Problem Description

Human Computable Passwords - Design and Analysis

Managing passwords is a significant problem for most people in the modern world. This project will be based around the paper "Human Computable Passwords" by Blocki et al. [BBD14], proposing a method for humans to be able to re-compute passwords from public and reliable storage. Passwords are calculated using a memorized mapping from objects, typically letters or pictures, to digits; the characters of the passwords are then calculated in the users head, using a human computable function.

The main objectives of the project can be summarized as the following:

- Understand and compare the "Human Computable Passwords" scheme with other related password management schemes.
- Design and implement a password management scheme applying the ideas of the scheme.
- Analyze if the construction could be utilized to provide secure password management in practical situations.
- Validate if the scheme is feasible to use, comparing the user efforts required to the security rewards.

[BBD14] J. Blocki, M. Blum, and A. Datta, “Human Computable Passwords,” CoRR, vol. abs/1404.0, 2014.

Assignment given: 12 January, 2015
Student: Anders Kofoed **Professor:** Colin Boyd, ITEM

Abstract

Password management is a major issue in the Internet centric world. This project presents the human computable password management scheme by Blocki et al., which makes it possible for human users to calculate passwords from publicly available challenges. The scheme is evaluated in terms of usability, and parameters affecting it discussed. Two applications are designed and implemented, one as a Google Chrome browser extension, and one as a web application.

The Chrome extension implements the scheme, utilizing the strengths of browser extensions with accompanying APIs. It handles challenge generation, management and storage, using the Google account of the user to keep the data persistently synced. Smart functionality provided by the Chrome extension framework makes it possible to monitor the site users visit, allowing the application to display the correct challenges without user interaction.

The second application is a web application built as an experiment and demonstration site. It demonstrates the scheme and allows users to learn the scheme by trial and error, then asks them to calculate challenges while recording calculation times and failure rates. The gathered data is analyzed using an exploratory approach, trying to find interesting characteristics related to usability.

The experiment gave indications that the scheme might suffer from high failure rates, limiting usability for some users. The failure rate was measured to be 0.0585, approximately every one out of 17 calculations was wrong. A measure to limit the consequences of this observation is suggested by categorizing the accounts, having different length passwords for different accounts.

Both applications were designed to investigate if the scheme could be implemented in a usable way, and if so, provide strong enough security to justify the efforts required of the users. The Chrome extension lowers the threshold for using the scheme, solving problems related to challenge management and presentation. The conclusion from the experiment was that failure rates are indeed an important usability factor which should be investigated more thoroughly, as it may limit the scheme severely.

Sammendrag

Passordhåndtering er et stort problem i den Internet-sentriske verden. Dette prosjektet presenterer passordhånderingssystemet "human computable passwords", laget av Blocki et al. [BBD14]. Systemet gjør det mulig for brukere å kalkulere passord ut ifra offentlig tilgjengelige objekter. Systemet evalueres med hensyn til brukervennlighet, og påvirkende faktorer diskuteres. To forskjellig applikasjoner implementeres, en Google Chrome nettleserutvidelse og en webapplikasjon.

Chrome-utvidelsen implementerer passordhåndteringssystemet, og drar nytte av styrkene en nettleserutvidelse tilfører. Applikasjonen tar hånd om generering, administrasjon og lagring av objectene som brukes til å kalkulere passord. Google-kontoen til brukerene gjør det mulig å lagre informasjon persistent. Smarte funksjonaliteter, muliggjort av Chrome-utvidelsesrammeverket, gjør det mulig å overvåke sider brukerene besøker. Applikasjonen viser de riktige objektene uten brukermedvirkning.

Den andre applikasjonen er et eksperiment lagd som en webapplikasjon og demonstrasjonsside. Passordhåndteringssystemet blir presentert og forklart, før brukere får mulighet til å prøve det. Brukerne blir bedt om å regne ut passord på tid. Kalkuleringstiden og feilraten blir så lagret for hvert forsøk. Dataen blir analysert på en utforskende måte, på utkikk etter interessante egenskaper relatert til brukervenlighet.

Eksperimentet viste at feilraten var høy, noe som kan hindre brukervenligheten for noen brukere. Feilraten ble målt til 0.0585, tilsvarende sirka 1 av 17 gale utregninger. Ved å katergorisere brukerkontoer begrenses konsekvensene av den høye feilraten, forskjellige kontoer får forskjellig passordlengde avhengig av sensitivitet.

Målet med begge applikasjonene er å utforske om passordhåndteringssystemet kan implementeres på en brukervenlig måte, og om innsatsen det koster brukeren er liten nok i forhold til sikkerheten systemet tilbyr. Chrome-utvidelsen senker terskelen for å begynne og bruke systemet, ved å løse problemer knyttet til administrasjon og lagring. Experimentet konkluderer med at feilraten er en viktig del av brukervenligheten til systemet, og bør utforskes nærmere, da en høy rate kan begrense systemet kraftig.

Preface

This report is the result of the master's project completed in the 10th semester of the Master's Program in Communication Technology at the Department of Telematics (ITEM) - Norwegian University of Science and Technology (NTNU).

Throughout the work with the thesis, I have been able to learn a lot about both application development and password management schemes. Having a practical approach to the research has been both interesting and inspiring.

I would like to thank my supervisor Professor Colin Boyd for all the help and guidance in regards to the project and the thesis.

Finally I would like to thank Ingrid, for both help with proofreading the report and as a participant in the experiment. Thank you for 5 lovely years in Trondheim; you have been, and are, the most important person in my life.

Trondheim, June 8th. 2014
Anders Kofoed

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Active site The site which the user is currently browsing.

Associative memory Lossy but secure storage. In this context it is the human memory.

Content script A script which is injected in the page loaded in the browser, can monitor fields and user activity, but does not have the permissions of the browser. Used in this project to monitor the URL and password fields of active sites.

Extensions core The core of the Google Chrome extension consisting of the application view and a background page. Responsible for the user interface and business logic.

HCP scheme The human computable password management scheme of Blocki et al. [BBD14].

Human computable function A function f computable in the human mind. Can only involve simple operations, and limited amount of terms and operations.

Password challenge A set of single digit challenges, after applying the correct human computable function yields a complete password.

Password management scheme Scheme or technique helping users remember passwords, without actually storing them.

Password management software Applications storing passwords for users.

Persistent memory Equal to writing something down, thus not secure and can be assumed to be publicly available.

Rehearsal schedule A series of points in time when an object-digit relation is rehearsed. It is deemed sufficient if a user can keep a relation in memory without forgeting it, by following the schedule.

Single digit challenge A challenge consisting of 13 randomly chosen objects, after calculation, yields one charater of a password.

Visitation schedule How often users visit their accounts.

Chapter 1
Introduction

Passwords have come to be the one authentication method used by nearly all Internet sites and services. Guidelines and policies instructing the users on how their passwords should be, and how they should be maintained are seen a lot. The problem with most of these recommendations, are that they expect too much of the users. Passwords should be long and complicated, changed every month and not reused on more than one account. These are all commonly seen as recommendations given by sites on the Internet, and it is clear that no user will ever be able to fulfill them without using some kind of scheme or by adapting their passwords to circumvent them. Password reuse or password schemes improvised by the users lead to a major loss in security, which, of course, is the opposite of the intentions when requiring passwords to be long and complex.

It does not seem that passwords as authentication mechanism are falling in popularity. This together with the obvious limitations of the human memory, means that there is a need for strong ways of managing passwords.

1.1 Motivation

Blocki et al. [Blo14, BBD14] has proposed a scheme allowing human users to calculated passwords based on publicly available challenges and a secret mapping stored in their own memory. This allow users to protect all online accounts using long passwords while only remembering one set of mappings. The proposed scheme relies on generating random challenges for each account, which then have to be visible when users log in to the desired account. Blocki et al. propose ideas on how to make it easy for users to both memorize the mappings and do the calculations efficiently. Mnemonic techniques to ease the memorization process and a special layout make the calculations more intuitive.

The scheme is designed in such a way that the passwords are safe as long as the secret mappings stay secret. If one password is lost, all the others are unaffected, which

is one of the biggest strengths compared with other password management solutions. Security of the scheme solely relies on the secrecy of the mapping, meaning that the *challenges*, which are stored, can be lost without passwords being compromised. Passwords are thus *not stored* anywhere, but calculated in the mind of the users.

Usability is key for a password management scheme to even be considered by most users. If the requirements for it to function are too high, compared to the security rewards, no user will bother learning it. Usability is mostly related to the time spent calculating the passwords in addition to the failure rate when calculating and initial cost of memorizing a set of mappings.

This project investigates how the human computable passwords scheme can be used by real users. Since the scheme is quite complex, it can be useful to have an application taking care of all the required overhead, such as generation and management of challenges. Such an application is designed and implemented, with a goal of making it so that users, without too much introduction, can use it to manage passwords. The application is presented and the usability rewards discussed.

Even if the application makes the scheme easy and feasible to use, it might still be too demanding in regards to time spent calculating. The second part of this project investigates how efficient and reliably human users can calculate passwords. An experiment asking the participants to actually calculate passwords based on randomly generated challenges, is designed. Calculation time and failure rate of the trials are recorded through the experiment. The study is structured as an exploratory experiment, trying to find interesting and possibly important characteristics in the usage statistics recorded through the experiment application.

1.2 Scope and Objectives

The project first presents human computable passwords as constructed by Blocki [BBD14], while also describing other related background material. Throughout this project *human computable passwords* will be referred to as *HCP*, e.g. HCP application, HCP scheme. The objectives of the project are summarized as follows:

- Study the scheme with a special focus on the usability parameters, discussing how the different components of the scheme affect the usability.
- Design, implement and demonstrate an application realizing the password management scheme.
- Discuss if the usability is improved through the chosen design.
- Design, implement and conduct an experiment investigating the limitations imposed by calculation time and failure rates of average users.

