authentication phase invokes the data breach detection phase which is explained in great depth in the next section.

```
1               # if epoch has ended, check for data breach
2               if endOfDay:
3                    f.close()
4                    # return the accounts for which a data breach has been detected
5                    blacklist = Checker.dataBreachDetection().copy()
6                    # dataBreachDetection will overwrite L and move some checks for next
7                    f = open("L.txt", "a")
```

Listing 4.7 Authentication Phase – Data Breach Detection call at the end of the epoch

4.3 Data Breach Detection Phase

We have mentioned in the previous section that at the end of each epoch the data breach detection occurs. We will be explaining the implementation of the data breach detection step-by-step and thoroughly in this section.

```
# This function does the Data Breach Detection phase for each login in L.txt
def dataBreachDetection():
    # L.txt contains a log with all the users (ids) that logged in during this epoch
    f = open("L.txt", 'r')
    lines = f.read().splitlines()
    f.close()
    # a dictionary to store each user's last used slot
    last_used_sw = {}

    # for every login in L.txt
    for line in lines:
        user_id = line[0]
        pass_initial_position = line[1]
        action = line[2]
        # if the action is password reset then reset the row and
        # remove the user and last used slot from the dictionary
        if action == "pReset":
            resetRow(user_id)
            if user_id in last_used_sw:
                last_used_sw.pop(user_id)

        # get their slots from checker's
        sws = getSlots(user_id);
   # the slot used is the one at the initial position in the database of the checker
   # since checker's database has not been updated since the last epoch
        real_password = sws[int(pass_initial_position)]

        # if the user is in the dictionary
        if user_id in last_used_sw:
 # if last used sw is not the same as the one used then there has been a data breach
            if last_used_sw[user_id] != real_password:
                blacklist.append(int(user_id))
                print("A data breach has been reliably detected!\n")
            else:
                print("User logged in with real password.\n")
        else:
            # add the user to the dictionary along with their used slot
            last_used_sw[user_id] = real_password

        # remove the slot used, shuffle the rest of the slots and insert the
        # slot used at Rc as it was retrieved from checker's CSRNG
        sws.remove(real_password)
        random.shuffle(sws)
        sws.insert(rc, real_password)
        # get the next random value from the checker's CSRNG
        rc = Rc.rc_rest() % Rc.LIMIT

        # update the slots in the checker's database
        updateSlots(sws, user_id)
```

Listing 4.8 Data Breach Detection Phase – main functionality

During this phase (demonstrated in Listing 4.8), the checking server C iterates over all the logins in logins file *L* and replays them. Logins file *L* contains all the users that logged in during the last epoch along with the initial position of the password used (see Chapter 3, Subsection 3.2.3). Throughout this phase, the user ID, the initial position of the password and the action performed are extracted from the logins file *L* and for each of these users there is a check whether more than one sweetword has been used. If that is the case, then a data breach has been reliably detected.

Basically, what happens is that if the action extracted from *L* is a login (we expand the case of a "password reset" action in the next chapter), then C replays this login in its own database. Firstly, note that C keeps a dictionary where the last used slot is stored for each user (see *last_used_sw* in Listing 4.8). For each login, C checks whether this user exists in *last_used_sw* and if they do then it checks whether the current used slot is the same as the last used slot. If it is, then we move on to the next login, else if they are different then a data breach has been reliably detected and the account under which a data breach has been detected is added in the *blacklist*. In the case where the user does not exist in *last_used_sw*, then we simply add them to it along with their current used slot.

The next step is to simulate this login in the checking's server side too i.e., in *DBc*. Hence, and similarly as in the Authentication Phase (Chapter 4.2), we place the slot used at the position as retrieved by Cs (the checking server C's CSRNG) and shuffle the rest of the slots.

```
1   f = open("L.txt", "w")
2   # simulate logins for users that did only one login
3   for user in last_used_sw:
4       # get the next random value from the authenticator's CSRNG
5       rs = Rc.rs_rest() % Rc.LIMIT
6       # simulate login for that user using the last used slot as the password
7       position = simulate_login(user, last_used_sw[user], rs)
8       # add the user along with the position of the sweetword used in L.txt
9       f.write(user + "," + str(position) + "," + "login\n")
10  f.close()
```

Listing 4.9 – Data Breach Detection Phase – Pass on real password to all the next epochs

The next and final step of the data breach detection phase is to simulate a login for all the users in *last_used_sw*, so that a login with the last sweetword used is simulated. Doing that will essentially propagate the first ever login, which happened during registration, in all the following epochs. This step is what makes our system able to detect a data breach under an account even if its user has logged in just once. In other words, it only takes one

log in with a different sweetword for the system to signal a data breach. As shown in Listing 4.9, for each user in *last_used_sw* we simulate a log in and record it in logins file *L*, to be used in the next data breach detection at the end of the following epoch.

