Examensarbete

Distributed Cipher Chaining for Increased Security in Password Storage

Examensarbete utfört i Datavetenskap
vid Tekniska högskolan vid Linköpings universitet
av

David Odelberg and Rasmus Holm

LiTH-ISY-EX--14/4764--SE

Linköping 2014
Distributed Cipher Chaining for Increased Security in Password Storage

Examensarbete utfört i Datavetenskap
vid Tekniska högskolan vid Linköpings universitet
av
David Odelberg and Rasmus Holm

LiTH-ISY-EX--14/4764--SE

Handledare: Jonathan Fors
ISY, Linköpings universitet
Hannis Albinsson
Spotify AB

Examinator: Jan-Åke Larsson
ISY, Linköpings universitet

Linköping, 5 juni 2014
Distribuerade cipherkedjor för ökad säkerhet i lösenordshantering

David Odelberg and Rasmus Holm

As more services move on to the web and more people use the cloud for storage of important information, it is important that providers of such services can guarantee that information is kept safe. The most common way of protecting that data is to make it impossible to access without being authenticated as the user owning the data. The most common way for a user to authenticate and thereby becoming authorized to access the data, or service, is by making use of a password.

The one trying to safeguard that password must make sure that it is not easy to come by for someone trying to attack the system. The most common way to store a password is by first running that password through a one way function, known as a hash function, that obfuscates it into something that does not at all look related to the password itself. Whenever a user tries to authenticate, they type in their password and it goes through the same function and the results are compared. While this model makes sure that the password is not stored in plain text it contains no way of taking action in case the database of hashed passwords is leaked.

Knowing that it is nearly impossible to be fully protected from malevolent users, the ones trying to safeguard information always need to try to make sure that it is difficult to extract information about users’ passwords. Since the 70s the password storage has to a large extent looked the same. What is researched and implemented in this thesis is a different way of handling passwords, where the main focus is on making sure there are countermeasures in case the database leaks. The model described and implemented consist of software that make use of the current best practices, with the addition of encrypting the passwords with a symmetric cipher. This is all done in a distributed way to move towards a paradigm where a service provider does not need to rely on one point of security.

The end result of this work is a working proof-of-concept software that runs in a distributed manner to derive users’ passwords to an obfuscated form. The system is at least as secure as best current practice for storing users passwords but introduces the notion of countermeasures once information has found its way into an adversary’s hands.

Cryptography, Password storage, AES, KDF, Scrypt
Abstract

As more services move on to the web and more people use the cloud for storage of important information, it is important that providers of such services can guarantee that information is kept safe. The most common way of protecting that data is to make it impossible to access without being authenticated as the user owning the data. The most common way for a user to authenticate and thereby becoming authorized to access the data, or service, is by making use of a password.

The one trying to safeguard that password must make sure that it is not easy to come by for someone trying to attack the system. The most common way to store a password is by first running that password through a one way function, known as a hash function, that obfuscates it into something that does not at all look related to the password itself. Whenever a user tries to authenticate, they type in their password and it goes through the same function and the results are compared. While this model makes sure that the password is not stored in plain text it contains no way of taking action in case the database of hashed passwords is leaked.

Knowing that it is nearly impossible to be fully protected from malevolent users, the ones trying to safeguard information always need to try to make sure that it is difficult to extract information about users’ passwords. Since the 70s the password storage has to a large extent looked the same. What is researched and implemented in this thesis is a different way of handling passwords, where the main focus is on making sure there are countermeasures in case the database leaks. The model described and implemented consist of software that make use of the current best practices, with the addition of encrypting the passwords with a symmetric cipher. This is all done in a distributed way to move towards a paradigm where a service provider does not need to rely on one point of security.

The end result of this work is a working proof-of-concept software that runs in a distributed manner to derive users’ passwords to an obfuscated form. The system is at least as secure as best current practice for storing users passwords but introduces the notion of countermeasures once information has found its way into an adversary’s hands.
Sammanfattning


Eftersom vi vet att det i princip är omöjligt att vara helt skyddad från användare som vill attackera systemet så måste den som skyddar lösenorden se till att det är svårt att få ut någon information från en lösenordshash. Sättet det görs på har sett mer eller mindre liknat ut sedan sjuttiotalet. Vad som undersöks i denna modell är ett annat sätt att spara lösenord, där huvudfokus ligger på att se till att det finns motåtgärder att ta till om databasen med lösenordshashar läcker. Modellen som beskrivs och implementeras består av mjukvara som använder sig av de standardmodeller som finns, med tillägget att resultatet krypteras med ett symmetriskt chiffer. All detta är gjort på ett distribuerat sätt vilket leder till att den som tillhandahåller tjänsten inte behöver lita på att en enskild server är säker. Samtidigt ska det vara omöjligt att få ut klartext lösenordet, även för den som kör systemet.

De resultat som nåtts är ett fungerande prototypsystem som på ett distribuerat sätt klarar av att skapa krypterade versioner av lösenord. Systemet är minst lika säkert som de standardmodeller som finns men introducerar också möjligheten att ta till motåtgärder om information på något sätt hamnat i fel händer.
Acknowledgments

We would like to thank Spotify for letting us do our thesis work at their offices. A special thanks to Hannis Albinsson, our supervisor at the company and of course thanks to the team we have been sitting with. They have all been giving useful input when needed and have generally kept our spirits up.

We would also like to thank our examiner Jan-Åke Larsson for taking an interest in the project and our academic supervisor Jonathan Fors for making sure we stayed on track.

Linköping, Juni 2014
David Odelberg och Rasmus Holm
Contents

1 Introduction 1
  1.1 Purpose .................................................. 1
  1.2 Issue .................................................. 1
  1.3 Prerequisites ........................................... 1

2 Background 3
  2.1 Password storage and Authentication through history .......... 3
  2.2 Current best practice for persisting passwords .................. 5
  2.3 Password leaks ........................................... 6
  2.4 Problems with today’s model ................................ 7

3 Theoretical Background 9
  3.1 Cryptographic hash functions ................................... 9
    3.1.1 Key derivation functions ................................ 11
  3.2 Symmetric cryptography ..................................... 12
    3.2.1 Block ciphers ....................................... 12
    3.2.2 Security ............................................ 14
  3.3 Hardware security modules ..................................... 14
  3.4 Secret Sharing ............................................ 15
    3.4.1 Schemas .............................................. 15
    3.4.2 Verifiable secret sharing .............................. 17
  3.5 Secure multiparty computation ................................. 18

4 Proposal 19
  4.1 Method .................................................. 19
  4.2 Model ................................................... 20
    4.2.1 Overview ............................................ 20
    4.2.2 Chaining AES ....................................... 21
  4.3 Alternative Models ........................................ 22
    4.3.1 Replacing HMACs with Hash and Encrypt .................. 22
    4.3.2 Secret sharing ....................................... 23
    4.3.3 Secure multiparty computation ......................... 25
    4.3.4 Calculations on client ............................... 25
4.4 Realization ......................................................... 26
  4.4.1 Concept implementation ............................... 26
  4.4.2 Our implementation ................................. 28
  4.4.3 Frameworks ........................................ 31

5 Analysis 33
  5.1 Security concerns ........................................ 33
  5.2 Complexity ................................................... 34
  5.3 Distribution and upgradability ...................... 35
  5.4 Side channel attacks ..................................... 35
  5.5 Oracle attack ............................................... 36
  5.6 Future work ............................................... 37
    5.6.1 Distribution ........................................... 37
    5.6.2 Rate limiting and proof of work ............... 37
    5.6.3 TLS .................................................. 39
    5.6.4 Multi language clients ........................... 39

6 Conclusion 41

Bibliography 43
1.1 Purpose

The purpose of this thesis work is to research and implement a distributed model for hashing, encrypting, and storing passwords for users registered on a multi-user service, such as an e-mail provider or a music streaming service. Areas looked at are what difference a distributed model bring to the table, in the sense of provable security but also what implications such a model might have in a real world scenario.

1.2 Issue

- What security concerns does a distributed password hash model raise and what types of attacks are feasible?
- What is the security complexity difference between our cryptographic construct and other commonly used ones such as Scrypt, PBKDF2SHA256 and HMACSHA256
- Is distributed hashing and an upgradeable schema worth considering?

1.3 Prerequisites

This thesis report assumes some prior knowledge in the areas cryptography, it-security, and programming. Some cryptographic constructs used in the developed system are explained more in depth while others are assumed to be knowledge that the reader already has.
In this chapter some background will be presented regarding how passwords are currently handled, how they have been handled through history, and some of the problems with these paradigms.

2.1 Password storage and Authentication through history

Today there exists a multitude of different ways for authenticating a user on a system. One of the more prominent and widely used ways, aside from just passwords, is two-factor authentication where the user proves ownership of some sort of artifact in the authentication process as well as supplying his or her password. Examples of this are Yubikey and RSA SecureID which both are physical artifacts required in the authentication process. Another is the use of biometrics, such as the fingerprint scanner on the iPhone 5S or on laptops. However, the use of regular password authentication remain dominant in order to authenticate a user to a system, especially on the web.

The fundamental reason for authentication to a system is that multiple users shall have access to it with different privileges while preventing every one else from accessing it. In the early days of computers there were only big main frames with a very limited amount of access points, often one or more fixed terminals in proximity to the main frame itself. In this setting the physical security around the terminals and main frame usually gave the authentication needed to use them.

Unix was one of the first multi-user operating systems and Unix itself and derivatives of it, such as Linux, are still very much in use today. The way a user was
authenticated in the beginning was that the password and the user name of the
user was stored in plain text in a file on the file system, e.g. /etc/passwd, and they
were simply compared upon a authentication attempt. The access to this file was
then heavily restricted in order to protect the passwords of the systems users.
This however proved quite inefficient, hard to protect and was vulnerable to all
sorts of attack, such as exploiting race-conditions and timing attacks.