- Conclude with a hypothesis about the practical limitations or advantages of the password management scheme.

No hypothesis will be stated in advanced, but data thought to be of intrest is collected. Afterwards, the data is presented and analyzed. No concrete result is given, but initial trends and characteristics are discussed.

Limitations. It is worth noting that the usability of the scheme is directly based on two things, the calculation performance(including the failure rate) and the effort spent memorizing and rehearsing the secret mapping. The experiment mainly investigates the calculation times and failure rates. Efforts related to the secret mappings are discussed, but memorizing a secret mapping is not part of the experiment trials. The reasoning for this decision is that it would be much harder to find participants willing to memorize a set of mappings without somehow compensating them for the efforts.

1.3 Outline

Chapter 2 presents relevant background material, mostly related to passwords and password management. Other methods for storing passwords are presented, showing the difference between password management software and password management schemes.

Chapter 3 describes HCP [BBD14], including definitions and human computable functions. Next, some new security features are presented, highlighting the relationship between the security parameters of the scheme, which allows users to adjust the scheme to their needs. A walk-through of the scheme showing how to set it up and how to calculate passwords is given in section 3.2.

The author notes that the chapter is partially reference work presenting the scheme of Blocki, but also new thoughts and descriptions (section 3.1.4, section 3.3) highlighting important features of the scheme.

Chapter 4 presents the HCP Chrome extension. First, the architecture and components of Chrome extensions are described, including important security features. Next, the extension design and implementation are presented, including the building blocks used to realize it. A short introduction to each of the components is given, before the actual implementation is described, with additional explanation of the code given in the appendix. Finally, a demonstration of the application is given, illustrating how it would be used in practice, highlighting the strengths of the application.

Chapter 5 describes the second part of the project, namely the usability experiment. The experiment is conducted through a separate application, which is presented, in addition to the experiment setup and results. This includes important choices and assumptions made trying to mimic the actual user experience when calculating passwords.

Chapter 6 contains concluding remarks summarizing the findings and experiences made throughout the project. Suggestions on further work is also given in this final chapter.

Chapter 2
Background

2.1 Passwords

Passwords are the common way of authenticating users upon access to sites on the Internet. The idea is that only users and the target service know the password, and the users have to provide the correct password before access is granted. Passwords are a much discussed theme and claiming that passwords are not always used in the correct manner is not an overreaction. The main problem seems to be that good passwords and the human memory does not go well together [YBAG00]. For passwords to be sufficient as authentication, users are forced into using long complex passwords, or even use one generated for them, with the problem being that it is easily forgotten. Furthermore, if users are able to memorize one "good" password, they will probably use this for all their accounts, if one of the services are compromised and user information leaked, all accounts may be compromised. With all of this in mind, it is easy to say that everybody should use complex, unique passwords for each account, but in practice this is not feasible. Florêncio and Herley [FH07] conducted a large scale study of password habits in 2007, revealing that a user on average has 25 different accounts protected by passwords. On average these sites were protected by about 7 distinct password, where 5 of them were rapidly re-used.

Password authentication requires the authenticating server to store something related to the password. If this is stolen the password will in most cases be compromised as well, even if the server did not store the clear text password. Attackers will, in most cases, be able to retrieve the password eventually. After obtaining the username and password for one service the attacker will try this user data on other services and compromise these as well.

Ives et al. [IWS04] discuss this "domino effect", where intrusion to one domain can compromise several others, if users have re-used passwords. They state in their conclusion that

> "Like dominos, when a weak system falls prey to hackers, information will be revealed that will aid the hackers in infiltrating other systems, potentially leading to the fall of many other systems, including systems with far better security than the first."

A normal users will typically try to log in by trial-and-error [ABK13], if the first password does not work, users will try with another password. This way passwords might be lost through phishing attacks where a user is tricked into trying to log in to a fake site.

2.1.1 Password Strength

How to measure the strength of passwords is a well known and discussed problem. The naive approach says that password strength is related to how strong a password is against brute force attacks [DMR09]. Length and complexity are the most thought of parameters to measure such strength. A perfect password would thus be one as long as allowed by the system, consisting of random characters, it would also be changed frequently. All these characteristics challenge how the human brain works.

In addition to the objective strength of the password, techniques making it harder for a computer to repeatedly try different passwords may be applied. Such techniques include CAPTACHAS [vABL04] which are puzzles supposed to require a human to be able to solve, making brute force attacks using a computer harder.

Yan et al. [YBAG00] investigate the trade-off between security of passwords and memorability allowing humans to remember them. An important point to this is that most sites apply advice and policies on how to create strong passwords, while not taking into account if the recommended passwords are hard or easy to remember. There is no point in having strong passwords if users are going to forget them. They suggest that passwords should appear random but be constructed using a mnemonic structure such as passphrases. The idea is to generate a random looking password by memorizing a familiar sentence and using the first letters of each word as the password. E.g using the familiar sentence "may the force be with you" as passphrase which would yield the password *mtfbwy* which looks random.

Florêncio et al. [FHC07] investigate another matter; do strong passwords accomplish anything? The point is that no matter how long and complex passwords users choose, they are still subject to the most dangerous and common threats (phishing, keylogging and access attacks). Especially access attacks, which includes shoulder surfing and direct access to a machine where an autofilling password manager is used, are unaffected by the strength of the password. The reason for enforcing strong passwords seems to be to protect against bulk guessing attacks.

Character set \ Password length	4	8	12	16
10	10^4	10^8	10^{12}	10^{16}
26	$5 \cdot 10^5$	$2 \cdot 10^{11}$	10^{17}	$4 \cdot 10^{22}$
36	$2 \cdot 10^6$	$3 \cdot 10^{12}$	$5 \cdot 10^{18}$	$8 \cdot 10^{24}$
62	10^7	$2 \cdot 10^{14}$	$3 \cdot 10^{21}$	$5 \cdot 10^{28}$
95	$8 \cdot 10^7$	$7 \cdot 10^{15}$	$5 \cdot 10^{23}$	$4 \cdot 10^{31}$

Table 2.1: Table showing the possible keyspace of passwords, given length and character set [SS09].

Table 2.1 illustrates the effects of password length and character set. The values represents the possible values a key might have, given the length l and character set size n. E.g. using only digits ($n = 10$) and length $l = 4$ gives 10^4 possible combinations. The bigger the key space is the harder it is to perform a brute force attack. Note that the key space increases more rapidly with increased password length, than by increasing complexity of the characters used. Keyspace is given by n^l, which grows exponentially when n is kept constant, e.g. 10^l, while it grows polynomially with n constant, e.g. n^{10}. If the character set is increased from 10 to 95 the key space increase 800 times given length of 4 characters, while increasing the length from 4 to 16 increase the key space one billion times with a character set of 10. Even if both character set and password length contribute to the strength of a password, length is the dominant factor.

The main point to take from this is that no matter how strong passwords users choose, they are still vulnerable. It is more important to limit the consequences of a possible password breach, by never re-using passwords on several accounts. This way, if an attacker manage to steal one password all the other accounts are still safe.

Password strength meters are a common way used by many sites to aid their users when selecting passwords. Common meters use colored progression bars together with a word or short comment stating if the entered password is evaluated as “bad”, “poor” or “strong”. Ur et al. [UKK+12] found that password meters actually lead users into choosing longer and stronger passwords, but they also argue that enforcing such policies might frustrate users and possibly lead them into writing passwords down, use weak password management schemes or re-use passwords. The most common requirements used by passwords meters of known web services can be summarized as the following [dM14]:

- Length and character selection are part of most password meters. It is normal to disallow passwords shorter, and sometimes longer, than a given range. A

variation of different characters can be required, namely different kinds of symbols and capital letters. Spaces may be allowed in between other characters, at the start or end of the password, or not at all. Some sites check for sequences of the same character as well.

- Personal information. Information registered by the users are evaluated by some meters, typically name, email and date of birth are checked against in original and transformed forms. This means that a password like "4nD3r?1991" ("anders1991" transformed) which look strong, will be evaluated as weak.

- Dictionary checks makes sure that the passwords does not include any dictionary words by matching it with a dictionary of common words.

The conclusion on what "good" passwords are, is not clear, but the one thing agreed upon seems to be that re-use of passwords is the biggest threat. It is a fact that the human brain is not capable of remembering different passwords for each account on the Internet, thus the need for an aiding application such as the one evaluated in this project. If users are able to at least have different passwords for each account, the consequences of a password breach is severely limited. The loss of one account will not compromise any other accounts.

2.1.2 Password Storage

Internet services using passwords as authentication method typically have to store information about their users, associating a username and a password. How this information is kept secure can not be controlled by the users, but is important for passwords to work, especially since many users tend to re-use their passwords on several accounts [IWS04]. It is a fact that these kinds of breaches do happen from time to time, and attackers might even post the credentials online[1]. If the service stores the passwords in plain text, leakage might directly lose passwords if the database gets lost.

Most online accounts try to make it harder for attackers to compromise the accounts of their users. The most used technique is hashing of the passwords before storing them, making it hard for attackers to recover passwords even if the database is broken. Hashing means applying a one-way function on the plain text password, producing a fixed length hash value which is stored instead of the plain text. When users want to log in the password is sent to the server where it is hashed again using the same one-way function and then compared to the stored hash value. If the database where these hashes are stored is compromised by adversaries, they can

[1]Hackers post millions of stolen Gmail passwords - http://www.cbsnews.com/news/russian-hackers-steal-5-million-gmail-passwords/

no longer directly use the data to steal accounts. The attackers would have to first perform an offline attack, cracking (see section 2.1.3) the hashes to recover the plain text passwords, which requires a lot of effort and time to carry out.