You can find the implementation of the simulation of a login in Listing 4.10, but basically the functionality is built with the same concept as the authentication phase described in the previous section.

```
1   # This function simulates a login for the given user, using the given password
2   def simulate_login(user, password, rs):
3       # get the slots for this user from the checker's database
4       swsC = getSlots(user)
5
6       # the position of the password/slot given, is the position of the real password
7       # in S's database
8       real_position = swsC.index(password)
9
10      # get the sweetwords of this user from the S's database
11      sws = getSws(user)
12
13      # extract the real password
14      password = sws[real_position]
15
16      # remove pass used, shuffle rest, insert pass in place given by Rs
17      sws.remove(password)
18      random.shuffle(sws)
19      sws.insert(rs, password)
20
21      # update row with new shuffled row
22      updateSws(sws, user)
23
24      return real_position
```

Listing 4.10 Data Breach Detection Phase – Simulate Login

4.6. Password Reset

We mentioned in the previous section that an action recorded in the logins file L could either be a “login” or a “password reset”. We expanded in great depth what happens in the case of a “login” event. In this section, we explain how the system reacts in the case of a “password reset” action.

Basically, as also stated in Chapter 3, Subsection 3.2.5, during this phase both the slots in *DBc* and the sweetwords in *DBs* are re-initialized.

First of all, one must first log into their account in order to execute a password reset action. Thus, even if an attacker uses a sweetword to gain access in the system and attempts a password reset, at the end of the epoch the data breach will be detected.

When a “password reset” action occurs, in the server S’s side, the first thing that happens is that we reset *DBs* based on the new password. Thus, see Listing 4.11, we first feed the new password to a *HGT*, that will produce the new 20 sweetwords and update the row of the specific user with these new sweetwords. Then, we simulate a login for that user, so that the new password is now propagated through the rest of the epochs, but in this case, the action recorded in the logins file L will be “pReset” instead of “login”, so that the checker will know to execute the password reset on server C’s side as well.

```
1   # This function resets the row in the Authenticator’s database in case of a
2   # password reset, by updating the row with the new sweetwords retrieved by a HGT
3   def resetDBs(userId, password):
4       sws = HGT(password);
5       updateDBs(userId, sws);
6       authenticate(userId, password);
```

Listing 4.11 - Password Reset *DBs*

In case of a “pReset” event, C will function as shown Listing 4.12. Therefore, the row relevant to the specific user in *DBc* will be reinitialized, so this resets the slots of the user in the same state as when they first register.

```
1   # This function resets the row in the Checker's database in case of a
2   # password reset, by updating the row from slots 'a' through 't'
3   def resetRow(user_id):
4       alphabet_string = string.ascii_lowercase
5       sws = list(alphabet_string)
6       updateSlots(sws, user_id)
```

Listing 4.12 - Password Reset *DBc*

If you recall Chapter 4.3 and more specifically Listing 4.4, you will see that a password reset is essentially a registration phase for the user doing the password reset.

Chapter 5

Evaluation

In this chapter we will be evaluating Lethe, whose implementation we built is described in the previous section. To effectively evaluate the system, we had to do a setup that we will be explaining in depth in this section. First, we had to populate our system with existing data from known data breaches. We have the function use of the paper's "HoneyGen: Generating Honeywords Using Representation Learning" [13] setup that makes a database with honeywords.

5.1 SQLite3

To test Lethe, we use SQLite3, which is a free database engine, to easily create and manage the database of the Server (S), as well as the Checker's (C). We choose SQLite3 because it is a file-based SQL database, so it has the advantage of easier installation and use [31]. Listing 5.1 shows how the database was created. We simply import the SQLite3 package and create the database by creating a connection object. We then create a cursor object to execute queries.

```
# create the database by creating a connection object
connection = sqlite3.connect("honeywords.db")
# create a cursor object in order to execute queries
cursor = connection.cursor()
```

Listing 5.1 - SQLite3

5.2 System Population

5.2.1 Table Creation

As shown in Listing 5.2, we create two tables with the use of the cursor described in the previous section. The creation of the tables is simple and makes use of SQL queries, since we are using SQLite3.The first table, *cheggHoneywords*, is the database of the Server S i.e., *DBs*, whereas the second table, *cheggHoneywordsChecker*, is the database of the Checker C i.e., *DBc*. Both databases have 21 fields, the first one is the User ID; therefore, it is the primary key of the tables. Table *cheggHoneywords* has 20 more "TEXT" fields from which 19 are the generated honeywords for each user and 1 is the real password of the user. Equivalently, *cheggHoneywordsChecker* has 20 more "TEXT" fields where all will be filled with slots ("a" through "t").

```
def initialBootstrap():
    # Drop the tables each time to recreate
    cursor.execute("DROP TABLE cheggHoneywords")
    cursor.execute("DROP TABLE cheggHoneywordsChecker")

    # Create the tables for each server respectively
    cursor.execute("CREATE TABLE cheggHoneywords (ID INT PRIMARY KEY NOT
                    NULL,Honeyword_0 TEXT,... Honeyword_19 TEXT)")
  cursor.execute("CREATE TABLE cheggHoneywordsChecker (ID INT PRIMARY KEY NOT
                  NULL,Honeyword_0 TEXT,... Honeyword_19 TEXT)")
```