The solution to the problem came in the mid 70s. The idea was to construct
a hash function, a one way function that takes an input and deterministically
generates an output, where the input could not be constructed from the output
[9]. Upon user registration the input to the hash function would be the user pro-
vided password and the output of the function would be stored on the file system
coupled with the users name for later comparison. When the user later tried to
authenticate he or she would provide the log in prompt with a user name and
password, just as before. The computer would then compute the hash function
value of the password provided. The resulting value would be compared with
the hash value persisted on disk associated with that user. If they were equal the
user would be authenticated. This removes much of the risk of storing passwords
since constructing the passwords from the hash functions output is very difficult.

This paradigm however comes with its own set of problems, one being that two
users with the same password would have the same hashed value or in more
general terms; every password maps to exactly one hash value. This allows an
adversary to construct large tables, known as rainbow tables, of precomputed
hash values from likely passwords, e.g. 2-6 letter combinations. It is then a sim-
ple task for an adversary with access to the hashed password values to compare
them with the rainbow table in order to figure out a users password.

The rainbow table problem was solved in the late 70s by generating a string of
random data, called salt, that is concatenated to the password before passing it
to the hash function. The salt is then stored along side the hash function value
and the user name. By using this construction with enough salt entropy the use
of rainbow table is no longer a feasible way to compromise user passwords. A
paper detailing this construction was published by Bell labs in -78 relating to
their work on Unix [16].

Attacks on this type of construction is now fairly limited with a sufficiently strong
cryptographic hash function in place, meaning that no further information can
be gained or derived from knowing the hashed value and the salt. What an adver-
sary can do in this situation is to target single users one at a time and try to brute
force the password by continually guessing a password, concatenating it with the
salt, hashing it and comparing it with the stored hash value.

Since the 70s there haven't been any new ground breaking constructs in how to
persist passwords to storage and protect them in case of a data leak. The idea
has been the same, to salt and hash the passwords before storing them. What
have changed is the hash functions being used and how they are being used. To-
day there are Key Derivation Functions, such as Scrypt, that both have adjustable
memory footprint and cpu-time in order to derive the key, the hash value, from the password. This concept of sufficiently slow hash algorithms to make an adversary’s brute force or dictionary attack more difficult is detailed in the paper from Bell Labs. Scrypt has simply adopted this to also deal with increased parallelization of GPUs by enforcing a larger memory requirements.

Even though Unix and its derivatives seem to have adopted this construct early on, many others have not; even though it is still considered best practice. Examples include many Microsoft products which are riddled with strange cryptographic constructs that actively weakens the users password or encrypt the plain text password with one or a few master keys. Others are online service providers that persist passwords in everything from plain text to poor in house cryptographic construction, often proven to be less secure than best practice.

## 2.2 Current best practice for persisting passwords

The current best practice involves using an approved KDF like Scrypt, PBKDF2, or a HMAC with a secure hash function and system wide key not stored together with the resulting MACs 3.1. This should be used together with a randomly generated salt, stored together with the alias of the user [22]. By using widely approved constructs no one storing user credentials has to reinvent the wheel in order to keep their user’s credentials safe. The idea is that constructs used to create password derivatives are the same for everyone and that they are not relying on a secret implementation in order to be secure. The reason for using approved functions, as opposed to inventing your own, is based on the fact that it is a really difficult thing to do. Not having them reviewed by knowledgeable people likely results in security holes being present.

The idea of using salts to prevent the use of rainbow tables has as mentioned been around since the 70s. In order for the salt to be considered good it needs to fulfill some properties. The salt should be randomly generated for each user and the purpose is to increase the entropy of what goes into to the hash function. The salt has to be sufficiently long to be able to prevent the possibility to make use of rainbow tables, by using a salt that is 32 bits long the number of results that can come out from the user having “12345678” as a password is $2^{32}$ instead of just one. If the salt is only one bit long, only $2^1$ values for each common password needs to be calculated by the adversary, which is a lot more feasible. Since the salt can be considered to be known to someone attacking the system, this doesn’t protect an individual password in case of a bruteforce attack, nor is it the intention of the salt.

It is also recommended to keep the password space as large as possible and not restrict the signs allowed to be used in the password beyond what is reasonable e.g. some special characters that can be used to attack the file system, but preferably no characters should be removed from the password space. The special characters should rather be escaped and encoded properly to lower the risk of injection attacks. In order to keep the password space large, the upper bound for the pass-
word length should be as large as storage space permits [22]. Even though the ideal situation is that the user has an easy to remember password, thus never forgetting it, it’s not recommended to keep the user from having a password with high entropy.

In case of compromise of the database measures need to be taken, today that involves asking the user to update their passwords on first login after detection. It can also involve preventing altering of sensitive information such as ways of recovery of password until a new password is in place. Due to the nature of one way functions, there is no other way to deal with a password leak. Changing the salt would change what is stored in the database but the end users would still be vulnerable, until the password is changed as can be seen in 2.3.

2.3 Password leaks

During the last couple of years the media has frequently reported that large companies have leaked their user information databases often including passwords in plain text or a derivative of said password, such as a hash. Example of such database leaks from companies are LinkedIn who leaked an estimated six million user names with their SHA-1 derived passwords in jun of 2012. In October 2013 Adobe leaked a staggering 130 million user records and passwords were both in plain text and 3DES encrypted. The notable thing about the Adobe leak is that even though the company at the time of the attack complied with the current best practice the adversaries targeted a backup of the old system, in which an inferior construct was used to protect the users passwords. There are many more of this type of leaks both from large and small companies online. During 2013 alone there were well over 2000 different leaks, totaling at some 238 million user credentials [14]. This is an astronomical amount and the full effect of this is probably still remaining to be seen.

It is worth noting that discovery of leaked user information is not always discovered at the time of the leak itself but rather when the leaked data surfaces on some forum, website, or on darknets. This suggests that many leaks may very well go undiscovered to the public and the company subjected to the leak.

Password leaks and companies’ information leaking can happen in many different ways. The target for adversaries is often company websites, this due to the nature of the ever more complex web applications. During recent years more functionality and logic has moved to the web, often in order to make them more accessible to the users. Some businesses, such as Google, Facebook and LinkedIn, only reside on the web where complex applications are created for their users. One of the problems with this paradigm is that users are allowed to execute code on these companies’ servers with no or very basic authentication. This shall not be viewed as reason not to use web based application but one needs to be aware of the problems that they come with. The code that do execute on the servers per the users request is usually well defined and the execution of arbitrary code is not allowed. There are however cases when the distinction between code and
2.4 Problems with today’s model

Data is hard to make. One of the more common attacks is called SQL-injection, where an adversary might gain access to execute arbitrary SQL code on the target's database. This happens when the code executing on the servers is not, during runtime, able to distinguish between user supplied data and the predefined code it is supposed to execute. Malicious data may be crafted by an adversary that will be interpreted as code when the SQL query is executed by the database. Adversaries tend to go for the low hanging fruit and with the explosion of web applications the last couple of years, that is what SQL injections have become. It seems that protecting oneself against this type of attack is very hard due to the underlying constructs of SQL and the binds to other programming languages.

The effect of a password leak can vary a lot depending on the data leaked. Using the best practice for storing the passwords offers some protection for the user, but this is by no means complete. Once the data is leaked it is out there forever and the only real option for a user today is to change their password. While this might seem trivial, it is not. Many people are creatures of habit and tend to have the same passwords across multiple sites[8]. A leaked password might therefore very well grant an adversary access to much more than the service the password came from. Many users therefore have to change the password on every service where that particular password is in use, and in the light of that it's no longer a trivial task. This means that whenever a user reuses a password they trust the service to not only safeguard access to that service, but also to every other service where that password is in use.

2.4 Problems with today’s model

As previously mentioned, there are some problems with how password handling is done in the industry today. They stem partly from businesses not adhering to best practices. Not every service provider salts their passwords properly e.g. using too short salts, and some service providers don't use a hashing algorithm at all. Since it's very difficult to protect yourself fully from SQL-injections this needs to be taken into consideration when deciding on how to handle passwords. Since a leak is forever a user's account will not be secure until the password has been changed, it is therefore common to involve the user once a leak has been detected. This is something that's not really desired since is has implications, for starters the service provider has to go public with the fact that their database wasn't protected in order to make sure users know to change their passwords. This can result in some bad publicity. But also the user might not receive information about the leak and might therefore not update the password before it's too late.

There are still a lot of outdated systems using old algorithms that could be considered broken. MD5 is not uncommon in web-applications and it is an algorithm that is susceptible to brute forcing in offline mode using GPUs [1]. The increased computing power, and lowered costs of GPUs has made it easier for private enthusiasts to build their own rigs and try to crack passwords from a leaked database.
without spending an unreasonable amount of money. Since the database is valid until passwords have been changed situations can emerge where there is enough time to successfully launch an offline attack and use the information obtained for whatever malevolent reason the attacker has. This puts a lot of responsibility on the end user to have a secure password since it becomes significantly harder to brute-force a longer password that’s been chosen to be hard to guess.

Since the standard is based on using one way functions there’s no way for the service provider to make a leaked database useless. Services therefore have to rely on the users to update their passwords or generate new, random, passwords for all users. In the first case it’s quite likely that the user will pick another unsafe password or possibly just use the same password they had before, if the service allows it. In the second case the user is quite likely to either switch back to the password they had before, or use some related password [8].

If all service provider started using good hashing algorithms and properly salted passwords there would still be the problem with the ever increasing computational power being released. Algorithms that are not susceptible to brute-forcing today might be possible to attack in a few years.
3 Theoretical Background

This chapter will give a short overview of important cryptographic concepts needed to understand our proposal, alternative models and more.

3.1 Cryptographic hash functions

A hash function is, simply put, a function, \( \text{hash} \), that takes a message \( m \), of arbitrary length, as input and produces a fixed-sized output \( h \) such that \( h = \text{hash}(m) \). Hash functions is a fairly common occurrence in software development and has many uses, often not related to cryptography or security. One common use case is hash-maps where a key is mapped to a value stored in e.g. an array for later retrieval. The key itself might be a string or something else that is not mappable to an index of an array. A hash function is then used, \( h = \text{hash(key)} \), where \( h \) then maps to an index in the array, as seen in Figure 3.1.