The problem with only hashing the passwords is that the hash of a given password will be the same every time, making it possible to recognize known password hashes through the use of big tables containing the hashes of known passwords. This way a hash might get cracked quite easily. To circumvent this problem a service might add a random *salt* to the password before hashing it. This way the hash of users' passwords will be unique even for users with the same password. A salt is a randomly generated nonce, which is concatenated with the password before hashing, and then stored alongside the hash value in the user database. To authenticate users, the server concatenates the entered password with the salt from the database entry of the individual users, then apply the hash function and compare with the stored hash value.

Example 2.1.
Let the following be the records of a user database, and H a cryptographically strong hash function.

Username	Hash	Salt
Bob	3aaF4A4PxZ	wRaE3Z9oa6
...	...	...

To authenticate, Bob would enter his password, $pwd =$ letno1in , which is sent to the server. The server then concatenates pwd with the salt from Bob's entry in the database, and then applies the hash function to the result. If this is the same as the stored hash, Bob is authenticated.

If $H(letno1inwRaE3Z9oa6) = 3aaF4A4PxZ \rightarrow$ Bob authenticated.

The hash function can be made even stronger by using a keyed hash function [BSNP96] or simply encrypting the hashes, requiring a shared secret key between users and the server. As a user, one can not control how different services store passwords, which is another important argument for never re-using passwords. Even if most sites store passwords securely, one badly configured site might compromise all the others.

2.1.3 Attacks

Passwords are often the only barrier stopping adversaries from directly accessing the accounts of users. The combination of user name and password are the easiest

point of attack, and thus the most valuable to break. There are several methods used to attack password authentication, trying to retrieve passwords. The most important attacks [SS09, FHC07], including access attacks and offline/online attacks, are discussed in the following section.

Capturing of passwords directly from the server responsible for the authentication involves the attacker acquiring password data through breaking into the data storage, eavesdropping on communication channels or through monitoring users by other means. The first, most basic threat is to simply steal the stored password from an insecure server, this would require a weak configured server storing the passwords in plain text. This is mitigated by storing only cryptographic hashes of passwords, which allows the server to authenticate users while preventing attackers from determining the actual passwords without *cracking* the hashes. Capturing can also be done through direct access attacks [FHV14] such as shoulder surfing or console access on computer with autofilling password managers or "remember me"-functions.

Online attacks are attacks directed towards the server's public user interface. Such attacks can be mounted simply by guessing username/password-pairs and sending authentication requests to the server, usually automated by scripts. The approach when carrying out such attacks are either breadth-first or depth-first. The former involves trying one password on all accounts first, before trying a new password. This technique would generate unusually high data load which would trigger alarms on the server.

Depth-first involves trying a lot of different password already knowing the username to attack. This would usually be blocked by lockout policies limiting the number of attempt allowed in a given interval.

Offline attacks are directed towards the backend of sites. To carry out such attacks, one would have to first access the system, gaining access to the password file without being detected. If detected while gaining access to the passwords, an system administrator might be able to force password resets by all the users, greatly limiting the time an attacker has to *crack* the passwords in the file.

Cracking requires the attacker to go through several steps. First acquiring the hash of a user account or a whole file of hashes for a site. Next, one would try to find a sequence of strings yielding the same hash as the actual password. Cracking is much harder if the server uses salts to randomize the hashes.

Rainbow tables [Bro11] is a technique employed by attackers to speed up this process. Rainbow tables are precomputed table of hashes, allowing the attacker to compute a set of hashes once and use these values several times, thus providing a

space-time trade-off. Using more space, since all the computed hash values would have to be stored somewhere, but allowing a much shorter computation time to brute-force a hash. The technique stores chains of hashes as shown in figure 2.1, storing only the first and last value of the chain. The attacker then searches for a given hash in the set of end points, if no match is found the hash function is applied and a new search conducted. This process continues, until a match is found, the plain text is then computed from the start value of the chain, applying the hash function the same amount of times it took to find a match in the chain.

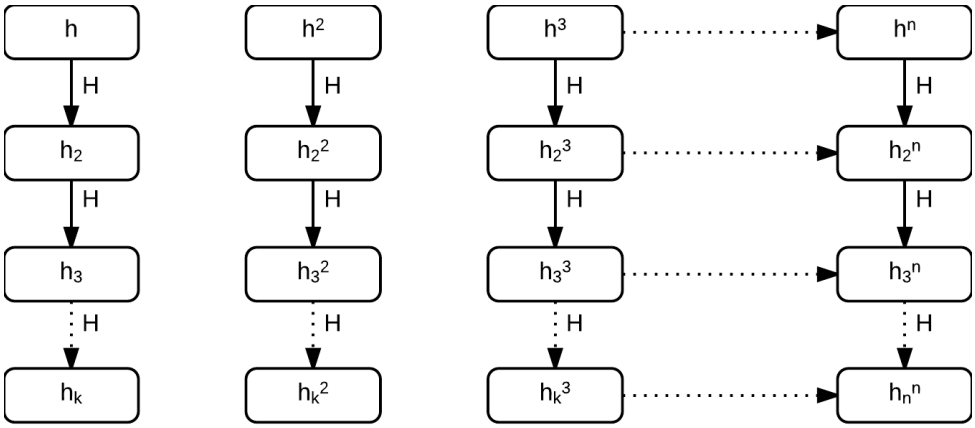


Figure 2.1: Rainbow table.

2.1.4 Offline-online Gap and Classifying of Accounts.

Florêncio et al. [FHV14] show that a password has to be able to withstand at least 10^6 guesses to be safe against online attacks. To survive offline attacks at least 10^{14} would be necessary. The point highlighted is that a password able to withstand 10^{10} guesses, is no more likely to survive an offline attack than a password only protected against 10^6 guesses.

They suggest that users should classify all accounts into categories ranging from "Don't-care" to "Ultra-sensitive". The idea is that there is no point in using complex and long passwords on an account which doesn't contain any sensitive information. For critical accounts related to finance, primary employment or other highly confidential documents it is advised to use multiple factors of authentication [AMV12]. Accounts used for e.g. social media or streaming would probably be categorized as "medium-consequence" as loss of such accounts would be more about time and effort, than financial loss. It is of course up to the users to categorize how critical it would be to lose each account (see [FHV14] for an example category division).

The important point is that accounts should be treated differently. There is no point in using a 20 characters long password on an online chess game account, while it would be reasonable on an online banking account. This can be utilized to make password management more adaptable and easy to use. Especially when using a password management scheme, as the one evaluated in this project, it is very beneficial to lower the average length of passwords, while still having long passwords for the important accounts. Categorizing of accounts will be utilized to improve the usability of the scheme implemented later.

2.2 Password Management

As seen in the previous sections passwords introduce a dilemma as passwords are supposed to be hard to "guess" and are thus hard to remember. The naive solution to this problem is to either use one password for many accounts or to write down passwords. To make the process of managing passwords easier, several techniques and tools have been suggested [GF06]. Examples are reminder features including the "I forgot my password"-function most websites implement, browser cookies allowing users to stay logged in across browser sessions or services storing the passwords. These are all "computer aided" tools, which will store or keep users logged in without actually having to remember the passwords. This usually limits the users to staying on the same machine while using the account, this way users will probably forget passwords and be forced into using the "forgot my password"-function eventually. A different approach is to use a technique to actually remember the passwords without storing them.

Consider a *password management scheme* to be a method helping users remember password without actually storing them. The term *password management software* or *password manger*s are used about solutions which store the passwords in some way.

2.2.1 Password Management Schemes

Password creation and memorization techniques assist users in remembering passwords, trying to circumvent the limitations of the human memory [Bad97].

Blocki et al. [BBD13] consider 4 different examples of password managements schemes to illustrate how users might choose and remember their passwords.

- Reuse Weak. When users select a random phrase or word w and reuse this as the password $p_i = w$ for all accounts. While maybe not very strong, this is the most simple example of a password management scheme.

- Reuse Strong. Same as reuse weak but users chose four random words w_1, w_2, w_3, w_4 and reuse the concatenation of these as the password $p_i = w_1w_2w_3w_4$ for all his accounts.
- Lifehacker. Users chose three random words w_1, w_2, w_3 as a base password $b = w_1w_2w_3$ as well as a derivation rule d used to derive unique data from the site names for each password [2]. Example of a derivation rule could be the first and last three letters of the service name. The password for account A_i would then be $p_i = bd(A_i)$ with $d(A_i)$ being the string derived from the site name. In practice a password generated using the method might look like "facthreerandomwordsook".
- Strong Random and Independent. Users chose new words $w_1^i, w_2^i, w_3^i, w_4^i$ for each account to be used as passwords $p_i = w_1^i w_2^i w_3^i w_4^i$.

It is clear that the three former schemes are much easier to use than the last one. Most users would prefer the first ones because they do not require much, if any, rehearsal while the one strong scheme would require too much effort in terms of rehearsing and memorization. This trade-off between usability and security is the main problem when designing password management schemes. For a scheme to be popular it cannot require too much extra rehearsal, while a secure scheme most of the time will require some.

2.2.2 Password Manager Software.

Password managers are applications meant to keep passwords safe the for users. These applications can either be stand alone programs or, more common, browser extensions such as LastPass [3]. LastPass provide an user interface to generate and store passwords for online services, as well as form fillers to enter them when logging in. The passwords are encrypted using a master password protecting the user credentials against both server leakage and insiders accessing the data. Such systems usually provide a lot of extra features such as automatically changing of passwords and syncing between devices.