Listing 5.2 Initial Bootstrap – Creation of tables *DBs* & *DBc*

5.2.1 Database Filling

To fill the table *cheggHoneywords*, we use existing passwords taken from known data breaches. We chose passwords leaked from chegg.com and generated their HoneyWords using HoneyGen [13]We then populate *cheggHoneywords* with 10000 users and their 19 sweetwords (produced using HoneyGen when fed with the real password) and one real password (from chegg.com known data breach), accordingly, as shown in Listing 5.3. We have these data placed in the file "chegg-com_sorted_preprocessed.txt" in the format

where the real password of each user is the first one in each line followed by the 19 generated sweetwords comma separated.

It is also important to mention that there should be a certain behavior for passwords that include special characters, like apostrophe (‘), since they result in “end of string” and thus cannot be entered in the database. This is the reason why we replace any apostrophes with double apostrophes, to overcome this problem and take any apostrophe as a character rather than an end of string.

```
11  f = open("chegg-com_sorted_preprocessed.txt")
12  lines = f.read().splitlines()
13  for line in lines:
14      sws = line.split()
15      query = "INSERT INTO cheggHoneywords VALUES ( {}, ".format(k)
16      real_passwords.insert(k, sws[0])
17
18      for i in range(0, len(sws)):
19          # help with special characters
20          sws[i] = sws[i].replace("'", "''")
21          query = query + '\'' + sws[i] + '\''
22          if i == len(sws) - 1:
23              query = query + ')'
24          else:
25              query = query + ","
26      cursor.execute(query)
27
28      query = "INSERT INTO cheggHoneywordsChecker VALUES ( {}, ".format(k)
29      c = 'a'
30      for j in range(20):
31          query = query + '\'' + c + '\''
32          if j == 19:
33              query = query + ')'
34          else:
35              query = query + ","
36          c = chr(ord(c) + 1)
37      cursor.execute(query)
38
39  connection.commit()
```

Listing 5.3 - Database Filling

After filling S’s database with 10000 users and their honeywords and real passwords, we fill C’s database with slots from “a” to “t”.

It is shown in Listing 5.3 that we carefully built the insertion query for both the servers and then simply executing it to populate the databases.

It is also very crucial to do a “connection.commit()” to commit the current transaction. Failing to do so, will result in no changes in the database. Equivalently, anything done is not visible and the data are not actually saved in the database.

Furthermore, note that in line 16 of Listing 5.3 we save each user's real password i.e., the first string in the line. We will be using that in a following section.

5.3 Database Shuffle

The next step after the creation of the tables and filling them in, is to do a shuffle of the database, since, as we said, the real password of all users is the first field inserted in each row. Thus, as shown in Listing 5.4, firstly, we fetch all the rows that were previously inserted in "cheggHoneywords", shuffle them using the "shuffle" function of the random built-in module in Python, and then update the row in the database for each user.

```
1   # This function shuffles the Authenticator's database randomly
2   # ---------------------- For evaluation purposes ---------------------- #
3   def shuffleDatabase():
4       rows = cursor.execute("SELECT * FROM cheggHoneywords").fetchall()
5       k = 0
6       for sws in rows:
7           sws = list(sws)
8           sws.remove(k)
9           random.shuffle(sws)
10          sws.insert(0, k)
11
12          for i in range(1, len(sws)):
13              sws[i] = sws[i].replace("'", "''")
14
15          connection.execute("UPDATE cheggHoneywords set "
16                             "Honeyword_0 = \'{}\', ..., Honeyword_19 = \'{}\'"
17                             "WHERE ID = {}".format(sws[1], ..., sws[20], k))
18          k += 1
19          connection.commit()
```

Listing 5.4 - Database Shuffle

5.4 Mass Registration

As we mentioned in the previous chapter, in a real-life application, whenever a user registers into the system, the registration phase occurs. For our purposes, where we populated the system with existing data, we had to do a massive registration phase for all 10000 users we inserted in our database, as described in the previous sections of this Chapter. Recall that the Registration Phase is basically the first login for each user, as that is what happens when a user registers into a system.

As mentioned in 5.2.2, we saved each user's real password during the creation of the tables. We use that, as well as the implementation of the authentication phase described in Chapter 4.4 of this thesis, to simulate each user's first login.

```
# This function does a mass registration phase for each user inserted in the
# Authenticator's database.
# That is, for every login, the real password is used to log in the user to the
# system.
# The login sets off the authentication phase.
# ---------------------- For evaluation purposes ---------------------- #
def massRegistrationPhase():
    # The fist step is to initialize/sychronize the CSRNG of both servers
    trustedBootstrap()

    # Log in each user
    for i in range(len(real_passwords)):
        Authenticator.authenticate(i, real_passwords[i])
```

Listing 5.5 - Mass Registration Phase

As you can see in Listing 5.5, the first step is to initialize the system using the trusted bootstrap (see Listing 4.3). Then, for each user, we use the authentication phase implementation to simulate a login using their real password (see Listing 4.6).