For this type of use the hash function itself must posses certain properties in order to work efficiently. Two of them are the following.

- A hash function should be \textit{deterministic}, meaning that given the same input the function shall always produce the same output.

- A hash function should posses \textit{uniformity}, meaning that every possible hashed value, \( h \), shall be generated

Figure 3.1: A simple example of a four bit output hash function
by the function with the same probability. This property provide a sort of collision resistance.

Hash functions use does however extend far beyond what is describes above and for their use in cryptographic situation we must extend the definition in order to create a Cryptographic hash function \[6\][15].

- A cryptographic hash function shall be pre-image resistant, meaning that given the hashed value, \( h \), it shall be hard to find a pre-image, \( m \), such that \( h = \text{hash}(m) \).

- A cryptographic hash function shall be second pre-image resistant, meaning that given the input \( m_1 \) it shall be hard to find the input \( m_2 \) such that \( \text{hash}(m_1) = \text{hash}(m_2) \).

- A cryptographic hash function shall be collision resistant, meaning that it shall be hard to find the input \( m_1 \) and \( m_2 \) such that \( \text{hash}(m_1) = \text{hash}(m_2) \)

An interesting note on these extra criteria for cryptographic hash functions is that no current known function, provably, posses them nor does there exist any proof that a function possessing these properties even exists. All current cryptographic valid hash function only, to the best of our collective knowledge, seem to possess them. Some commonly used cryptographic hash functions are SHA2 and SHA3. There are quite a few deprecated algorithms, such as MD5 and SHA1, that no longer are considered to fullfill the three properties of a cryptographic hash function[20]. They are however still found in many legacy systems and new, poorly implemented systems or protocols.

A cryptographic hash function that has the properties listed above is useful for many things and is from here on referred to only as a hash function. Example use cases are integrity checks and as a part of digital signatures. Given a large message \( m \), for which integrity shall be protected, a user may pass it through the hash function generating the digest \( h \). If just one bit is changed in the original message the resulting digest will be be completely different, thus enabling another party, knowing the digest, to verify that the message has not changed. An adversary may however change the message and rehash it, effectively making the construct useless if both the message and hash are passed through the same medium with a man-in-the-middle. Therefore this is often combined with an asymmetric cipher that signs the hash, which then can be verified, using the public key of the signer, by any party. The reason for just signing the hash and not the entire message, which would be preferable, is that asymmetric ciphers are much more computational expensive to use and would generate a signature as big as the message itself[17].

A similar way of ensuring a messages integrity is to use Message-Authentication-Codes, MAC, in which a key, shared between the parties, is used to calculate a code for a specific message which ensures the integrity of the message. Hash functions can be used for this type of symmetric signing and verification of messages and is called HMAC. A hmac is constructed from a hash function as the following
hmac(key, m) = hash((key⊕opad)||hash(key⊕ipad)||m)) where opad = 0x5c5c...5c and ipad = 0x3636...36 and each one block long for the corresponding hash function being used[13].

Another important use of hash functions is the persisting of passwords on different systems for later use in an authentication process. In fact it is nothing new and was first suggested and developed, to more or less today’s present form, in the late 60s and 70s [23] [16] [9]. The basic construct looks as follows, \( h = hash(salt||password_1) \) where salt is random data, unique for each user, stored in plain text along with the resulting digest \( h \) and a user identifier. On an authentication attempt by a user, he or she provides a user identifier and the password, \( password_2 \). The digest \( h \) and salt, \( salt \), is then retrieved from storage, \( h' = hash(salt||password_2) \) is then calculated. If \( h = h' \) the user is considered authenticated. This construct ensures that no passwords in plain text can be stolen from a system, the salt adds entropy and ensures that users with the same password do not end up with the same digest \( h \) persisted in storage which protects against the use of a rainbow table in an attack. An adversary would have to resort to brute force every users password separately.

Today the best practice looks a little different. It is suggested to use a Key-derivation-function, see 3.1.1, or a HMAC [22]. The HMAC would be used with a system wide secret key, \( key \), and the code, \( h \), persisted to storage would in our example be \( h = hmac(key, salt||password_1) \). The reason for the move from a regular hash function, that was suggested in the 70s, to HMACs or Key-Derivation-Functions, KDF, is the increased speed and parallelization of computation. Hash functions are designed to be very fast and collision free. This makes it possible, today, to brute-force or deploy a dictionary-attack on bad passwords within a reasonable time frame.

### 3.1.1 Key derivation functions

A key derivation function is more or less a slow hash function with some extra feathers and the concept was first introduced as crypt(3) in Version Six Unix from -78. The notation used, with some exceptions, looks like the following, \( dk = kdf(key, salt, cost, dkLen) \). The derived key, \( dk \), is what is emitted to storage and equivalent to the digest, \( h \), of a hash function. The key, \( key \), corresponds to a password in the case of using it for password authentication. The cost, \( cost \), is some measure of how much computational effort it will take to derive the key and the length of the derived key is determined by \( dkLen \). While these variables are common occurrences in KDFs they are not set in stone. Some KDFs such as bcrypt omits \( dkLen \) and Scrypt’s \( cost \) is split up in different variables for cpu cost and memory cost[12][19].

One of the more commonly used KDF is PBKDF2 which takes a pseudo-random function, most often an HMAC, which is seeded with a password and salt and then repeated an arbitrary amount of times with the result from the previews and the password as seed. This allows for an implementer to effectively decide on how long it should take to compute the derived key. This concept is especially
effective in a password based authentication schema where the system only has to do this once per authentication while an adversary trying to brute-force the password must do this millions of times. This makes an successful attack much less feasible if a single derivation takes a few hundred milliseconds.

3.2 Symmetric cryptography

A symmetric cipher is a cipher in which the keys for encryption and decryption are the same or in some manner related. Symmetric ciphers exist in different forms, both as block ciphers and stream ciphers. In a stream cipher each bit translation from the clear text to its encrypted form is done individually. While in a block cipher the clear text is divided into blocks of some fixed size, e.g. 64 bits, that is encrypted in one iteration of the encryption process. The data is encrypted using a secret key that shall not be known to anyone but the parties involved in the cryptographic exchange. This due to that anyone with the key is able to decrypt the encrypted messages of interest. As opposed to an asymmetric cryptographic function where there's two different keys, one for encryption and one for decryption. In the asymmetric case the key used for encryption is called the public key since it is only used to encrypt messages and therefore can be public. The private key however can only be known to the one who is the intended recipient of the messages.[17]

3.2.1 Block ciphers

A block ciphers used by itself, in general, with a the same input block and key will always produce the same output block. An adversary looking at the cipher text will notice if some things are sent more often than others or if some general information can be deduced regarding data structures. This problem is addressed by using different modes of operations when running block ciphers. The simplest mode of operation is known as ECB, electronic code book, and it will always give the same output for the same input[17]. An example of how this works can be seen in Figure 3.2a.

![Figure 3.2: Block Cipher Encryption Modes](image)

ECB is not recommended when security is a concern since it might be revealing more information than what is intended, such as preservation of data structure. A
Symmetric cryptography

3.2 Symmetric cryptography

popular mode of operation used to overcome the problem with ECB is called CBC, see Figure 3.2b, and it stands for cipher block chaining. In this mode the first block in the plain text is XOR:ed with an Initialization Vector, IV. If the IV is a random sequence, which it should be, this scrambles the original plain text. Then for each block encrypted it’s XOR:ed with the output from the previous block. The end result is that structures from the plain text are not preserved and the end result is better protected. There are several other modes of operation that can be used for block ciphers to make the construction more robust and have cipher text reveal less information about the original structure. They usually depend on using a random IV and some form of feedback from the last block processed. It can be done by mutating a IV between each block, using the encrypted or sometimes the plain text to mutate the result or the input to the next block[18].

AES AES stands for Advanced Encryption Standard and the algorithm beneath this standard is called Rijndael which is related to the inventors’ names, two Belgian cryptographers. The AES algorithm is a fast algorithm that has been the de facto standard for federal government in the U.S since 2002. AES implementation of Rijndael has a block size of 128-bits and can be used with keys of length 128, 192, or 256 bits[11]. The Rijndael algorithm, by itself, has however support for block sizes up to 256 bits and doesn’t have an upper limit to the key size[5]. The algorithm is based on a substitution-permutation network. AES is considered to be a secure algorithm and no feasible attacks on the construct has been discovered. There are some known attacks but they all take far to much time to be considered practical in any foreseeable future.

Twofish Twofish is a block cipher based on the older cipher Blowfish. It was developed by Bruce Schneier among others. The Twofish algorithm also uses a block size of 128 bits and support key sizes of 128, 192 or 256 bits. It does however have a different structure than the one found in Rijndael cipher. The structure of the Twofish algorithm is a so called feistel network and a simple example of it can be seen in Figure 3.3. Twofish was one of the finalist for the AES competition where the Rijndael algorithm was chosen. This algorithm is however not as fast as AES and this is one of the reasons it was not selected at the AES competition.

Authentication Codes When a message has been encrypted it’s desirable to be able to verify that no part of the message has been altered before encryption. It’s therefore possible to make use of authentication codes proving the authenticity of a delivered message. As opposed to
encrypting a message, where every encrypted block is sent to the receiver, generating a MAC for an encrypted message is often done in a so called CBC-MAC mode. In the CBC-MAC mode no IV is used and only the last encrypted block is sent, called a tag. In order for the MAC to be secure it should not be encrypted with the same key as the original message was encrypted with. An example of this can be seen in Figure 3.4.