Built-in browser password managers. Most modern browsers provide “remember password” functions. These functions act similar to software like LastPass, by storing the users’ passwords in some fashion, then reproduce it when logging in.

These kinds of systems and applications require the users to trust that the implemented systems are secure enough to prevent adversaries, both insiders and outsiders,

[2] How to Update Your Insecure Passwords and Make Them Easy to Use - http://lifehacker.com/5631203/how-to-update-your-insecure-passwords-and-make-them-easy-to-use

[3] LastPass - https://lastpass.com/

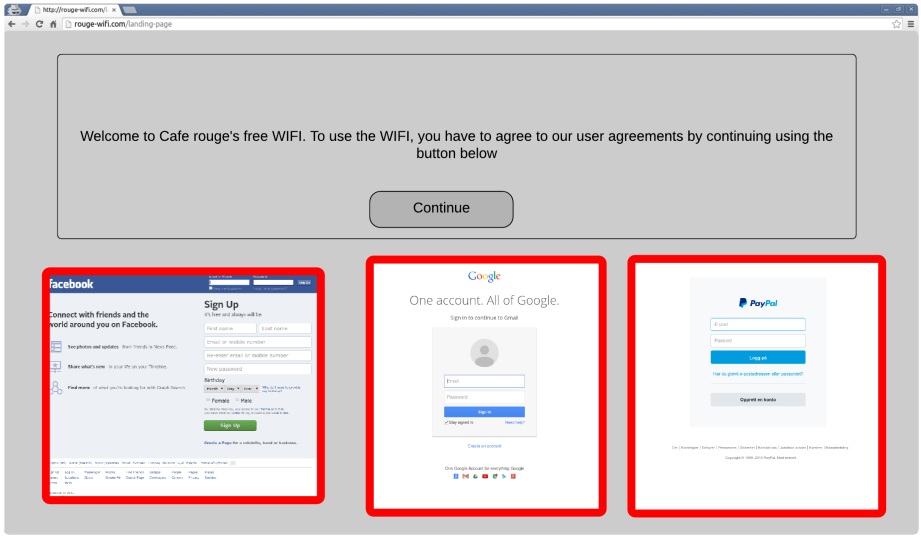


Figure 2.2: Rouge wifi landing page containing iframes with common sites, used to steal password from an autofilling password manager.

from stealing credentials. Software aiding the users by storing passwords often also provide autofill-functions, automatically filling in the username and password of the associated site. Silver et al. [SJB+14] propose attacks exploiting these autofill functions to extract passwords from the password manager, the most basic example attack is shown in example 2.2. Despite the obvious weaknesses in many password managers, they still argue that password managers can strengthen credential security if implemented correctly.

Example 2.2.
Consider users connecting to a open wifi at a coffee bar or another public place. It is not unusual to present the users with a "landing page" asking for approval of some usage agreement. The rogue wifi provider could include multiple iframes[4] pointing to the login pages of common sites the users probably have stored credentials for. See figure 2.2. By injecting Javascript in the iframes the attacker can extract all the usernames and passwords autofilled by the password manager. Note that the iframes displayed in the figure are not visible to the users, to them it looks like a standard landing/welcome page. Silver et al. [SJB+14] claim that six out of ten password managers were vulnerable to this simple attack.

[4]Iframe-http://www.w3schools.com/tags/tag_iframe.asp

Even if the password manager is secure against autofill-attacks, or if it does not include the feature, the manager might still be at risk. If the account storing all the usernames and passwords where to be compromised, all the sites would be compromised, so it is important that the password manager is even more secure than the sites themselves.

Zhao et al. [ZYS13] identified several vulnerabilities in the LastPass implementation, even though no known breaches has been reported. They investigate different types of attacks, including attacks on local decrypted credentials; request monitoring attacks which tries to intercept request between the password manager and related cloud-storage; as well as brute-force attacks trying to crack the master password. The conclusion is that password managers are double-edged swords, in theory they help make password authentication stronger, but if implemented slightly wrong may be a major vulnerability. Storing all the passwords at one place makes an obvious point of attack for adversaries since breaking the manager most of the time would break all accounts stored within.

Chapter 3
Human Computable Passwords

The previous chapter concludes that managing passwords for online accounts is a major issue for modern Internet users. It seems to be impossible to remember and maintain enough strong passwords to keep all accounts secured. The scheme presented in this chapter is designed to help users maintain and remember multiple strong passwords, while also protecting these after multiple password breaches. HCP takes advantage of the human brain allowing users to calculate passwords from public challenges, using their own mind to do so.

The HCP scheme is proposed by Blocki et al. [BBD14]. In addition to the scheme itself, the proposal introduces security and usability notions used to analyze the proposed scheme. This chapter describes the scheme as well as associated security and usability concerns. The first section consists of definitions and notations as used by Blocki [BBD14] to describe the scheme. Next, human computable functions are introduced as this is the main component used in the password management scheme. How these functions can be used to generate and memorize unique passwords in practical cases is presented. Finally, usability concerns related to the scheme are reviewed.

3.1 Password Management Scheme

The main idea of the HCP scheme is to have a set of challenges stored in persistent memory, typically on a computer or even a piece of paper. Users then use a mapping and a function to calculate the response to each challenge, which eventually gives the password. It is worth emphasizing that this is different from other "traditional" password managers, in that the *passwords are not stored*, only challenges helping users remember passwords.

To create a new password, random challenges are generated, users then compute the passwords from these. To reproduce the password later, the same challenges are

displayed to the users, who then can calculate the same password. This procedure is explained further in section 3.2, and in algorithms 3.2 and 3.3 .

3.1.1 Definitions and Notation

Memory types considered are either *persistent* or *associative* memory [Bad97]. This project follows the settings of Blocki et al. [BBD13, BBD14] where persistent memory is equal to writing something down or somehow storing it reliably, but not securely. When talking about persistent memory, it can be assumed that this is publicly available, or at least that an adversary has undisclosed access to the data. This should be emphasized since this is a strength to the scheme, nothing, except the secret mapping, needs to be kept secret after establishing the needed prerequisites.

Associative memory is the memory of the users, namely their human memory. This memory is different from the persistent memory in that it is totally private but needs to be rehearsed to not lose data. In a password management scheme rehearsing should optimally be part of the natural activity of the users. The best case would be if users could rehearse and keep all their passwords in associative memory by simply logging in to their accounts as normal. This is a central challenge for all password schemes [BBD13].

The password management scheme uses a random mapping between a set of objects to single digits which has to be memorized by the users. This mapping is denoted as $\sigma : [n] \to \mathbb{Z}_d$. If $X_k \subseteq [n]^k$ is the space of ordered clauses of k variables, let C be a clause chosen at random from X_k. C is now a set of k objects (e.g. $(2, 4, 7, 8)$). Now $\sigma(C) \in \mathbb{Z}_d^k$ is the mapped variables corresponding to challenge C. C can consist of any type of object, such as pictures letters or digits, with the mapping σ always being to digits.

Example 3.1.
If $\sigma(x) = x + 1 \bmod 10$, and $C = (10, 25, 36)$, then $\sigma(C) = (1, 6, 7)$.

One of these challenges, C, is referred to as a *single digit challenge*, which will consist of k ordered objects chosen at random. The function $f : \mathbb{Z}_d^k \to \mathbb{Z}_d$ is a human computable function as discussed in the next section. Users respond to a challenge C by computing $f(\sigma(C))$. A complete password challenge, $\vec{C} = (C_1, \ldots, C_t) \in (X_k)^t$, will consist of t separate, single digit challenges. The response to $\vec{C}$, namely $f(\sigma(\vec{C}))$, is the complete password.

The password management scheme works by generating one challenge, $\vec{C}$, for each user's accounts $A_1, \ldots, A_m$. The challenges $\vec{C}_1, \ldots, \vec{C}_m \in (X_k)^t$ are stored in persistent memory. When users want to log in to a service they are shown the

challenge $\vec{C}_i$ corresponding to account A_i, they then calculate the responses to all the single digit challenges, producing the password.

3.1.2 Human Computable Functions

At the core of the scheme is a human computable function f and the memorized mapping σ. The scheme requires the composite function of these two, $f \circ \sigma$, to be *human computable*, which means that the function should be easily computable by the users without aids. To fulfill this requirement the function cannot involve many operations, since the complexity and thus computation time would be too high. As shown by Miller [Mil56], a human can only store 7 ± 2 pieces of information at a given time. On the other hand humans are quite good at simple operations such as addition modulo 10. For example "1+6+5+3+8+9+3+1+4+6+7+7+6 mod 10" would be easy for most humans to compute by simply doing one operation at a time, updating the answer after each addition. With this approach only one piece of information is stored in memory of the users at any time. The problem with such an expression is the amount of terms.

The requirements needed for a function to be human computable can thus be summarized as the following, and formalized in requirement 3.1:

- Can only involve "simple" operations, mainly addition and recalling from long-term memory.
- Limited amount of terms.
- Limited amount of operations.

Remark 3.1.
All operations used in the human computable functions discussed in this project are modulo 10, as this is the most natural for most humans.

Requirement 3.1.
Function f is said to be $\hat{t}$-human computable if a human can compute it in his head in $\hat{t}$ seconds.

Blocki et al. [BBD14] believe that a function f is human-computable if it can be computed using a fast streaming algorithm, meaning that the input is presented as a sequence of objects that only can be evaluated once. The algorithm would have to be simple since humans are not good at storing intermediate values [Mil56]. Typical operations fast enough for the human to compute in his head is addition modulo 10 which is natural for most humans to do quickly, and recalling a mapped value $\sigma(i)$ from memory.