This completes the Registration Phase for all 10000 users currently in our database. Now, every user in the system has logged in once, thus the password used in this first login is the considered real password and will from now on be passed through all epochs. This is the only sweetword that shall be used to login a user and that means that the use of any other sweetword will mark a data breach under the account.

5.5 Login Simulation

After creating and filling the databases for both the Server S and the Checker C and doing the first login with the correct password for each user, now it's time to simulate some logins for each user and check at the end of the epoch if the Checker will detect a Data Breach if it happens. We then calculate the possibility of a successful attack.

5.5.1 Simulation using correct password

To simulate logins, firstly, we took "iterations" random user ids. Then, we use the same file used during the Database filling, to get the real passwords of the users and simply call the authentication phase for each random user using the correct password.

We tested this function simulating 200 correct logins. This experiment resulted in all the logins being successful and no data breach was detected, which is what was expected since all the logins were happening with the real password, the one used to register the user to the system.

```
# This function simulates correct logins "iterations" times, using random user IDs
# and their correct passwords, as retrieved from the file used during filling the DB
# -------------------------- For evaluation purposes -------------------------- #
def simulateCorrectLogins():
    # Find total number of users
    number_of_users = cursor.execute("SELECT COUNT(*) FROM cheggHoneywords")
    for u in number_of_users:
        u = list(u)
        number_of_users = u[0]

# Fill an array of "iterations" size, with random user IDs from 0 to total number of
# users
    user_ids = np.random.randint(number_of_users, size=iterations)

    # Open the file to get the correct passwords of each user
    f = open("chegg-com_sorted_preprocessed.txt", "r")
    lines = f.read().splitlines()
    f.close()
    correct_passes = []
    for user_id in user_ids:
        line = lines[user_id]
        sws = line.split()
        correct_passes.append(sws[0])

    # Simulate a login event for each user, which will each time trigger an
    # authentication phase
    Authenticator.authenticationPhase(user_ids, correct_passes)
```

Listing 5.6 - Simulate logins with real passwords

Hence, we know that our system successfully recognizes a correct login without mistakenly recognizing a data breach and invoking any actions.

5.5.2 Simulation using random sweetword

Similarly, as in the simulation of correct logins, to do a random login we also firstly take "iterations" random user ids. However, in order to choose which random sweetword we will be logging in the user with, we fill an array of "iterations" size with random numbers from 1 to 20. Each number stands for the random position of password for each user. We, then, fetch each user's sweetwords from *DBs* and use the random position to log them in. It could be the real password or not. Statistically, there is a 1/20 possibility that the random position contains the user's real password. Then, in the same manner as before, we call the authentication phase for each random user using the random sweetword.

```
# This function simulates random logins "iterations" times, using random user IDs
# and a random password between the 20 retrieved from Authentication Server S's
# database
# -------------------------- For evaluation purposes -------------------------- #
def simulateRandomLogins():
    # Find total number of users
    number_of_users = cursor.execute("SELECT COUNT(*) FROM cheggHoneywords")
    for u in number_of_users:
        u = list(u)
        number_of_users = u[0]

    # Fill an array of "iterations" size, with random user IDs from 0 to total
    # number of users
    user_ids = np.random.randint(low=0, high=number_of_users, size=iterations)
    # Fill an array of "iterations" size, with random numbers from 1 to 20, which
    # stands for the random position of password of each user
    passes_positions = np.random.randint(low=1, high=20, size=iterations)

    # For each user, retrieve the random password from the authenticator's database
    passes = []
    for i in range(len(user_ids)):
        sws = cursor.execute("SELECT Honeyword_0, ..., Honeyword_19 FROM
                        cheggHoneywords WHERE ID={}".format(user_ids[i])).fetchall()
        sws = list(sws[0])
        passes.append(sws[passes_positions[i]])

    # Simulate a login event for each user, which will each time trigger an
    # authentication phase
    Authenticator.authenticationPhase(user_ids, passes)
```

Listing 5.7 - Simulate logins with random sweetwords

5.6 Results

The function in Listing 5.7 - Simulate logins with random sweetwords is the one that we will be using to conduct our results, since it basically impersonates the actions of an attacker. In the case where an attacker has compromised the database and has the sweetwords, they would be doing random logins with a 5% possibility to choose the correct sweetword. We are hoping to see that Lethe will be able to detect the rest 95% of the attacks.

Also, for the effective evaluation of our implementation, we consider that the end of an epoch i.e., when the Data Breach Detection phase occurs, is after every "*EPOCHS*" number of logins.