![Figure 3.4: CBC-MAC mode](image)

### 3.2.2 Security

When looking at attacks possible to perform on a symmetric cryptographic function what is meant is attacks that have a better time complexity than a brute-force attack. For the widely used ciphers mentioned, no attacks have been found with a feasible time complexity. Some theoretical attacks have been found targeting the AES-algorithm, they are considered theoretical since they have a far too large time and data complexity to be considered possible to actually carry out. Using related key attacks some example attacks on AES [2] give a time complexity of $2^{99.5}$ for AES-256 and $2^{176}$ for AES-192, both clearly not possible to make use of in a real world scenario. These attacks are theoretical as of today, but it’s possible that better attacks will emerge over time or that some derived version of these attacks become plausible in the future. Therefore it can make sense to encrypt using multiple cryptographic constructs e.g. first encrypting with AES and then with Twofish[17]. Whenever considering this it is important to remember that the keys for the different ciphers should not be the same, nor should they be related in any way. If the keys are the same it is easy to see that the security has not been enhanced in any way. Encrypting with multiple algorithms might be a good idea for information that needs to be persisted for a very long time in a very secure manner. In almost every use case using a single cipher will likely provides sufficient security.

### 3.3 Hardware security modules

A hardware security module, HSM, is piece of hardware, often a plug-in to existing hardware such as a PCI-card or a network attached device, that performs cryptographic operations using on secure keys. Secret keys for symmetric and asymmetric ciphers have to be stored somewhere and the problem with off the
self computers is that it is hard to store such keys and securely and have them interact with the rest of the world. If an adversary gains full access to a server where keys are used by a software, there is very little that can be done to protect them from him or her. If this is part of your threat model the solution is to use an HSM as an auxiliary service. The HSM stores the key and in turn promises to perform specific cryptographic operations with that key on behalf of the server\cite{10}. They are also used to take load off other systems in symmetric and asymmetric cryptography due to their speed in performing such tasks with its special purpose hardware.

Using this paradigm, no keys are ever accessible to the system and the threat of them being exposed is therefore minute. HSMs usually deploy all kinds of measures, in order to protect its keys, such as having interfaces that only allows for cryptographic querying, tamper detection that deletes all keys if tripped, obfuscated memory storage to protect against reading it with electron microscopy.

Due to the high cost of HSMs they are rarely used in generic deployed systems but are a common occurrence in the public-key-infrastructure at certificate authorities and banks.

An interesting note is that an HSM does not make the cryptographic constructs used more secure and it does in fact increase the attack surface of a system. It is often not the cryptography that breaks in an attack of a system nor is it what is targeted. The mathematics behind cryptography is usually overwhelmingly strong, it is instead the implementation that is subject to attack. Simply put it is easier to steal a secret key, through a side channel attack or similar, than trying to derive it. HSMs are useful because they encapsulate the sensitive parts of a cryptographic protocol very well. It is a pragmatic approach that when correctly used can increase the security of a system immensely.

### 3.4 Secret Sharing

The basic idea behind secret sharing, also known as secret splitting, is that a secret is partitioned into multiple shares and distributed among the participants. The original secret should then be reconstructable if a sufficient number of participants cooperate and share their secret shares with each other. Shares themselves should have no use on their own, nor provide any information about the original secret. An example is that ten parties, each have a share, out of which seven must cooperate in order to calculate a decryption key for some document. This is in general referred to as a threshold schema and is denoted \((t, n)\) – threshold where \(n\) is the total number of shares and \(t\) is the number of shares needed to reconstruct the original secret.

#### 3.4.1 Schemas

There are a few ways that a secret sharing schema can be constructed as well as trivial constructs for corner cases where \(t = 1\) and \(t = n\) in \((t, n)\) – threshold. In
the case of \( t = 1 \) all participants simply have the original secret. For the case of \( t = n \), which translates to that all participants have to cooperate, we randomly generate \( n - 1 \) binary strings which constitute all but one of the shares \( p_i \). The last share, \( p_n \), is then calculated as follows, \( p_n = s \oplus p_1 \oplus p_2 \oplus ... \oplus p_{n-1} \) where \( s \) is the binary representation of the secret. That way all the participants has to cooperate in order to recover the secret. To recover it, \( s \), we simply XOR all the constructed shares resulting in the secret, \( s = p_1 \oplus p_2 \oplus ... \oplus p_n \).

![Figure 3.5: Degree two Polynomials through two points](image)

There are two major schemas of secret sharing which both use a similar idea behind the mathematical construct. Adi Shamirs schema, Shamir’s Secret Sharing, is one of the two. The idea behind it is in principle very simple and is based on the fact that two points are needed to define a line, three points are needed to define a second degree polynomial and so on. This means that we need \( t \) points on a curve in order to define the polynomial of the degree \( t - 1 \). Having \( t - 1 \) points on a polynomial of the degree \( t - 1 \) will reveal no information of the functions value for \( x = 0 \), as seen in Figure 3.5. To construct a \( (t, n) - threshold \) secret sharing game, with Shamir you start by defining a finite field such that \( 0 < t < n < P \) where \( P \) is a prime number. Then \( t - 1 \) numbers are randomly generated and assigned as \( a_1, a_2...a_{t-1} \), the secret \( s \) is then assigned to \( a_0 \). After the assignments, a polynomial is constructed as follows, \( f(x) = a_0 + a_1x + a_2x^2...a_{t-1}x^{t-1} \). The secret is then the functions value for \( x = 0 \). To now generate the \( n \) shares in the schema, \( f(x) \) is calculated for \( n \) different \( x \) where \( x \neq 0 \). A share now consists of pair \((x_i, f(x_i))\). To reconstruct the original function, \( f(x) \), reveling the secret \( a_0 \) we need \( t \) number of \((x_i, f(x_i))\) pairs. When \( t \) pairs are available the reconstruction is done by preforming polynomial interpolation using Lagrange polynomials[21]. The original function is then computed in the finite field by

\[
f(x) = \sum_{i=0}^{t} f(x_i) \cdot \ell_i(x)
\]
where

\[ \ell_i(x) = \prod_{0 \leq m \leq t \atop m \neq i} \frac{x - x_m}{x_i - x_m} = \frac{(x - x_0)}{(x_i - x_0)} \ldots \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \frac{(x - x_{i+1})}{(x_i - x_{i+1})} \ldots \frac{(x - x_t)}{(x_i - x_t)} \]

The other major schema is Blakey’s. This schema is constructed around the fact that two nonparallel straight lines only intersect at one point. The generalization of this is that \( t \) nonparallel \((t - 1)\)-dimensional hyperplanes all intersect at one point, a simple example of this can be seen in Figure 3.6. To construct a \((t, n)\)–threshold secret sharing game from this we define the secret, \( s \), as a point, \( p_0 \), in a \( t \)-dimensional space. Then \( n \) shares are then defined as \( n \) different \((t - 1)\)-dimensional hyperplanes in which the point \( p_0 \) lies and then distributed to the participants. An important property of the planes is that none of them are parallel with another, meaning that no plane can be described in terms of the others. In order to reconstruct the secret, \( t \) number of planes have to be known to determine the point \( p_0 \) in the \( t \)-dimensional space[3]. Blakey’s schema is less space efficient then Shamir’s as it takes up \( t \) times the space for each share. This can however be reduced by predetermining the planes used in the schema.

![Figure 3.6: Three non parallel two dimensional planes intersecting in one point](image-url)

**3.4.2 Verifiable secret sharing**

In all secret sharing schemas we must trust that other participants tell the truth about there share, in the game, when sharing it. This means that a malicious participant might lie about his or her share in order to gain access to others’ shares and thereby gaining an advantage. To counter this problem all players want to be able to verify that another player is actually supplying him or her with a correct share. A way of achieving this might be that the game maker hashes and publishes the digests of the shares prior to the start of the game. This will however leek information and a share could potentially be constructed without being
shared by any player. A better solution to this problem is to use secure multiparty computation instead, see section 3.5

3.5 Secure multiparty computation

Secure multiparty computation (MPC) is a branch of cryptography where the intention is for a number, $n$, of parties which have individual inputs $x_1,...,x_n$ and wish to compute a function $f(x_1,...,x_n)$ without revealing their own individual input to any other party. The idea of MPC has been around in the cryptographic community for a long time and was exemplified in The socialist millionaires’ problem, where two millionaires are interested in finding out which one of them is richer without having to reveal their individual richness. This problem was stated in the eighties by Andrew Yao and formulated in a protocol described in [24]. Since then the research in the area has been more generalized and protocols for any number of nodes have been invented. It has been proved that both logical AND and XOR gates can be realized in an MPC-protocol. As a result of being able to use these gates, any function can be calculated in MPC. The same thing that is achieved by using MPC can also be achieved by using a trusted third party (TTP), responsible for performing the calculations, and then letting each party get the result back. This however includes a new trust relationship to the TTP which isn’t desired since one of the cornerstones of cryptography is that no one can be truly trusted.

Most protocols used in MPC rely on so called full-mesh networks, meaning that every node taking part in the computation needs to have a connection to every other node. This means a lot of connections as it means that $n^2$ connections will be needed. One way to handle this is to let a number of nodes be dedicated calculating nodes, not having any information as input to the function. When using a protocol of this type the information sent to the different calculation nodes is sent in a secret shared form. The calculating nodes will then perform their calculations on the secret shared information and the result will be in in secret shared form. For this to be considered secure the information sent to the computational nodes is sent in a secret shared form letting the computational nodes work on secret shared information thus never letting any other party know the original information.

As of today MPC is not commonly used in real world applications, as it is not yet widely known and not really effective on large scales and difficult computations. It has been used for performing AES encryption but the time needed for encrypting a single block is much larger than what can be considered usable in a multi user environment. These types of project are, as of today, still mostly research projects. Although it should be mentioned that MPC has been used to carry out an auction on the danish rights to grow sugar beets. In that case there was an issue of trust between the parties involved and the use of a TTP was considered too expensive. This was carried out as an experimental research project and is described in full in [4].
4.1 Method

The method for this thesis work been a deductive one. We started of by formulating a few properties that a cryptographic construct, for in our mind, securely persisting users password, should have. Some properties that exist in current best practices and others that they do not have. They were as follows.

- **The construct shall be at least as secure as current best practice**
  In order for the system to have a reason to exist it needs to live up to the standards of today, at least.