For primitive operations in P, the following are considered:

- *Add* takes two digits x_1 and x_2, and returns the sum $x_1 + x_2$ mod 10.
- *Recall* returns the secret value $\sigma(i)$ corresponding to an input index i. The mapping σ is memorized by the users, allowing the recall operation to be done quickly in the users' head.
- *TableLookup* involves looking up the x'th value from a table of 10 indices.

Definition 3.1.
A function f is $(P, \tilde{t}, \tilde{m})$-computable if there is a space $\tilde{m}$ streaming algorithm computing f using $\tilde{t}$ operations from P.

Remark 3.2.
Space $\tilde{m}$ means that the algorithm requires no more than $\tilde{m}$ memory slots during calculation. Slots are typically used for storing values and executing primitive operations such as addition [AMS99].

Example 3.2.
The function $f \circ \sigma(i_1, \ldots, i_5) = \sigma(i_1) + \cdots + \sigma(i_5)$ is $(P, 9, 3)$-computable, since it requires 9 operations from P, 5 recall operations and 4 add operations. $\tilde{m} = 3$ since a sequence of additions $i_1 + \cdots + i_n$, requires one slot for storing the sum, one slot for storing the next value in the sequence and one slot to execute the addition.

Blocki et al. [BBD14] conjecture that a user H will be able to calculate a $(P, \tilde{t}, 3)$-computable function in $\hat{t} = \gamma_H \tilde{t}$ seconds. They believe that users with moderate mathematical background should be able to achieve results yielding $\gamma_H \leq 1$. This conjecture is part of the experiment presented later in this project.

Conjecture 3.1. *[BBD14] For each user H there is a small constant $\gamma_H > 0$ such that any $(P, \tilde{t}, 3)$-computable function f is $\hat{t}$-human computable with $\hat{t} = \gamma_H \tilde{t}$.*

3.1.3 Secure Human Computable Functions

Blocki et al. [BBD14] suggest a family of human computable functions defined as follows.

$$f_{k_1,k_2}(x_0, \ldots, x_{9+k_1+k_2}) = x_j + \sum_{i=10+k_1}^{9+k_1+k_2} x_i \mod 10,$$

$$\text{with } j = \sum_{i=10}^{9+k_1} x_i \mod 10 \quad \text{and} \quad k_1 > 0,\ k_2 > 0$$

Definition 3.2.
$f(x_0, x_2, \ldots, x_{13}) = \left(x_{((x_{11}+x_{10}) \mod 10)} + x_{12} + x_{13}\right) \mod 10$

Definition 3.3.
$f \circ \sigma(x_0, x_2, \ldots, x_{13}) = \big(\sigma(x_{(\sigma(x_{11})+\sigma(x_{10}) \mod 10)}) + \sigma(x_{12}) + \sigma(x_{13})\big) \mod 10$

This project uses one of these functions, with $k_1 = k_2 = 2$. From now on this will be the function referred to as f, the function is defined in definition 3.2. For an in depth analysis of the function see "Usable Human Authentication: A Quantitative Treatment" [Blo14].

In addition to f, a mapping function σ is used. Definition 3.3 defines the composite function of f and σ which is used later in the password scheme. The response to a challenge $\vec{C}$ is calculated using this function $f \circ \sigma$. Figure 3.1 and table 3.1 summarize how the system works, random challenges are stored persistently in a database and a secret mapping is stored in the associative memory of the users. A challenge is then converted into a password by applying the mapping and a human computable function to each single digit challenge. The combined results of these calculations yield the complete password of the site.

Blocki argues that an adversary would have to see $\tilde{\Omega}(n^{1.5})$ challenge-response pairs to be able to start recovering the secret mapping σ. A realistic mapping σ would probably consist of no more than 100 object to digit mappings. A secret mapping consisting of $n = 100$ mappings would require an attacker to steal 1000 challenge-response pairs (100 accounts given password length of 10) to recover the secret mapping. In practice this might be the tricky part of the scheme, memorizing a mapping of 100 object-digit mappings might be possible, but probably too hard for a “normal” user to bother doing. It might be more reasonable to use a smaller set of mappings which will lower the security of the scheme, while making it more accessible for novice users.

Remark 3.3.
The analysis from hereon will use the observation from Blocki’s thesis [Blo14], claiming that an adversary needs to see $\tilde{\Omega}(n^{1.5})$ challenge-response pairs before starting to recover the secret mapping.

An example mapping which could be feasible in practice is characters to single digits, with characters from the alphabet and digits between 1 and 10. This mapping would yield $n = 26$ which would require an attacker to recover significantly less challenge-response pairs. With $n = 26$ the amount is down to 133 compared to the 1000 with $n = 100$. Still, this would require to fully compromise 13 or more accounts with password lengths of 10 characters. How many objects n in the mapping function, should be decided after evaluating how many accounts, and how sensitive the information associated are.

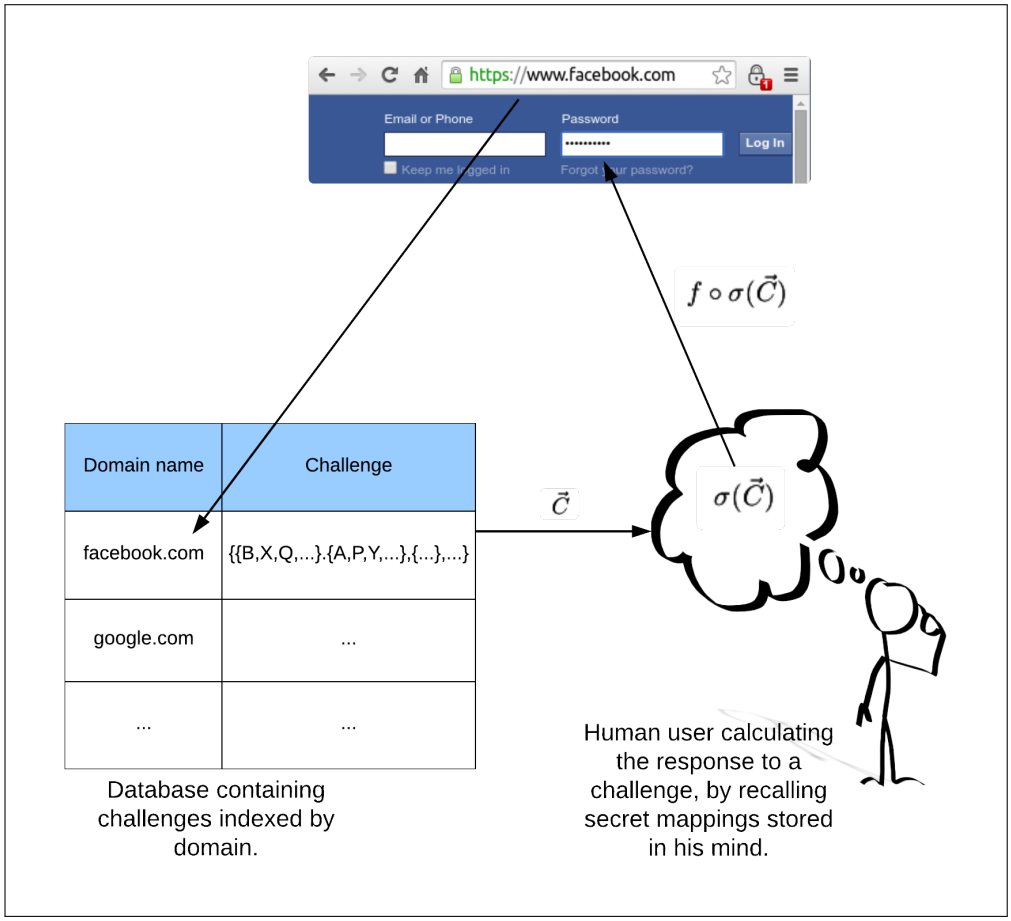


Figure 3.1: The database contains challenges indexed by domain name, which is fetched and displayed to the users. User can then calculate the response to this challenge by using the secret mapping σ which is only stored in their minds.

3.1.4 System Parameters

The previous section has described how the password management scheme works. The most important characteristic that should be emphasized is that there is two possible sources of information leakage, namely the database used to store the challenges and the different site's password storage. The latter is e.g. the database of facebook.com which will consist of salted hashes of the users' passwords. The challenges are, as mentioned, stored in persistent memory, which is assumed to be available to an adversary, so leaking the challenge alone is not necessarily a problem. If password hashes are leaked from online services the password to that exact service might be lost, but the scheme still ensures that no other passwords can be deduced from the

$f(x_0, \ldots, x_{13})$	A human computable function used in the password management scheme investigated in this project. Defined in definition 3.2
$\sigma(x)$	Random mapping to be memorized by the users. It takes in an object of some sort and returns a digit between 0 and d, d can be assumed to always be 10.
C	A single digit challenge, this is a challenge consisting of 13 randomly chosen ordered objects. A single digits challenge in this project is a list of 13 random letters (e.g. ("B", "E", ...)). One of these is used to compute *one* character of a user's password.
$\vec{C}$	A password challenge, consisting of t single digit challenges. A password challenge yields a complete password after calculating the response to all single digits challenges contained in it.

Table 3.1: Summary of notation.

compromised password. An adversary attacking the scheme would try to recover the secret mapping σ since this would allow him to compute all the passwords of a user. Such an attack, if possible, would require a set amount of challenge-response pairs (namely $n^{1.5}$), as discussed in section 3.1.3 (see remark 3.3).