5.6.1 Password Reset

To make the matters more realistic, we introduced the possibility of a "password reset" event, thus not all actions are "logins" since an attacker might attempt to also change the

password of an account they have successfully compromised. In fact, we pass some "password reset" actions with a possibility of 20% (see Listing 5.8).

```
1   if rand <= 0.2:
2       # user should have logged in before doing a preset
3       logged = user_ids[0:i - 1]
4       if user_ids[i] in logged:
5           action = "pReset"
```

Listing 5.8 - Introducing possibility of password reset

As we said, a user must login first before doing a password reset. If an attacker has used a sweetword to log in first before doing a password reset, and that sweetword is not the real password, then our system will be able to detect the data breach at the end of an epoch, regardless of the fact if they attempted a password reset or not.

5.6.2 Experiment Results

Experiment 1

We simulated random logins for 2000, 4000, 6000, 8000 and 10000 iterations with an epoch of 1000 logins, and show the results in Figure 5.1 and Listing 5.9.

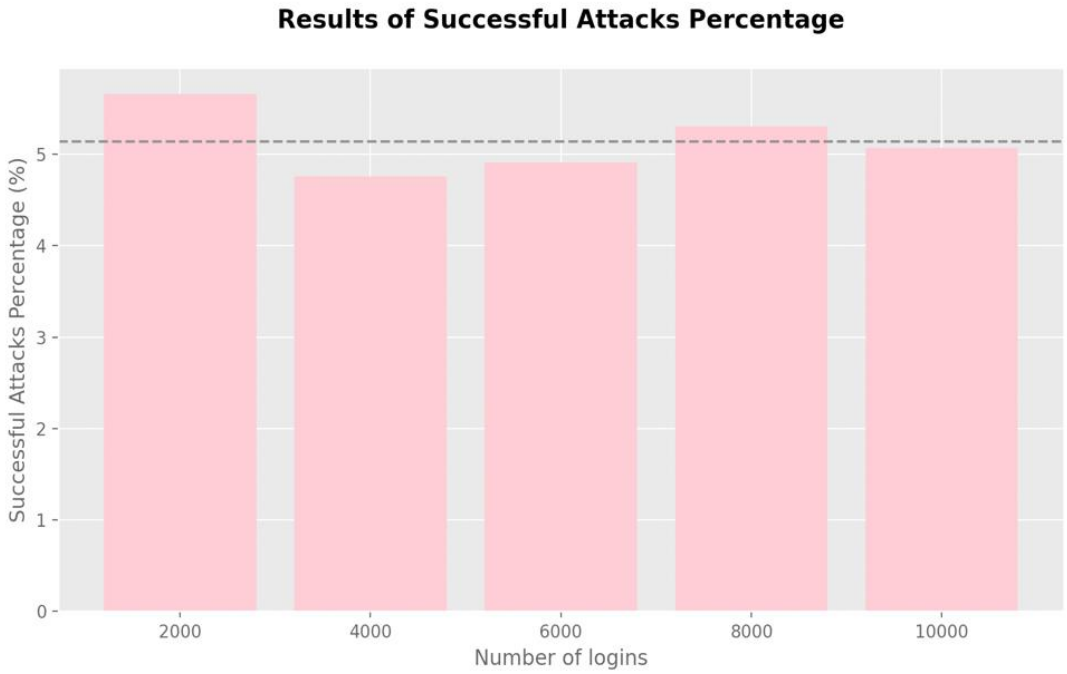

Figure 5.1 - Results of Experiment 1 – 2000 to 10000 logins

As you can see in **Error! Reference source not found.**, for every number of logins, the number of successful attacks at the end of the last epoch was around 5%, which was what we predicted since as we said, the attacker has a 1/20 possibility to guess the real password out of the 20 sweetwords. More specifically, Lethe was able to detect around 95% of all the attacks.

```
1   Start Random logins for 2000 logins:
2   Successful Attacks: 107 / 1890, 5.661375661375661%
3   Detected Attacks: 1783 / 1890, 94.33862433862434%
4   Done with random logins for 2000 logins.
5
6   Start Random logins for 4000 logins:
7   Successful Attacks: 130 / 2732, 4.758418740849195%
8   Detected Attacks: 2602 / 2732, 95.2415812591508%
9   Done with random logins for 4000 logins.
10
11  Start Random logins for 6000 logins:
12  Successful Attacks: 236 / 4810, 4.906444906444907%
13  Detected Attacks: 4574 / 4810, 95.0935550935551%
14  Done with random logins for 6000 logins.
15
16  Start Random logins for 8000 logins:
17  Successful Attacks: 311 / 5866, 5.301738833958405%
18  Detected Attacks: 5555 / 5866, 94.6982611660416%
19  Done with random logins for 8000 logins.
20
21  Start Random logins for 10000 logins:
22  Successful Attacks: 343 / 6762, 5.072463768115942%
23  Detected Attacks: 6419 / 6762, 94.92753623188406%
24  Done with random logins for 10000 logins.
```

Listing 5.9 - Results of Experiment 1 (console) - 2000 to 10000 logins

The specific measurements of our experiment is shown in Listing 5.9. As you can see, the total number of logins that actually occurred is less than what we initiated at first (e.g., instead of 2000, only 1890 actually happened) and that is because when Lethe detected the attacks under some specific accounts, it added those users in the blacklist and therefore did not accept any more logins from them. For all the simulations, we can see that Lethe always detected around 95% of the attacks.