- **The construct shall behave like a one-way-function in the derivation of passwords**
  The information about the plain text password must be near to impossible to obtain, for everyone, as soon as it enters the system. Even though the systems makes use of symmetric encryption, it should never be possible to find out the original password after it has been stored.

- **The construct shall create an asymmetric effort relationship between an attacker and allowed verification**
  The system intends to keep the user experience of logging in as it is today. An adversary however should have to gain access to more than the database in order to be able to start attacking an individual users password.

- **The construct shall provide measures for protecting users passwords in case part of the systems is compromised**
  In case a database leak is discovered their should be ways of dealing with that. More specifically in terms of securing the users’ passwords without
involving the users.

- **The construct shall work with legacy persisting schemas**
  Changing the way passwords are persisted today means that every single user in a system needs to update their password. This system is intended to be possible to deploy without asking the user to update anything. Therefore the system can have no limits on what format the passwords are currently stored in.

- **The construct shall be maintainable**
  The construct needs to be easy to use for a developer and it needs to be easy to handle for an administrator. It should also be easy to handle an increasing amount of users.

Early on we had a construct in mind that, at least at first glance, seems to fulfill the properties above. This lead to researching different cryptographic constructs that had potential to be satisfactory. In doing so we have constantly weighed how usable a construct is in relation to its provable properties. While a provable secure construct is preferable to use, we still deal with a real world problem. Therefore we want to find a solution that is viable in practice, hence taking a more pragmatic approach in both selection and implementation of a construct or model.

### 4.2 Model

In this section we will look at and discuss what cryptographic construct we have selected in order to propose a solution for how users passwords shall be persisted.

#### 4.2.1 Overview

The model, on a high level, consists of a chain of servers each performing AES encryption of the original text sent in. There is no restriction on what is entered into the system in terms of if it is plain text or some form of hashed value. The idea is that each server performs an encryption with an individual key, and then passes it on to the next server level where the same operation is carried out, with exceptions for the first, and last, level. The first level is also responsible for running a key derivation function on the input. This is to make sure that the original plain text password is well protected in case the system is compromised. And as the reader might expect the last level does not pass it on for further encryption, it sends its result back the pipeline to be returned to where the request originally came from. This is what is to be persisted in the database as it is something that can not really be used to deduce something about the original input, as long as the original key derivation function follows the rules described earlier in 3.1.1.

Adding the layers of AES-encryption is intended to make it possible to roll back passwords and then reencrypt them without exposing the original password. As stated the intention is to run the software on several different nodes, which can range from $1..n$ where $n$ denotes the length of the crypto-chain. Each node is
assigned a level, and nodes on level $n$ have a key specific to that level. The intention of running this software as a chain on multiple servers is to add points which need to be compromised in order for a password leak to be a threat. Since we use AES-encryption it is necessary to compromise all the AES keys as any single encryption with AES is considered secure in the foreseeable future, see 3.2.2. In this software construct, running it with a chain of length $n$ means that $n$ keys need to be compromised in order to get back to the result that came from the KDF. Compromising up to $n - 1$ keys does not make it possible to get back something that can be bruteforced since that will always be encrypted with AES. In the case where all layers are compromised, the AES keys can be considered available to the attacker and that attacker can begin to decrypt the entire database back to a state of KDF results, and that means that in a worst case scenario the system can be considered as secure as today’s best practices. There is a difference between gaining access to the keys, and gaining access to the code running on the server. The second case is further covered later in the report.

As it is necessary for a software that is responsible for such a crucial part of a system as logging in, there is a need for redundancy. Therefore no single layer can rely on a single machine to be solely responsible for all requests to that level. Therefore the software has been built in a distributed way, in order to scale both vertically and horizontally. So for each layer added in the chain, more than one machine needs to be added to that layer, with the same configuration. It is then the responsibility of each layer to send its requests to a machine that has a decent response time and is likely to be able to process the request. This is relevant for any service with users all over the world since the user might not be willing to wait for a service, even if the underlying reason is security.

### 4.2.2 Chaining AES

The intention of using a chain of AES is not to make the encryption more secure in the sense that more encryption equals more security. If someone finds a plausible attack on AES, encrypting multiple times doesn’t make it more secure. The AES chain is rather a way to keep it more secure in the sense that more points need to be compromised, for an attack to be successful. The level of security in this systems’ cryptographic construct is still considered to be that of AES. As mentioned earlier the result of an AES encryption can be further encrypted with another cryptographic construct such as the Twofish algorithm. The security in the system does however not rely on being secure for a very long time since the persisted passwords are intended to be updated on a regular basis, which could be a year, a month or what the service provider considers to be a reasonable interval.

A reasonable question is why using more of the same would increase security in any way i.e. if an adversary is able to compromise one of the nodes, why wouldn’t the other nodes be as easy to get access to. The question would in deed be a valid one and there is no clear way to answer it. First of all it should be mentioned that this is not really addressed in the model as such, this is rather something that would be a job for the administrator of the servers to handle. If there are security
holes that can be easily used for taking over a server and thereby extracting the keys, that is something that is not innate to this system. Our recommendation in this matter however would be that running the same setup for all servers is not recommended. The same setup on all nodes would mean that as soon as a security hole is discovered at one node, the same hole can be used for all the other nodes as well. This would effectively make it a system that is relying on a single part of it being secure. A system administrator might for example use different operating systems on each level such as Windows, Ubuntu and OpenBSD.

4.3 Alternative Models

As mentioned above, 4.1, we have looked at a few alternative models according to our criteria and their benefits and draw backs will be presented below.

4.3.1 Replacing HMACs with Hash and Encrypt

One of the current best practices today is the use of HMAC-SHA-256 as a protective function, 2.2. This means that a service provider would have a site wide secret key that is used to compute the message authentication code of all users’ passwords. This is sometimes referred to as using pepper. It looks as follows.

```java
public String protect(String password, String salt ){
    return salt + hmacSha256(getKey(), salt+password);
}
```

Using this implementation it is important that the site wide key is not stored in a database along with the protected form of the user’s passwords. Since a leak of the database would then reveal that key and a regular brute force attack could take place.

This construct has a few very pleasant properties which align well with the ones specified in the section above, 4.1. It does however lack maintainability and countermeasures in case systems are compromised. In the case the database of HMACs are leaked an adversary can not start a brute force attack without the site wide secret key. The downside is that a the site wide key can never be changed, due to the nature of HMAC, which gives an adversary a lot of time to compromise that key. In this sense the model is not maintainable nor does it provide countermeasures in terms of replacing all data in the database.

A, in our minds, better alternative would be to use a cryptographically valid hash-or key derivation function, such as SHA256 or Scrypt, in combination with a secure cipher, such as CTR-AES or Salsa20. This would essential accomplish the same thing as the HMAC way of deriving passwords. It creates, just as the HMAC, a asymmetric difficulty for an adversary in the sense that multiple system has to be compromised in order for an attack on the users derived passwords to begin. The new protected form of the users password would now look as follows.
public String protect(String password, String salt, int version) {
    byte[] iv = hash(version + salt);
    return version + salt +
    encrypt(
        getKey(version),
        iv,
        hash(salt + password)
    );
}

An important note to make here is that the \textit{iv} is, just as the salt, not secret but it is important that they are unique due to the use of stream ciphers to allow for a custom length digest from the hash function.

What this allows for, in comparison to the HMAC, is the change of the site wide key. We can decrypt a user’s protected password and then encrypt with a new key. This makes the construct maintainable and offers countermeasures in case the database is compromised. This is in fact very close to our chosen model, the big difference is that we apply the encrypt function an arbitrary amount of times in our model.

An important implementation aspect of this construct, as well as the HMAC, is that the site wide key that is used shall not stored in the same database nor context that the protected form of the users password is stored. This due to the fact that if the database is compromised the shared site wide key will be as well. A common solution is to have an encrypted key store for these keys.

4.3.2 Secret sharing

The idea behind the model described above is very much to eliminate a single point of failure. Not in the sense that we have multiple machines that do the same thing but rather that multiple machines has to be compromised before the sensitive information is leaked. If one or two points are compromised it will not give an adversary any sensitive information. This gives the defenders of the system a chance to discover an adversary and intervene. In security theory this is generally not a good idea since it does create a larger attack surface for an adversary. The idea that makes this construct more secure is that every that not all security is relying on a single database or system. Instead multiple systems are used where an individual system, or a subset of systems reveals no information to the attacker.

The simplistic approach to secret sharing in the case of storing user’s derived passwords might be to, instead of storing them in one database, store different partitions of the derived password in as many separate databases. This however provides very little protection and with the increased attacked surface of multiple databases might quite possibly be much less secure. For this construct to work we must assume that all portions by themselves reveal no information about un-partitioned derived passwords and their original form. The problem is that it
does; this is due to the fact that cryptographically secure hash functions, which would be the underlying the creation of the derived passwords, are weakly and strongly collision resistant. When we partition the derived password one might expect that particular partition to occur in multiple derived passwords. However a digest, such as SHA256, is so collision resistant that this would very rarely happen if the partitions are big enough, say 64 bits. Using only one of these 64 bit partitions would give rise to more collisions but few of them would be in the subset of bit combinations that are accessible to users, mainly keyboard characters from the ASCII table. What this means is that a brute force attack on a leaked database of partitions is just as possible as if all the partitions were leaked. Hence we would only be decreasing the security of the system by increasing the attack surface and complexity of the system.

A better alternative would then be to use secret sharing. The fundamental concept would still be to use a regular password derivation function, just as in current best practices, for creating a digest. Instead of storing partitions of this digest in different database we use this digest as a secret in a secret sharing schema, such as Shamir. This means that, when enrolling a password, we generate \( n \) shares from the digest which are all stored in separate databases and \( t \) of them has to be online later in order to reconstruct the original digest for comparison upon user authentication. The benefit with this schema over the earlier mention one is that even if all shares from \( t - 1 \) databases are accessible to an adversary it provides, mathematically provable, no information about what the original digest is. Hence the adversary learns nothing about what a users derived password is.