There are some interesting trade-offs related to the parameters of the HCP scheme. A bigger set of mappings makes it increasingly hard to recover σ, but it becomes harder to memorize and rehearse as well. It is reasonable to say that complexity of a mapping function grows linearly with the number of mappings n, and the resistance versus attackers grows polynomially, thus much quicker than the complexity, see Figure 3.2. In other words, for each mapping added to σ, n is increased with one and the security multiplied with 1.5. The trade-off which would have to be evaluated is how much effort the users are willing to put into memorizing the mappings, versus how secure they want it to be. This should be evaluated in regards to how "important" the passwords and the accompanying accounts are, and how many accounts users plan on having. It is not worth memorizing a large set of mappings only to store a few passwords, since there would not be enough mappings to "lose" for an adversary to recover even a small mapping set.

Another relation is between password length and number of accounts which would have to be stolen. With n mappings, the number of accounts needed to start recovering σ is a function of the password length as seen in Figure 3.3. If the

passwords are very long only a few logins would have to be stolen to recover σ. This is important to take note of since one of the main strengths of the password scheme is that even if one account is compromised all the others are still secure since each site has a different, "unrelated" password. If the revelation of only a few accounts could compromise the secret mapping, all the passwords of the users might be lost.

Users requiring very secure passwords might generate very long passwords of 20+ characters for each of their accounts. If, by chance, the number of mappings was smaller than suggested, all these "strong" passwords might be lost if only a few of them was to be compromised through a password breach. Users are not advised to use short passwords, but the secret mapping needs to be long enough to support the length and number of passwords a user wants to generate using the scheme.

The number of accounts needed to recover the mapping can be used as a practical way of describing the security of the scheme, observation 3.1 defines this parameter as an inequality reliant on the password lengths x and the number of mappings n. Figure 3.4 illustrates the relationship between password lengths and number of mappings needed to achieve different levels of security. How high the parameter $\hat{a}$ should be depends on how long and how many passwords users intend to have.

Observation 3.1.
The security of human computable function including a mapping function, $f \circ \sigma$, as defined in definition 3.3 and section 3.1.2, can be described through the expected number of accounts $\hat{a}$ which needs to be compromised to start recover the secret mapping σ, given passwords of length x and n mappings in σ. $\hat{a}$ is then

$$\hat{a} < \frac{n^{1.5}}{x} \tag{3.1}$$

.

Example 3.3.
A user plans on having passwords of length 20 for all of his many important accounts, and wants these to be securely stored even if it requires him to use more time on rehearsal. In this case, assume that a user wants his accounts be secure even if 100 accounts are leaked. Using Equation 3.1 with $x = 20$ and $\hat{a} = 100$, gives $100 < \frac{n^{1.5}}{20} \implies n > 159$. This means that the user would have to memorize at least 159 unique random mappings to achieve the desired level of security against leakages.

If users were to save only a few shorter passwords, for example requiring only security allowing loss of $\hat{a} = 20$ accounts and passwords of length $x = 15$, they would need to memorize at least $n = 45$ mappings. Users would have to evaluate how many mappings are realistic to memorize, and decide on a reasonable level of security.

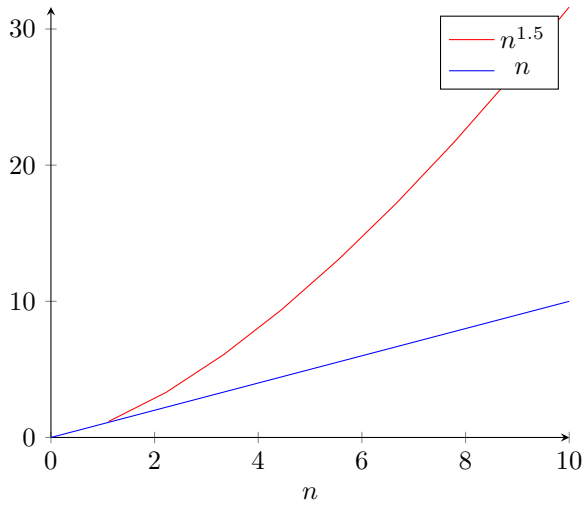


Figure 3.2: Number of challenge-response pairs required to start recovering mapping σ as a function of the number of mappings in the mapping n.

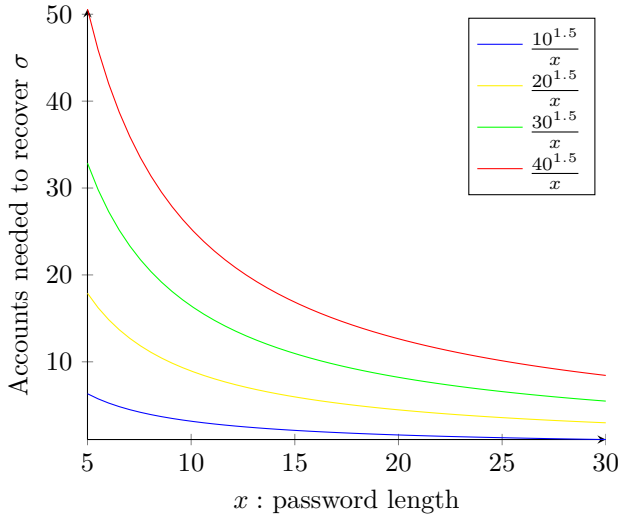


Figure 3.3: Number of accounts needed to start recovering the mapping σ, given different values of n.

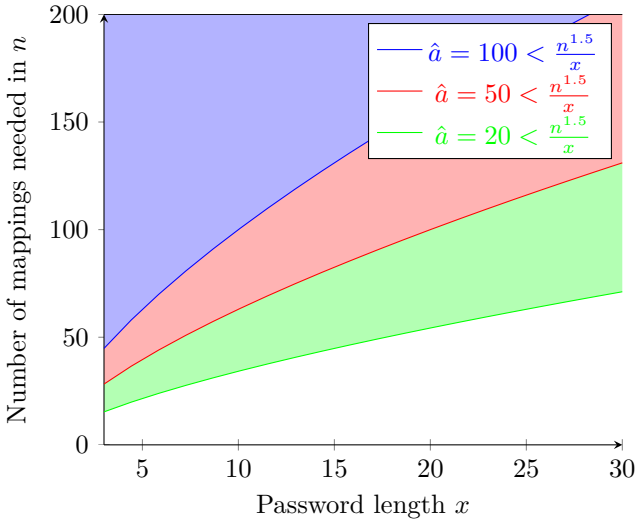


Figure 3.4: Inequality plot of 3 different values for $\hat{a}$

3.2 Practical Usage

This section illustrates how the HCP scheme works in practice, using the principles described in the previous sections. Before using the scheme a user has to go through a setup procedure involving memorization of a randomly generated mapping as well as setting up all accounts with passwords calculated from challenges. Section 3.1.4 discussed how the scheme can be tweaked to fit different needs a user may have, depending on the required level of security and the amount of accounts and passwords a user may have. A summary of the setup and authentication procedures are presented next.

3.2.1 Setup Procedure.

1. A secret mapping is the first prerequisite required before the scheme can be used. A random mapping of length n is generated from a set of objects chosen by the users, to digits in $\{0, 1, \ldots, d\}$. Users will typically choose what type of objects to use, this can be an alphabet or a user chosen set of pictures. A system will then choose n of these objects and assign a random digit between 0 and d, d is normally 10. Algorithm 3.1 shows how the mapping would be generated from a set of objects by assigning random digits to each object. Remember that this mapping function is supposed to only be stored in the memory of the users, and can thus not be evaluated anywhere else than in the mind of each individual user.

2. Memorization of the mapping is the next, and most costly procedure. Users have to learn the mappings by heart, and be comfortable they will not forget them. After memorizing, the mapping is deleted and not stored anywhere else than in the mind of the users. After finishing this step, there is no way to recover the mapping if users forget it. This might seem like a barrier, but the fact is that the memorization is a one time cost for many users. Active users will naturally rehearse and thus not forget their mappings as long as they keep using the scheme and regularly calculate passwords.

3. Passwords can now be generated for all the users' accounts. Algorithm 3.2 describes the process of creating new passwords for all accounts.

   a) First users choose the desired length of the password, t, to be generated. In practice this length is not static for all accounts, when adding a new account to the system users will categorize the account, which will decide the length of the password, and thus the number of challenges generated.

   b) t random challenges are generated and shown one by one to the users, which calculate the responses to these. Each of the responses are one character in the new password.

   c) The calculated password is then sent to the server typically through a "change password"-form.

4. the same procedure (1-3) can be done for all the accounts users want to include in the password scheme.

Algorithm 3.1 Generate mapping σ.

Input:

- Base d.
- $O_1, \ldots, O_n$ objects, typically letters or pictures.

1: **for** $i = 1 \rightarrow n$ **do**
2: $k \in_R [0, d]$ ▷ k is a random integer between 0 and d
3: $\vec{S} \leftarrow \vec{S} + (O_i, k)$

4: **function** $\sigma(O_x)$
5: search($O_x \quad in \quad \vec{S}$)
6: **return** (O_x, i)

Output: σ

3.2.2 Authentication Procedure.

1. Authenticating with a site, which password was previously generated using the scheme, start of by selecting the correct site. The corresponding challenges will then be displayed starting with the first one.

2. Users calculate the response to each challenge, the same way they did when generating the password. If the calculations are done correctly, the result should be the same.

3. After calculating the response to all t challenges, the password can be submitted to the server which checks if the hashed value is the same as the stored one. If it is, the user is authenticated.

Remark 3.4.
The notation $f(\sigma(\vec{C}))$, as used in the algorithms, is equal to the composite function $f \circ \sigma(x_0, \ldots, x_{13})$ as used in definition 3.3.

Algorithm 3.2 Create new challenge for account $A_j \in (A_1, \ldots, A_m)$

Input:

- t desired length of password.
- σ secret mapping memorized by the user.
- f a human computable function.
- $O_1, \ldots, O_n$ objects, typically letters or pictures.