This experiment brings us to the conclusion that Lethe will only be unable to detect an average of around 5.14% of all attacks (grey intermittent line in Figure 5.2), since an attacker might actually guess the real password out of the 20 sweetwords.

Experiment 2

We also consulted a second experiment with even more simulations of logins. As shown

in Figure 5.2, in this experiment we simulated from 2000 to up to 20000 logins.

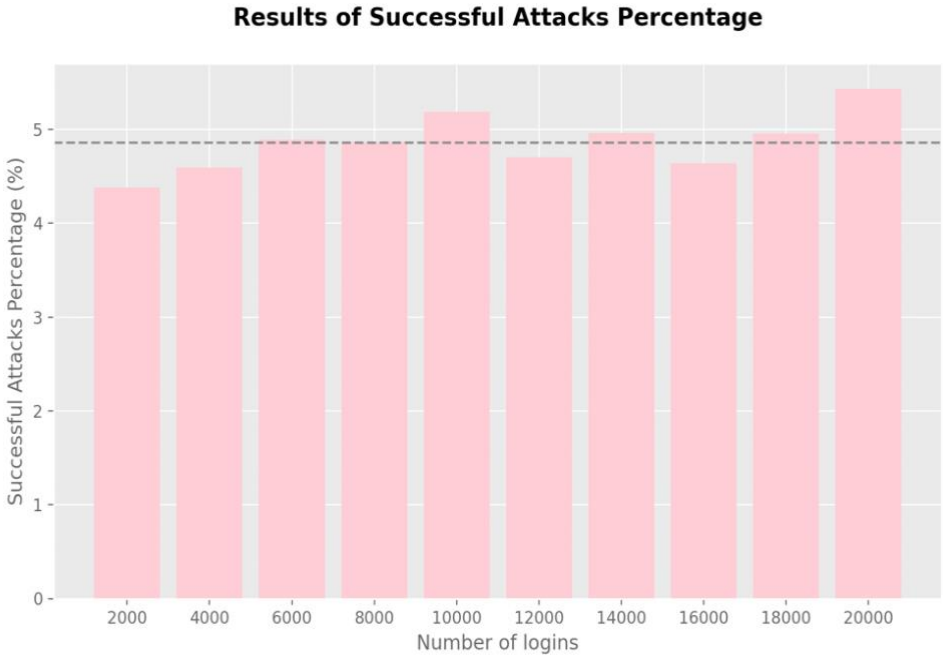

Figure 5.2 - Results of Experiment 2 - 2000 to 20000 logins

Similarly, as in the first experiment, for all the simulations, the number of successful attacks in the end is around 5%. Thus, again, Lethe was able to detect around 95% of all the attacks.

During this experiment, Lethe, actually, did not detect an average of less than 5% of all attacks (see grey intermittent line in Figure 5.2), which is even less than the 1/20 possibility of an attacker picking a random sweetword. This of course can fluctuate since we got the sweetwords randomly whereas a real attacker might choose a sweetword based on a strategy. However, we anticipate on HoneyGen's guarantee that it meets all the expected security requirements and, therefore, does not allow more than the random guessing baseline, which is the optimal solution [32]. HoneyGen is proved to achieve close to the optimal robustness against state-of-the-art honeyword- distinguishing attacks [13].

```
1   Start Random logins for 2000 logins:
2   Successful Attacks: 83 / 1894, 4.382259767687434%
3   Detected Attacks: 1811 / 1894, 95.61774023231257%
4   Done with random logins for 2000 logins.
5   …
6   Start Random logins for 10000 logins:
7   Successful Attacks: 301 / 5802, 5.187866253016201%
8   Detected Attacks: 5501 / 5802, 94.81213374698379%
9   Done with random logins for 10000 logins.
10
11  Start Random logins for 12000 logins:
12  Successful Attacks: 345 / 7338, 4.7015535568274 74%
13  Detected Attacks: 6993 / 7338, 95.29844644317252%
14  Done with random logins for 12000 logins.
15
16  Start Random logins for 14000 logins:
17  Successful Attacks: 389 / 7839, 4.962367648934813%
18  Detected Attacks: 7450 / 7839, 95.03763235106518%
19  Done with random logins for 14000 logins.
20
21  Start Random logins for 16000 logins:
22  Successful Attacks: 400 / 8621, 4.639832966013223%
23  Detected Attacks: 8221 / 8621, 95.36016703398677%
24  Done with random logins for 16000 logins.
25
26  Start Random logins for 18000 logins:
27  Successful Attacks: 451 / 9102, 4.954954954954955%
28  Detected Attacks: 8651 / 9102, 95.04504504504504%
29  Done with random logins for 18000 logins.
30
31  Start Random logins for 20000 logins:
32  Successful Attacks: 461 / 8489, 5.430557191659795%
33  Detected Attacks: 8028 / 8489, 94.56944280834021%
34  Done with random logins for 20000 logins.
```

Listing 5.10 - Results of Experiment 2 (console) - 2000 to 20000 logins

Chapter 6

Related Work

In this chapter we discuss other proposes of data breach detection systems. Firstly, we go through papers that require a trusted component. We then discuss about Amnesia, which is an approach that does not require a secret state.