Shamirs secret sharing also allows for another important aspect which is that in the case where we discover that an adversary has compromised part of the system we can generate a new polynomials and shares for all stored derived password. After this is done we can safely destroy all old shares in our system. This means that even if an adversary had gained access to \( t - 1 \) of the databases, and all its content, it now becomes useless and it would be impossible to use that information to discover anything about the original derived passwords. The benefit of using this construct in this context is that changing the shares can be done in offline mode, without any user involvement, as well as periodical a preemptive measure. This gives it both the property of being maintainable and offers countermeasures in case part of the system has been compromised.

There are however two main draw backs to this construct. The first is that the storage of the shares would be \( n \) times as large as the original digest. The second is, when using Shamir, that one service would have to be able to request all shares for one digest in order to reconstruct it for comparison when a user is trying to authenticate. If that service were to be compromised an adversary would in one sweep gain access to all the shares for all users rendering the construct no safer then if all the digest had been store in the same database. If however \( t = n \), and no redundancy exist, a distributed chaining of XOR could be constructed in order to at least offer some more protection in the sense that bulk selects from a database could not be done by the requesting service.
Some of this problems can be solved with secure multiparty computation.

### 4.3.3 Secure multiparty computation

As mentioned in 3.5 there are ways for an arbitrary number of parties to securely compute a function together, without revealing their individual part of the input. This could theoretically be used to derive some form of hash on a password that can only be calculated by those specific parties. One schema of achieving this could be that of letting a number of nodes collaborate on computing an AES encryption on a hashed value of the password. This would have some nice properties as it would mean that all passwords in a running services would be possible to update, which is one of the requirements desired in this system. It would also imply that gaining information of any node would not present a vulnerability in the system as the nodes would collaborate using information that gives them no knowledge beyond what they know themselves.

As one of the requirements on the system is that it should be at least as secure as today’s best practices the password would have to be hashed before it’s secret shared and sent to the computational nodes, in a secret shared format. The encryption part of the protocol is there to fulfill the requirement of being able to have countermeasures in case the database leaks. By letting a number of nodes collaborate on encrypting the password the asymmetric requirement would be satisfied, as an adversary would not be able to decrypt without gaining knowledge about every computing party’s information. An example of a MPC protocol that have been implemented for AES encryption can be seen in [7]. In this protocol the key, the cleartext and the ciphertext can all be communicated in a secret shared format. If this were to be implemented for password storage, the cleartext would not be cleartext, but rather a digest from a hash.

This construct, however, does not hold up in the real world as it simply is not fast enough to be usable in an environment where multiple users need to get access to their accounts at high rates. Since MPC is not efficient enough to be used in the real world this model is considered mostly as a concept that might be practical in some, not so distant, future. The idea of letting multiple parties collaborate to compute some function seems good in practice but as a possible implementation the authors of the report consider using secret sharing alone is a more reasonable approach.

### 4.3.4 Calculations on client

Making use of a CPU-heavy function with a configurable memory footprint like Scrypt certainly have some nice properties, as mentioned earlier. It can however be a problem to make efficient use of. Setting it to have a large memory footprint and letting some server machine calculate the function poses a problem, given that the user base is large enough. The number of machines needed for handling login and sign up would grow as the user base grows. That is the case no matter which function is used of course, but if something like Scrypt is used, the number of machines needed will grow at a faster rate. As money is limited this might be
implausible on a larger scale. Making use of such functions on servers might also increase the risk of being vulnerable to denial of service attacks since the server will be occupied for a long time every time someone tries to log in, or sign up. A provider can try to mitigate this by using some a CAPTCHA or some other proof of work schema making it harder to flood the server with requests.

An alternative model that can potentially scale in a better way would be to let the client perform the bulk work, as opposed to doing it on the servers. This would imply that it is possible to run Scrypt with performance heavy settings, effectively rendering bruteforce attacks unfeasible for someone gaining access to the database. Knowing that what comes in to the server is already sufficiently hashed implies that the server can perform an encryption on it without having to worry about first making sure that if the encryption key leaks, the plaintext passwords leak. Making use of this would satisfy both the needs of security as well as the needs of having countermeasures mentioned in 4.1. Seeing as how this in a lot of ways seems to be a good model, let’s address some of it’s drawbacks.

As services grow in user base, it is not uncommon for them to extend to mobile platforms. This poses limitations in what type of calculations can be carried out on the client side. A regular smartphone today would not be particularly happy about running Scrypt, this would mean that some form of benchmark would be needed and then tuning the algorithm to run efficiently on the device. Doing so might result in running the algorithm in such a way that it is not difficult to parallelize it and perform a bruteforce attack in case of breach. The very intention of running the computational heavy algorithms on the client side might therefore work in a counter productive way. In theory this is a nice solution and given that limitations on hardware were not a problem, it would be a great solution. Unfortunately there are such limitations and this implies the world probably isn’t ready for this solution just yet.

4.4 Realization

In this section we will discuss our implementation of the selected model from above. This will not only be limited to the realization of the cryptographic concept but also discuss the decision and implementation of infrastructure needed around it.

4.4.1 Concept implementation

The software that has been developed is rather large and complex in order to deal with distribution of computation, redundancy, security and different operations. Our basic concept of deriving a password can still be described in relative few lines of code as follows.
4.4 Realization

```java
public byte[] deriveKey(byte[] password, byte[] salt,
            int version){
    byte[] digest = kdf(password, salt);
    return encrypt(digest, salt, version, 0);
}

//key is in this case the output from deriveKey
public byte[] deriveNewKey(byte[] key, byte[] salt,
            int currentVersion, int newVersion){
    int depth = getLevelDepth(currentVersion);
    key = decrypt(key, salt, currentVersion, depth);
    return encrypt(key, salt, newVersion, 0);
}

private byte[] encrypt(
            byte[] digest, byte[] salt, int version, int level){
    digest = crypt(true, digest, salt, version, level);
    if(level != getLevelDepth(version))
        return encrypt(digest, salt, version, level+1);
    return digest;
}

private byte[] decrypt(
            byte[] digest, byte[] salt, int version, int level){
    digest = crypt(false, digest, salt, version, level);
    if(level != 0)
        return decrypt(digest, salt, version, level-1);
    return digest;
}

private byte[] crypt(boolean encrypt, byte[] digest, byte[]
salt, int version, int level){
    byte[] key = getKey(version, level);
    byte[] iv = sha256(salt, version, level);
    return aesCtr(encrypt, key, iv, digest);
}
```

The deriveKey() method above is now used for deriving a new key from a user’s password and in essence acts like a one-way-function, like SHA256 or any KDF. It is called when enrolling a new password coupled with a user. The version and salt shall then be persisted alongside the resulting digest. To authenticate a user after enrollment the same method is used, a simple comparison is made on the result from the deriveKey() method and the already persisted digest and if they are equal the user would be authenticated. From an implementation point of view this makes it act like existing protective functions.
The method deriveNewKey() would be used in the case of a breach of the database or periodically for safety. The leaked digest and version number could then be replaced by the resulting digest and the new selected version. Once all the digest and version numbers have been replaced with new ones the AES keys coupled with version number of the leaked digests would be destroyed. This renders all the leaked data useless and the leak itself becomes irrelevant as long as at least one of the AES keys used remains secret and uncompromised.

The relationship between deriveKey() and deriveNewKey() is then the following:

\[
\begin{align*}
\text{digest}_0 &= \text{deriveKey}(pwd, salt, version_0) \\
\text{digest}_1 &= \text{deriveNewKey}(\text{digest}_0, salt, version_0, version_1) \\
\text{digest}_2 &= \text{deriveKey}(pwd, salt, version_1) \\
\text{digest}_1 &= \text{digest}_2
\end{align*}
\] (4.1)

What this means is that when deriveNewKey() is performed the result would be the same as if deriveKey() was used with the new version using the original plain text password. The next time a user is trying to authenticate the new version would be used when deriving the key to be compared. This construct makes it possible to change the hashes in offline mode, without users' involvement, and could be compared to the user changing his or her password in the paradigm of current best practices.

This simplistic model only show how it works and is not recommended to be used in the real world. This is because it would run on one single machine that would contain all the AES keys related to all versions and level. It means that there would be no reason to have more than one AES-key or level per version. In fact this would be the same thing as the discussed alternative model in section 4.3.1, the only difference would be obfuscation. What we have done, in our implementation, is replacing the recursive calls in the encrypt and decrypt methods with RPC calls to different machines, thereby increasing the attack surface while providing no usable information unless all machines are compromised in a no redundancy scenario.

### 4.4.2 Our implementation

As mentioned the concept itself is trivial while the infrastructure surrounding it is not. Much of the effort in this thesis work has been put into creating that infrastructure. The goal has been to create a system that encapsulates the construct and have it act like a hash oracle service. This has been done by making the system independent from any outside systems, such as databases, DNS or third party transport systems or protocols. This is mainly to reduce the attack surface and unexpected consequences.

#### Inner workings of key derivation

The process of performing a key derivation is distributed in order to reduce the consequence if one node or a few nodes...
are compromised. We refer to the nodes communicating with outside systems as border nodes and all nodes are assigned to a level. The border nodes are always on level zero. All digests produced by the system are tagged with a version, which in combination with a level describes which AES key to use. This means that AES keys are always local to a specific level and is never communicated outside of it.

A node is in this case a separate machine running our software with slightly differing configuration.

Figure 4.1 illustrates a request to the service for deriving a key. It is only $node_0$, in the illustration, that allows an outside request to be made to and is therefor the border and level zero node. The other nodes only allow traffic and requests from nodes in the system and are at level one and two respectively. When a outside service makes a HASH-RPC request to the border node, it provides the node with an initial digest, a salt and which version to use when deriving the key. This digest can be a plain text password in byte form or a precomputed digest in order to deal with legacy implementations. The first thing the border node does is applying a key derivation function on the provided digest using the provided salt and preconfigured settings. What KDF is used and the length of the final derived key is also up to the user where our system, at the writing moment, supports Scrypt, Bcrypt and PBKDF2-SHA256.