1: **for** $i = 1 \to t$ **do**
2: $\quad k \in_R [0, n]$
3: $\quad \vec{C}_i \leftarrow \vec{C} + \{O_k\}^{14}$
4: $\vec{C} \leftarrow (\vec{C}_1, \ldots, \vec{C}_t)$
5: **(User)** Computes $(p_1, \ldots, p_t) = f(\sigma(\vec{C}))$
6: **(Server)** Store $h_j = H(p_1, \ldots, p_t)$
7:
8:

Output: $\vec{C}$

3.3 Usability

Blocki et al. [BBD14] consider three usability parameters defining the usability of a human computable function, namely *calculation time*, *memorizing* σ and *rehearsing* σ. In addition to these, another influencing factor is introduced, *failure rate*. This section discusses these requirements and how to influence them.

Algorithm 3.3 Authentication process for account $A_j \in (A_1, \ldots, A_m)$

Input:

- Account $A_j \in (A_1, \ldots, A_m)$
- Challenges $\vec{C} = (\vec{C_1}, \ldots, \vec{C_t})$ from account A_j.
- Hash h_j and hash function H.

1: **for** $i = 1 \rightarrow t$ **do**
2: Display C_i to the user
3: **User** Compute $p_i \leftarrow f(\sigma(C_i))$
4: ▷ p_i is the i'th character of the password for account A
5: $\vec{P} = (p_1, \ldots, p_t)$
6: **if** $h_j = H(\vec{P})$ **then** ▷ **(Server)**
7: Authenticated on account A_j
8: **else**
9: Authentication failed

- The effort required to memorize the secret mapping.
- The extra rehearsal required of the users to not forget the secret mapping.
- How long it takes a human user to calculate the responses to a set of challenges, eventually producing the password.
- How reliably a human user can calculate password without mistakes.

All of these requirements might limit the usability of the scheme, and are thus worth discussing. This project focuses mostly on the last two requirements, related to computation time and failure rates. These parameters are later tested (see chapter 5) through implementing the scheme as a web app and having participants try it out while timing their efforts.

3.3.1 Memorizing the Secret Mapping.

As seen in the previous section, users have to memorize a random mapping from object to digits before starting to use the scheme. This is most likely the biggest and most frightening barrier for any user considering to use the scheme. There are several techniques supposed to help memorizing relations easier, examples are the method of loci [Bad97] which is supposed to enhance memory by visualization. Mnemonic helpers showing objects merged together might help memorize relations as in the case of this project. Blocki et al. [BBD14] propose using mnemonic helpers if the mapping consist of letters to digits. These helpers would typically be a set of pictures showing a visual transition from a letter to a digit. This way might make it easier for users to remember it instead of only being shown "A=1" etc. Some user might also

feel that it is easier to memorize other things than letters, such as pictures. Users might even get to choose the set of pictures to be used themselves as long as the corresponding digits are chosen at random.

3.3.2 Rehearsing the Secret Mapping.

After memorizing the mappings, the users have to rehearse it frequently enough to not forget them. Blocki et al. [BBD13] define a model estimating the cost of this rehearsal. Applying this model to the password management scheme gives insight to how much different types of users have to rehearse. The model predicts how long a user will remember an association between i and corresponding mapping $\sigma(i)$ without further rehearsal.

Definition 3.4. [BBD14]
A **rehearsal schedule** for an object-mapping association is a series of points in time $t_0 < t_1 < \dots$. A rehearsal requirement says that all object-mapping association pairs must be rehearsed at least once in the time interval $[t_i, t_{i+1})$, to not forget them.

Requirement 3.2. [BBD13]
Constant Rehearsal Assumption (CR) says that a rehearsal schedule where users rehearse every i'th day, e.g. day 1, 2, 3 etc., is sufficient to remember the mappings.

Requirement 3.3. [BBD13]
Expanding Rehearsal Assumption (ER) assumes that users will become better at remembering the mapping for each consecutive rehearsal, thus only needing to rehearse every 2^i days. E.g. day 2, 4, 8, 16 etc.

The difference between these two assumptions about human memory is that CR assumes that users have to keep rehearsing every i'th day for as long as they want to make sure to not forget anything. This might be too pessimistic since it is reasonable to assume that it gets easier to rehearse for each rehearsal. This is what ER assumes, if a relation has been rehearsed i times it does not have to be rehearsed again in 2^i days. ER is the most intuitive assumption to make and is backed up by experiments on how the human brain forgets over time [Squ89, Bad97].

Visitation Schedule

Every user will eventually have a unique visitation schedule which will vary greatly from user to user. The model uses a Poisson process to model the visitations schedule for a given site A_i, with parameter λ_i. The average time between visits, $\frac{1}{\lambda_i}$, is assumed to be known for each visitation schedule. A site visited every day would yield $\lambda_i = 1$ day, and $\lambda_i = \frac{1}{365}$ days for a site visited annually.

Next, the model uses four different types of users which may have accounts of 5 different account types based on visitation frequency. The users can be: very active, typical, occasional or infrequent, while an account can be visited daily, every three days, every week, every month or annually. Table 3.2 defines how many of each type the users have respectively. For example, very active users are said to have 10 accounts they visit daily and 35 visited annually.

Visitation schedule λ_i	1	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{31}$	$\frac{1}{365}$
Very Active	10	10	10	10	35
Typical	5	10	10	10	40
Occasional	2	10	20	20	23
Infrequent	0	2	5	10	58

Table 3.2: Visitation schedules. λ_i is the average time between visits to an account.

Extra Rehearsals

Users will visit their accounts according to their visitation schedules, and every time the user will rehearse a set of object-mappings naturally by computing their passwords.If an object-mapping association is not rehearsed through normal usage users would have to rehearse the association to prevent forgetting it. Blocki et al. [BBD13] predict how much extra rehearsal, $E(ER_t)$, is required in addition to natural usage. In the context of a password management scheme it is clear that smaller values for $E[ER_t]$ yield less effort required by the users.

Table 3.3 shows the expected number of extra rehearsals required by the different types of user given the length of the mapping function n, during the first year. It is computed given the visitation schedules in table 3.2 and uses the expanding rehearsal assumption as defined in definition 3.3. Details on how the table was computed and proofs of the theorems used can be seen in appendix H of "Human Computable Passwords" [BBD14].

User	$n = 100$	$n = 50$	$n = 30$
Very Active	0.396	0.001	≈ 0
Typical	2.14	0.039	≈ 0
Occasional	2.50	0.053	≈ 0
Infrequent	70.7	22.3	6.1

Table 3.3: Extra rehearsals required of the users during the first year to remember σ [BBD14].

The results in table 3.3 clearly demonstrates that the HCP scheme does not require much rehearsal at all if used frequently. In fact, for very active, typical and occasional users, memorizing the mapping is a one time cost. After memorizing it at the beginning, using the scheme will provide enough natural rehearsal to maintain the mapping in memory. When users compute the response to a challenge $\vec{C}$, they will have to recall the mapping of up to 5 $((\sigma(x_{11}), \sigma(x_{10}), \sigma(x_{12}), \sigma(x_{13}), \sigma(x_j))$ values of i for each character of the password. A password length of 10 would yield recalling, and thus rehearsing, 50 values of i. The same trade-off as discussed in section 3.1.4 can be observed here. The more complex the mapping is (larger values of n), the more effort is required when memorizing, but no extra rehearsals are necessary even with a larger number of mappings.

3.3.3 Computation Time and Failure Rates.

The final requirement which may limit the usability of the scheme is calculation time. If users can not compute the response to a challenge correctly in a reasonably short amount of time the scheme would not be usable. How much time users can tolerate is of course individual, but a too long computation time will directly effect the usability. In addition to the time spent calculating, it is important that the users are able to consistently compute the correct responses. If the failure rate is too high, in respect to the password length, the scheme will not function at all. It is thus more important to have a low enough failure rate than a short calculation time.

Improving Usability.

To make the computation as easy as possible the challenges are presented to the users in a practical way, facilitating fast and reliable calculation. Using the human computable function from definition 3.3, a challenge $C = (x_0, x_1, \ldots, x_{13})$ could be displayed as shown in Table 3.4.

$$f \circ \sigma(x_0, x_2, \ldots, x_{13}) =$$
$$\underbrace{\big(\sigma(x_{\underbrace{(\sigma(x_{11}) + \sigma(x_{10}))}_{\text{Step 1, 2, \& 3}} \mod 10}\big)}_{\text{Step 4 \& 5}} \underbrace{+\sigma(x_{12}) + \sigma(x_{13})\big)}_{\text{Step 6 \& 7}} \mod 10 \quad .$$

To evaluate a challenge using this function and the layout template from Table 3.4 users goes through the following steps:

1. Recall the mapping x_{10}
2. Recall the mapping x_{11}
3. Add the values from the two previous steps $j = \sigma(x_{10}) + \sigma(x_{11})$.

4. Locate the element x_j from the table.

5. Recall the mapping $\sigma(x_i)$.

6. Recall the mapping $\sigma(x_{12})$ and add this to the previous value, $z = \sigma(x_i)+\sigma(x_{12})$.

7. Finally recall the mapping $\sigma(x_{13})$ and add it to the previous value, obtaining the final sum $y = z + \sigma(x_{13})$k.

8. y is the response to challenge C.

x_{10} x_{11}	x_{12}	x_{13}
$0 : x_0$	$5 : x_5$	
$1 : x_1$	$6 : x_6$	
$2 : x_2$	$7 : x_7$	
$3 : x_3$	$8 : x_8$	
$4 : x_4$	$9 : x_9$	

Table 3.4: Layout template for displaying challenges.