Various papers exist that propose different data breach detection systems; however, they all require some form of a trusted component. This component is assumed that it will not be compromised, even if the attacker breaches the target.

Erguler [14], introduces an alternative approach that focuses on realistic looking honeywords. Basically, this method suggests a honeyword generation algorithm that chooses the honeywords by exploiting the existing user passwords in the system. While this HGT strives to result in honeywords that are indistinguishable from the correct passwords, it has not yet verified the expected measurements i.e., *flatness* and *success-number graphs* [32]. Also, this method still requires a trusted component i.e., the honeychecker.

I. Mokube and M. Adams [26] suggest the use of Honeypots. Essentially, Honeypots use false accounts that if accessed, set of a data breach alarm. A Honeypot is a network that is closely monitored and uses decoys to take advantage of cyberattacks and collect data about the adversary. Therefore, compromising or attacking a honeypot, increases its success. Fundamentally, Honeypots distract the adversary using false accounts, warn about attacks and in case of an attack and provide an examination regarding the attack. Hence, a Honeypot's worth relies on its interaction with the cybercriminals since it observes the attacker's behavior. If someone interacts with a honeypot, then that interaction is considered an attack. However, implemented Honeypot systems assume that a part of the system under measurement is trusted.

Tripwire, proposed by DeBlasio et al. [11], utilizes Honeypots but does not rely on an aspect of the system under measurement to not be compromised. Tripwire focuses its operation on the problem of password reuse. Particularly, this schema uses decoy emails, indistinguishable from the rest of the user's emails, and use them to register honey accounts in Internet sites. These accounts share the same unique email and password, hence attracts attackers with a password reuse attack opportunity. Thus, if access is gained any of these false account, this instantly signals a data leakage. Still, Tripwire relies on the assumption of a trusted component i.e., the email provider.
Several other methods have been suggested [4], [24], but they also seem to have limitations. For example, Almeshekah's approach [4] requires a machine-depended function.

One approach that does not require a persistent secret state at the target to detect the injection of a honeyword is Amnesia, proposed by Wang et al [33]. Amnesia is a honeywords-based breach detection framework that uses honeywords to detect a target's database breach probabilistically.
Amnesia trains the target by utilizing monitors. Essentially, the target monitors the entry of passwords stolen from other sites. Incorrect passwords entered under the same accounts at monitors act like they have been entered locally at the target. In this paper, a cryptographic protocol i.e., private containment retrieval (PCR), is used. By leveraging this protocol, in the case of an unsuccessful login, a monitor transfers the used password to the target. This happens only in the case where the attempted password is one of the sweetwords associated with the same user account, else the target does not learn.
Amnesia is a framework that uses a probabilistic model verification. In summary, this framework utilizes a marking technique where the most recently used sweetword is marked (with probability 1). The rest of the sweetwords are marked separately with a specific probability (p_{mark}). In the case where the adversary logs in an account using a honeyword (i.e., a sweetword that is not the user chosen password), then the real password of the user gets unmarked (with probability $p_{remark} = 1 - p_{mark)}$. Consequently, the next time the legitimate user logs in, giving a password that is not marked, an alarm sets off since a data breach has been reliably detected.

Chapter 7

Future Work

In this chapter, we first outline the limitations Lethe faces. Then, we briefly discuss some possible improvements of Lethe.

6.1 Lethe's Limitations

Recall that Lethe is initialized by a trusted bootstrap (Chapter 3, Section 3.2.1). During this phase, each server initializes a cryptographically secure random number generator (CSRNG). Both servers use the same sensitive seed to initialize their CSRNGs and the seed is then discarded. Consequently, the two CSRNGs (i.e., *Rs* and *Rc*) are synchronized. If an attacker compromises the CSRNGs and gets the seed before it gets discarded, then they will be able to produce the same random integers and consequently have access to the positions of the used sweetwords. The real password for each user is more likely the sweetword that is mostly used.

As we previously mentioned, Lethe is unaware of the users' real passwords. In fact, if any of the sweetwords under an account is given, Lethe authorizes access to the system. It is only at the end of the day, when the data breach detection occurs, that Checker checks whether two sweetwords were used under the same account and if so, declares a data breach. Therefore, even though Lethe reduces the delay between a data breach and its detection [18], it still permits attackers to operate unnoticed for a limited period. Nevertheless, the target system might still sustain serious harm in this time span.

Additionally, like all the existing approaches at the moment, Lethe cannot guard against attackers that have gained full access on the system's memory and passively monitor the system for the entirety of its operation. Doing that for an extended length of time might disclose some of the users' real passwords, similar to other honeywords-based data breach detection systems [6], [26], [29], [33]. However, note that legitimate users seldom authenticate themselves explicitly [15], [35]. Hence, attackers might not risk getting detected by monitoring a system for a long duration, to only get a few passwords.