The border node then turns in to a regular node in the network. It gets the version from the request and look up the associated AES key coupled with the version and level zero. This will be used to perform AES-CTR encryption on the resulting digest from the KDF. We then derive a new IV from the salt, version, and node level using SHA256. The IV is not necessarily secret but since we use AES in CTR mode it becomes important to use different IVs for encryption, as with all stream ciphers.

Once the digest has been encrypted the result is sent to a node at the next level, $node_1$ at level one in this case. $Node_1$ then performs, just as the border node, an AES-CTR encryption on the result from the previous node but with a level specific key and a new unique IV.

This step is repeatedly performed on all nodes in the defined chained until the last node is reached, $node_2$ in the illustration. Once the last node has applied its own layer of encryption the result is returned to the original border node, $node_0$, which returns it to the requesting service.

**Replacing a derived key**

One of the main reasons for using the method described above for key derivation is that it is partially reversible, mak-
ing countermeasures possible in case of a leak. This is done by simply decrypting a resulting digest and then encrypting it using a different set of keys. Figure 4.2 illustrates the flow through the nodes for this use case.

In the scenario that some part of the system has been compromised, the first action would be to fix the problem. The later step is to ensure that the compromised data, in this case database entries or leaked keys from any of the nodes, can not be used to gain any information at a later date.

The process of replacing all the derived keys, referred to as rehashing by the authors, would start by supplying all the nodes with a new set of keys coupled with a new version number. This is done in a very implementation specific way where one node in each layer is asked to generate a new AES key and couples it with the new version in memory and on disk.

Once this is done an outside service would start iterating through a database with current user coupled derived keys. For each key a request of rehashing it would be sent to the border node, node0 from figure 4.2. This request would contain the current digest, salt, version of the digest and the new version we want the digest to have. The border node will start the process by sending the request to the last node in the chain, node2. Node2 then proceeds to decrypt the digest by using AES-CTR with the key, coupled with its level, version, and the IV that is derived from the salt, version, and level. Once node2 has done this it sends the partially decrypted digest to level-1, node1 in this case which repeats the process. When the chain of decryption reaches a border node it decrypts the digest, just as all the other nodes, but then it starts the encryption process described in section 4.4.2.

When encryption is done the digest is returned to the external service and can now be persisted to the database, replacing the old digest. Once all the digests from the database has been subject to this process all the old keys on the nodes shall be destroyed. This destruction is done by an rpc request to the effected nodes. When the keys are destroyed and an adversary is missing at least one piece of information, meaning that he or she is missing at least either one of the keys or the derived keys, there is no way to perform any realistic form of attack on the data in order to retrieve or reconstruct a user’s passwords.

**Distribution and redundancy management**

As mentioned before this construct is rather simple and can be described in a few lines of code. For it to be useful the key set has to be handled in a distributed way in order to prevent one point of compromise. Our implementation is made to be as generic as possible in this case. The service can be run on a single machine with a single layer or scaled to an arbitrary number of layers with an arbitrary number of machines on each level.
The scaling of the service could easily be done by simply providing all of them with a configuration containing all information needed about the other nodes. This is however not a pleasant way of scaling a software due to the increasing demand for configuration. Instead we have tried to keep the configuration to a minimum. Aside from the obvious benefit of not doing a large amount of configuration the cluster can amongst itself decide on AES key and version configuration which can reduce the attack surface. Simply put it can reduce the attack surface since no sensitive information has to reside outside the cluster, on a system administrators computer or such.

The configuration needed to make the distribution part of the software to work only requires an address and port to one other node in the network and a cluster key which is used to verify that the node shall participate in the cluster. This way nodes can be added and removed from any level in the cluster at any time. It is accomplished by first letting the border nodes create a mesh amongst themselves. When a node is connected it announces itself to a node in the mesh, and this update to the typology is eventually propagated out to all other nodes. This allows for scaling a cluster with minimal configuration needed.

### 4.4.3 Frameworks

The software built as the implementation part of this master thesis relies heavily on a framework called akka. What akka is is a scala implementation of the actor model. For this thesis all code is written in Java, as akka exposes a Java API. To give a fair explanation of what akka is and why it was chosen, let's first give a brief explanation of what the actor model is. The actor model is an idea of not keeping track of data using objects and states. Instead the only thing that's passed around is messages. It is an event driven paradigm where an actor simply receives a message and then performs some logic depending on the content of that message. What the actor model is supposed to bring to the table is a way of building scaling applications without having to worry about the sequence of execution. Since an actor will only ever perform some action whenever it gets a message, it is an inherently concurrent model. The actor model was chosen for this project since the only things that needs to be kept track of can be passed along with imutables, and since it needs to be able to scale to many concurrent requests it seemed like a good idea to avoid writing the concurrency model and thread handling ourselves.

So what is akka? Akka is, as mentioned, a framework developed to make use of the actor model with Java code. What the framework offers to the developer is a nice way to handle scaling of an application to basically any size needed as the framework supports remoting and distribution out of the box. This means that messages can not only be sent within the same JVM but they can also be sent to other machines running the same type of setup. Since all of the servers in this application, with some minor exceptions, run in the same way it will only need to be configured with own keys and then deployed in the same way on every server. The actors are made use of in this system by letting them be senders and recipients of the RPCs sent between servers. When receiving a message an actor
will check what type of action it should perform and then perform that action and pass it on to the next server in the chain, with information to that server about what the next action should be.
Now that some theory has been laid out and an experimental implementation has been implemented, it is about time to revisit the questions formulated in the beginning. In this chapter the intention is to give some answers and deduce whether the proposed solution solves some of the problems currently present. Since the authors have opinions in this matter there will be reflections based on those as well. The intention is to make it clear for the reader what is based on scientific research and what is opinions. It is the opinion of the authors that this doesn't pose a problem as long as it's made clear which is which. Just to freshen the memory of the reader, the questions that will serve as ways of analyzing the material are the following.

- What security concerns does a distributed password hash model raise and what types of attacks are feasible?
- What is the security complexity difference between our cryptographic construct and other commonly used ones such as Scrypt, PBKDF2SHA256 and HMACSHA256
- Is distributed hashing and an upgradeable schema worth considering?

### 5.1 Security concerns

When using a distributed schema in order to hash passwords more places that can be attacked will of course be present. This is intended in this system as the idea is to make it impossible to compromise only one part of the system. More places to attack should not be mistaken as meaning that successfully attacking any of those places fully compromises the system. Unless this is addressed
most likely won't give any extra protection since the same type of attack then could be used to attack every server. This can in some ways be mitigated by making the servers different in some sense, otherwise all servers can be attacked using the same type of attack. Consider a scenario where six machines, thus six levels of encryption, are used to perform the hashing of passwords. If all of these six machines are running the same distribution of Linux, then finding a way into one of them would imply finding a way into all of them, thereby gaining access to all the AES-keys and being able to decrypt what's stored in the database. In this scenario what would be exposed to the attacker is the digest from the KDF which the password went through.

This does however become extra sensitive when the server at the very first level is compromised, since the password is being sent it would mean that compromising the first level would imply that it's possible to listen in on what is being sent there, and thereby gaining knowledge of the password before it's even run through the cryptographic key derivation function. In the case where what's sent into the system is a legacy hash using some older and possibly deprecated cryptographic hash function such as MD5 or SHA-1 the attacker would gain access to something that would be possible to perform an offline bruteforce attack against. The digest would however have to be collected by an adversary in an online setting, meaning that only digests of users trying to authenticate would be collected.

5.2 Complexity

Regarding the complexity of the system we are limited to what has been done in the field regarding how secure the used algorithms are. It's not feasible for the authors to perform cryptanalysis on any algorithms chosen as we lack both knowledge and time. So what we can say about this matter is that the system is, according to cryptanalysis of the algorithms used, safe in regards to algorithmic complexity. As can be seen in 3.2.2 it is not feasible to attack something that has been encrypted with AES. The only feasible way to attack the system would therefore be to gain knowledge about AES-keys by attacking the system itself.

The default KDF used in the system is Scrypt, which is a fairly new function. When it comes to functions of this sort it might be hard to find an indisputable mathematical proof stating that the function can actually be regarded as secure. So basically what can be looked at is performed cryptanalysis on functions and see if they stand the test of time. When it comes to Scrypt in particular it's to the best of our knowledge considered a secure algorithm that is hard to parallelize using GPUs and therefore hard to perform bruteforce attacks against. It can be seen in [19] that as far as we know it is in fact the case with Scrypt. One metric used in that particular paper is dollars that need to be spent in order to crack a password of a particular length in a year. The metrics show some interesting things that are relevant to any system used to persist passwords, the most important being that the money that need to be spent is in a concerning way related to the length of the passwords. When dealing with passwords no longer than six
5.3 Distribution and upgradability

As can be seen throughout the theory presented earlier in this thesis the concept of moving from a single point of security is something that is considered a more secure way of handling data. This can be seen in both secret sharing and secure multi party computation. None of these concepts are new in the cryptographic community since protocols for using them have been around for a long time. They do however work as a proof of concept that using multiple nodes for communication can in some cases provide a higher security, that can be proven. As this is more or less the intention of the system created during this thesis it is considered, in theory, to be a good direction to move in. By distributing the hashing and encryption model some of the proven concepts of cryptography are put to use, not in the same way or using the same protocols, but using the underlying idea of letting multiple parties contribute in order to secure data.

Another important feature of this system is the possibility to handle a leak of the database in offline mode, thereby doing it without the involvement of the user. This is really only worth considering as long as the user’s password can be handled in a secure way, i.e. never being exposed to anyone because of the upgrade. Performing the upgrade in this system is done completely internally since all the information needed for upgrade is present within the system itself, it’s capable of generating and distributing keys to all nodes within a running system. It is therefore considered a safe operation and the most critical part of the system is the entry point where the original digest enters the system.