6.2 Advancing Lethe

Currently, Lethe does the data breach detection offline and consequently reduces the probability of compromising the checking server C. C is placed in a different isolated environment. However, one could utilize SGX [1], [10], [20], and place the selected code and data in an enclave inside the main authentication server S.

Fundamentally, SGX helps protect data in use via unique application isolation technology. Therefore, C's CSRNG could be protected inside a hardened enclave using Intel SGX. Enclaves are protected memory regions, essentially non-addressable memory pages, reserved from the system's physical RAM and then encrypted. This prevents software attacks even when the app, operating system and BIOS are compromised. Hence, by isolating C's CSRNG we allow Lethe to leverage just a single main server for both authentication and data breach detection. Doing that, consequently, minimizes the overall network communication overhead.

Chapter 8

Conclusion

In this thesis, we provide the first fully functionate prototype of Lethe; a deterministic honeywords-based data breach detection framework that detects a database leakage timely. Lethe keeps a zero persistent secret state and requires no trusted components, other than a trusted bootstrap, in contrast to currently implemented approaches that utilize honeywords. Based on our results, by mimicking a sophisticated adversary's behavior, Lethe ensures the detection of 95% of all data breaches. Hence, only 5% of all data breaches are successful, revealing that the adversary cannot achieve more than the random choosing success rate.

Additionally, Lethe allows attackers to fully compromise the utilized HGT, since it leverages machine learning technologies that automatically ensure the *non-reversibil*ity property; even by giving the same password, it is impossible to reproduce the same honeywords.

Finally, Lethe reduces the delay between a data breach and its detection. If the adversary attempts a login using a different sweetword than the one used by the user during the registration phase, Lethe guarantees the detection of a data breach at the end of the day, when the data breach detection occurs.

Bibliography

[1] "12th Generation Intel CoreTM Processors, Datasheet, Volume 1 of 2," January 2022, [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/655258.

[2] "2018 Credential Spill Report." [Online]. Available: https://www.coursehero.com/file/107589941/ShapeCredentialSpillReport2018pdf/

[3] N. AlFardan et al, “On the Security of RC4 in TLS”, in the Proceedings of the 22nd USENIX Security Symposium, 2013.

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

[5] M. Alsaleh, M. Mannan and P. C. van Oorschot, "Revisiting Defenses against Large-Scale Online Password Guessing Attacks," in IEEE Transactions on Dependable and Secure Computing, vol. 9, no. 1, pp. 128-141, 2012.

[6] 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.

[7] J. Bonneau, C. Herley, P. Oorschot, and F. Stajano, “The quest to replace passwords: A framework for comparative evaluation of web authentication schemes,” in Proceedings of the IEEE S&P, pp. 553–567, 2012.

[8] 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.

[9] K. Chanda, “Password Security: An Analysis of Password Strengths and Vulnerabilities,” published in Mecs, 2016

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

[11] 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.

[12] A. Dionysiou and E.Athanasopoulos, “Lethe: Practical Data Breach Detection with Zero Persistent Secret State” in Proceedings of the 7th IEEE European Symposium on Security and Privacy, 2022.

[13] 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.

[14] 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.

[15] 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.

[16] L. Grigas, “Learning Password Security Jargon: Dictionary Attack,” 2022, [Online]. Available: https://nordpass.com/blog/what-is-a-dictionary-attack/

[17] 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/

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

[19] IBM Security, “Cost of a data breach report 2020,” 2020. [Online]. Available: https://www.ibm.com/security/digital-assets/cost-data-breach-report/

[20] Intel® Software Guard Extensions, 2022, [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html

[21] 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.

[22] 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.

[23] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, “Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms,” in Proceedings of the 33rd IEEE Symposium on Security and Privacy, 2012.

[24] 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.

[25] C. Liu, “Alibaba Victim of Huge Data Leak as China Tightens Security,” Bloomberg, 2021. [Online]. Available: https://www.bloomberg.com/news/articles/2021-06-16/alibaba-victim-of-huge-data-leak-as-china-tightens-security

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

[27] B. Pinkas and T. Sander, “Securing passwords against dictionary attacks”, in Proceedings of the the 9th ACM conference on Computer and communications security, 2002, pp. 161-170.

[28] Shape Security, “2018 credential spill report,” 2018. [Online]. Available: https://info.shapesecurity.com/rs/935-ZAM-778/images/Shape_Credential_Spill_Report_2018.pdf

[29] 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.

[30] X. Shu, K. Tian, A. Ciambrone and D.D. Yao, “Breaking the Target: An Analysis of Target Data Breach and Lessons Learned,” arXiv, 2017

[31] SQLite Home Page, 2022. [Online]. Available: https://www.sqlite.org/index.html

[32] 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.

[33] 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.

[34] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, “Targeted online password guessing: An underestimated threat,” in Proceedings of the ACM CCS pp. 1242–1254, 2016.

[35] 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.