5.4 Side channel attacks

Since there are multiple nodes to attack in this system, side channel attacks need to be taken into consideration. What is meant by a side channel attack in this context is what happens when successfully compromising one of the nodes further into the chain, i.e. a mid level node. Gaining full access to a node in the middle would imply that the length of the chain is shortened to the number of nodes running below the one that is compromised. Since owning a mid level node re-
sults in it being possible to listen to the incoming traffic on that node, it would at that point no longer matter if it’s encrypted further down the road since it would already be known what that node is sending out. As long as the compromised node is not a border node, what can be intercepted has already been run through a strong KDF and then been encrypted with a strong cipher. It would however render all the nodes later in the chain completely useless since they would no longer need to be compromised in order to attack the passwords.

If an adversary is interested in taking down the system entirely, effectively rendering it unusable, all nodes on a level would need to be compromised. The system is designed to be redundant so just taking down one of the nodes would not take it down entirely. In the case where an adversary gains access to all nodes on a single level, it would be possible to perform a denial of service attack by killing those nodes. If all the servers go down to the hands of an adversary, all legit users might try to gain access at a later point when the service comes back online. This unfortunate scenario could result in an unintentional denial of service since a service with a lot if users could overflow itself even if all the requests comes from legit users.

### 5.5 Oracle attack

The most sensitive node in the system is considered to be the first one, partly because it gets the original digest sent to it. There is also another sensitive aspect that potentially makes the border nodes more vulnerable. If an adversary were to gain access to it and therefore be able to send its own requests to it, the system could be used as an oracle for bruteforcing passwords. The attack can’t be considered a bruteforce attack in the traditional offline way, but rather as a way for an adversary to continuously send their own requests over and over and look at the results and compare them to a database. First off, it should be mentioned that this is a slow process as each request sent to the system will be bound by the time it takes to execute a request. That time will not be insignificant as there is a configurable KDF running for each request. As opposed to a traditional brute-force attack where the same functions running on the system are run in a faster setup, this is a way to use the actual systems to perform the bruteforcing for you. It is not as fast, but the costs for doing it are basically non existent.

This attack has similarities with just using the client interface for guessing, this however is not really the same thing. The client will probably have some rate limits that keeps it from guessing an infinite number of times e.g. a timed lockout after x failed login attempts. This can also result in a denial of service since the system might be jammed by the adversary’s requests for as long as it’s being used for the adversary’s malevolent purposes. Ways of mitigating this can be seen in 5.6.2.
5.6 Future work

During the work on this thesis we have, as mentioned, implemented a prototype of this software. It is however no where near ready to use in production and requires a lot more work in order to put it in to a usable state. The primary discussion of this section will relate to the changes and future work needed in order to do so.

5.6.1 Distribution

As of now the software runs and works in a distributed manner as described above. Constructing distributed systems is however very hard. They give rise to a lot of different, hard to solve, problems such as race conditions and making it hard or impossible to determine sequence of events, causality becomes indeterminable. There is of course ways to solve most of the problems associated with the paradigm, but it is hard and requires a lot of work and an expertise in this particular domain. The authors have made their best efforts in creating something stable and the system acts accordingly under test conditions, with actions happening in the correct order. This is however far from certain to happen during production. The main problem with our paradigm lays with that a global state of nodes, keys and versions has to be maintained.

There are two main ways forward in solving this problem. The first is to enhance the already existing code with vector clocks and other feathers to ensure the global state and the causality of the system. As mentioned it is not an easy task and should really be done by a domain expert and it would mean a considerable amount of work.

A better way forward might be to implement a third party software that is designed to keep track of a global distributed state. This can be done using software such as zookeeper which originated in the hadoop software stack. It has however been broken out to its own product and is released as an open source project in the Apache foundation.

5.6.2 Rate limiting and proof of work

One risk of using a computational intensive way for user authentication is that the system makes a perfect target for denial of service attacks. In our case it is the use of an key derivation function, Scrypt, pbkdf2 or bcrypt, that makes the system vulnerable for this types of attacks. If, let’s say, we tune our key derivation function to run in 100 milliseconds at full computer load. If an adversary now sends more then 10 messages a second it will effectively queue up other traffic. Simply put, the system very easily gets overloaded due to the asymmetric relationship between the amount of data received and the computation needed to evaluate it. An adversary could easily bring down a number of border nodes this way effectively denying any legitimate authentication requests to be processed.

A simple way to reduce this risk is to put another service in front of the border nodes that enforces a rate limiting. If an adversary however finds its way to a
border node anyway this is of little use. A better and suggested way could be to enhance the border node to client protocol to include a proof of work schema. The goal is to ensure that an adversary does not have an asymmetric advantage on needed computation compared to the servers. There are many such schemas and one, hashcash, relies on any hash or one way function and is easily designed. The protocol can be implemented as follows.

- A client request a border node to use its service
- The border node responds to the client with a challenge consisting of a nonce, a time stamp, a hash function to use and a maximum value of the resulting hash.
- The client takes the nonce, appends the time stamp and some random bytes or a counter. It then hashes all of it and checks if the resulting hash digest is less then the maximum the value the server provided. Other wise the process is repeated until the hash digest is less then the maximum value
- Once the client found the random bytes or counter that generates the hash, it sends it back to the border node.
- The border node receives and verifies that the random bytes or counter indeed results in a hash digest smaller then the one it originally provided. If so the client is authorized to make one request of the border nodes services.

The implementation for solving the challenge on the client side could look as follows

```java
def solve(long nonce, long timestamp, byte[] maxValue, Hash hash):
    long counter = randomLong();
    while(hash.digest(nonce, timestamp, counter) > maxValue):
        counter++;
    return counter;
```

The amount of work the client has to do before a client is able to use the service is with probability easily calculated. If the first bit, big endian, in the hash digest has to be zero, half of the digest maximum value, the client has a 50% probability to guess right the first time, 75% chance the second and so on. If the first two bits have to be zero the first guess is at 25% to be correct the second 43.75%. By increasing the number of starting bits that have to be zero, meaning decreasing the maximum value of the hash digest, we can tune the amount of work the client has to do before the use of the service is allowed. The number of tries the client has to do before finding a valid digest is exponential to the number of starting zeros due to the collision resistance of cryptographic hash functions. The border node on the other hand only has to do one.

By tuning the amount of zeroes we can see to it that a client has to do as much computational work as our service when performing the clients request. Meaning
that if the key derivation function takes 100 milliseconds to compute the client proof of work shall take at least the same amount of time to complete.

Since this service always shall be implemented behind another service and not directly facing the client the protocol has to be proxied to the client. Meaning that it might not be the most convenient solution but could be a pleasant configurable enforced protocol.

5.6.3 TLS

In the prototype system the communication between the nodes is not being encrypted. This is something that should be altered in case this system were to ever go into production. A good idea is to add TLS support and make use of certificates. Using certificates will have some benefits regarding security. First of all using the TLS protocol means that it’s easier for any given node to know more precisely with whom it is communicating. Preventing requests from unintended sources will work as a way of mitigating denial of service attacks since the server will not keep connections to someone outside the system.

Further it has the benefit that the traffic between nodes can be sent in an encrypted manner. Encrypting the traffic to and from every node will result in it being difficult to deduce any information from eavesdropping. In the TLS protocol, messages can also be verified with authentication codes. As a result, any given node can verify that what was sent to it was what the sender intended.

5.6.4 Multi language clients

One important aspect of the developed product is that it is accessible and supports many different platforms. Even though the border node to client protocol, as is today, is rather straight forward to implement it always makes it easier for users to have a client API implementation. Since the service has been written in Java it has a easy to use client but it would be preferable to implement the API as a client in other popular programming language used primarily on the web such as python, ruby, javascript and go.
Neither concept nor the implementation of this thesis work has yet been shown to work well with a large number of users as the system has not been tested with a large database where all the passwords have been updated. The implementation has however been tested to work in encrypting a lot of random data, that has not been persisted. As far as the authors can tell the system works fine. In order for this to be considered secure more testing needs to be done and it needs to be shown that it in fact works well with a large number of database entries.

One of the core concepts of this system is the ability to have countermeasures in case users’ password hashes leak. This part of the system is considered to be shown to work as tests have been carried out where passwords are rehashed with new keys. What this implies is that this concept has, in this regard, its merits and a rather considerable advantage over current best practices. Even if this advantage is not all that pressing today, seeing as how it is quite difficult to bruteforce a password stemming from a system where Scrypt is used, this advantage might be more present in the future. What can be said though is that a since lot of systems are in need of changing how they protect users’ passwords today, there might be good reason to change to a schema that is able to work with legacy systems. This will also imply that there is no reason for a service provider to keep a deprecated version of the database, as in the case of the Adobe database leak. For the services that need to update, it could potentially be reasonable to look at an upgradable model. For services currently using current best practice the best way to go is probably to keep their system as is.

If we can ensure that the communication between nodes is secure some things can be stated to be true about the system. The most important regards the overall security of the system. Since there are no inhouse cryptographic constructs being
used, we can state that the system is at least as secure as the KDF the system uses, Scrypt by default. If no such claims can be made it can be stated that the system is at least as secure as a legacy system that makes use of it. These claims can be made due to the fact that a KDF is by default not reversible. Since the algorithms are considered secure in their own right, they are also considered secure to use in conjunction. If something passed through Scrypt becomes easier to bruteforce if it is also encrypted with AES, serious flaws in the algorithms are present. Had the system been entirely dependent on reversible ciphers, the systems security would very much rely on safeguarding the keys. How secure the system would be in that case can not be approached without the context the system is running in and in this report the intention is to limit conclusions to the system itself.

The system implemented is only to be considered a concept implementation. What has been shown is that it is possible to create such a system and to use it to derive hashes to be persisted to storage. The authors are still fairly convinced that making use of both one way functions and encryption is a good way to go as it would address the fact that databases will always leak and be compromised.


[9] Arthur Evans, Jr., William Kantrowitz, and Edwin Weiss. A user authen-


Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagens hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© David Odelberg and Rasmus Holm