Each step depends on at most two earlier steps, allowing users to do the calculation without having to store more than two values in memory at any time. After finishing steps 1-3 they will only keep one value in memory, the previous intermediate results are now irrelevant and can be forgotten.

Example 3.4.
Table 3.5 shows the same layout using example objects. The example challenge is $C = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$. Let the mapping function σ be the position in the alphabet mod10, $\sigma(A) = 1, \sigma(B) = 2, \sigma(J) = 0$ etc. The evaluation of this challenge would be the following.

1. $\sigma(K) = 1$, $\sigma(L) = 2$ (K is the 11th letter in the alphabet, thus 11 mod 10 = 1)

2. $(11 + 12) \bmod 10 = 3$

3. $x_3 = D$

4. $\sigma(D) = \underline{4}$, $\sigma(M) = \underline{3}$, $\sigma(N) = \underline{4}$

5. $(4 + 3 + 4) \bmod 10 = \underline{\underline{1}}$

K L	M	N
$0 : A$	$5 : F$	
$1 : B$	$6 : G$	
$2 : C$	$7 : H$	
$3 : D$	$8 : I$	
$4 : E$	$9 : J$	

Table 3.5: Layout template for displaying challenges.

Classification of accounts. Section 2.1.4 discussed how different accounts should be treated differently depending on the consequences of compromise of the accounts. This can be utilized to limit the number of single digit challenges a user needs to calculate when logging into each account. When adding an account to the system users have to classify the account in terms of how big consequences a breach would have. This way the users are in control of how much effort is spent calculating passwords versus how strong the passwords are. This will essentially lower both the average total calculation time and attempts needed to calculate each password.

This chapter has presented the ideas behind the HCP management scheme. It has shown how the scheme works with a secret mapping memorized by the users and a human computable function, which together allow users to compute passwords from challenges stored in persistent memory. Algorithms describing the procedures has been shown and described, as well as discussing the usability challenges of such a scheme.

Chapter 4
Application

This chapter describes how browser extensions are built in the Google Chrome web browser, focusing on architecture and security features. Browser extensions can be utilized to build an application implementing the *HCP scheme* as described in chapter 3. The scheme is different from other traditional password managers since it does not *store* the password, but the challenges *help* the user to remember strong passwords. The idea by using browser extensions to implement this is to have an extension monitor the password fields of the sites a user visits and update the challenges depending on the current state of the active site. This technique is described and a prototype extension demonstrated.

4.1 Browser Extensions

Modern computer users shift towards doing more and more work through their web browsers. Web applications have become popular due to the ubiquity of browsers, thus allowing web apps to run anywhere. A web app can run on any platform running a web browser. Updates can be applied quickly without having to distribute patches to a possibly huge amount of devices.

Browser extensions add additional features to the web browser allowing users to tweak the experience of the web pages visited. Typical examples are extensions adding to, or tweaking already present features of the browser such as changing how bookmarks are managed, or adding additional features such as blocking advertisements. Lately browser extensions have been extended even further allowing standalone applications to be developed running as native applications [1]. This allow developers to create desktop apps using the same technology as in web apps, mainly HMTL5, Javascript and CCS.

[1] A new breed of Chrome apps - http://chrome.blogspot.no/2013/09/a-new-breed-of-chrome-apps.html - accessed: 2015-03-02

This section presents Google Chrome browser extensions, including architecture and security mechanisms.

4.1.1 Extension Security

Browser extensions introduce some security concerns which must not be forgotten while developing applications using this environment. Chrome extensions run in the browser with access to both the DOM (Document Object Model) of the active page as well as the native file system and connected devices. The overall architecture of a Chrome extension is summarized in Figure 4.1 and described in the chrome extension documentation[2]. This section describes the architecture considering security concerns relevant when developing chrome extensions which handle sensitive data such as passwords.

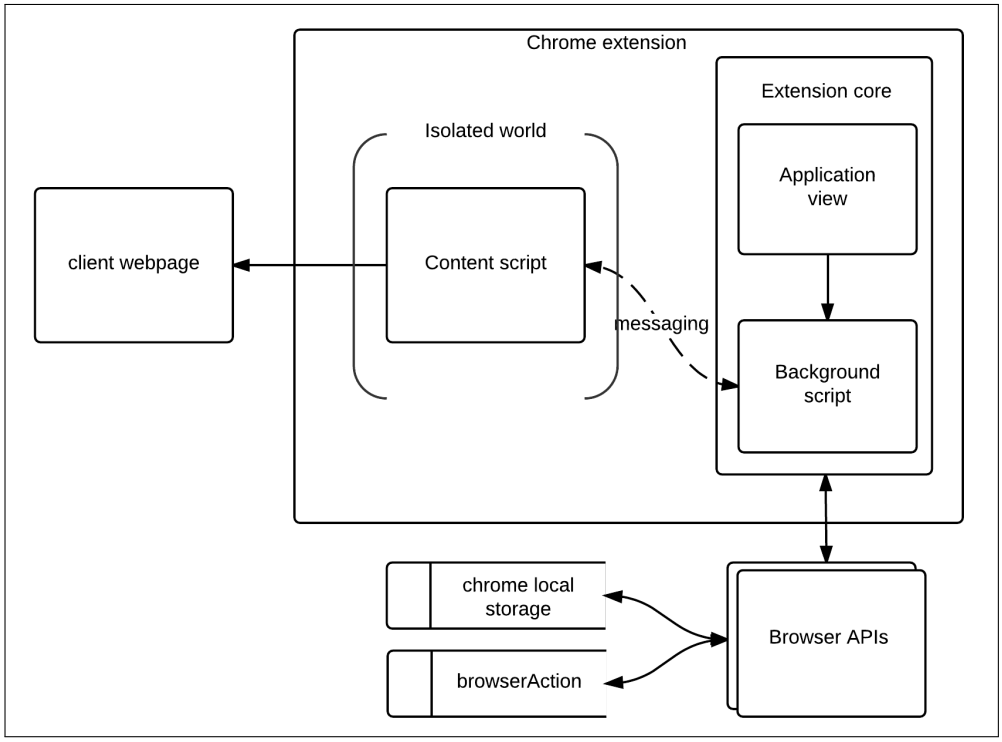


Figure 4.1: Chrome extension architecture.

Earlier extensions written for IE and Firefox ran in the same process as the browser and shared the same privileges. This made extensions an attractive entry

[2]What are extensions? - https://developer.chrome.com/extensions/overview - accessed 2015-03-02

point for attackers, since a buggy extension could leave security holes leaking sensitive information or even provide an entry point to the underlying operating system. For these browsers several frameworks for security have been proposed [LLV08, DG09], trying to mitigate vulnerabilities is browser extensions.

The Chrome extensions architecture is built from scratch with security in mind. Chrome uses a permission system following three principles [LZC+]; *least privileges*, *privilege separation* and *process isolation*.

Least privilege specifies that extensions should only have the privileges they need to function, not share those of the browser. The privileges of each extension are requested in the *manifest* file[3]. This json file needs to be included in all Chrome extensions, and consist of all the permissions needed by the extension as well as some meta data and version information. This is done to prevent compromised extensions from exploiting other permissions than those available at runtime. An example of a manifest file can look like this:

```
{
    "name": "Example extensions",
    "description": "An example extensions to demonstrate how the
                    manifest file works.",
    "version": "1.2",
    "manifest_version": "2"
    "background_page": "main.hmtl",
    "permissions": [
        "bookmarks",
        "storage",
        "https://*.ntnu.no"
    ]
}
```

This extension has specified access to the bookmarks API, Chrome local storage and all sub domains of ntnu.no. Extensions can request different permissions in the manifest file including web site access, API access and native messaging. If an extension contains weaknesses it will not compromise any other parts of the system not covered by the specified privileges. For the least privileges approach to work properly each developer should only request the permissions needed. Barth et al. [BFSB10] examined this behavior and concluded that developers of Chrome extensions usually limit the origins requested to the ones needed.

[3]Manifest file format - https://developer.chrome.com/apps/manifest - accessed 2015-03-04

Privilege separation. Chrome extensions are, as mentioned, divided into components; content scripts, extension core and native binaries. The addition of native binaries allows extensions to run arbitrary code on the host computer, thus posing a serious security threat. This project does not use this permission, and does not have the accompanying problems neither.

Content scripts are Javascript files allowing extensions to communicate with untrusted web content of the active web page. These scripts are instantiated for each visited web page and has direct access to the DOM of these, allowing both monitoring and editing of DOM elements. To be able to inject content scripts to a visited page, the origin of the site has to be added to the manifest file. Other than this permission, content script are only allowed to communicate with the extension core. It is important that the privileges of these scripts are at the minimum level since they are at high risk of being attacked by malicious web sites [BCJ+14], due to the direct interaction with the DOM.

The *extension core* is the application interface responsible for interaction with the user as well as long running background jobs and business logic. The core is written in HTML and Javascript and is responsible for spawning popups and panels, as well as listening for browser action. The typical way to activate an extension is by clicking an icon in the navigation bar, which then activates either a popup or a detached panel. The core is the component with the most privileges as it does not interact with any insecure content directly, only through direct messaging to a content script or using http requests if the target origin is defined in the manifest.

In addition to this, the core has access to the extension APIs, these are special-purpose interfaces providing additional features such as alarms, bookmarks, cookie and file storage. The APIs are made available through the manifest file and only those specified there can be used. Figure 4.2 illustrates the interaction between the background page, content scripts, panels and active web page. The information flow starts by clicking the extension's icon in the navigation bar which launches the background page spawning a panel in the browser. A content script is injected in the current web site (google.no in the example), the script now have access to the DOM of this site and can communicate with the background which in turn can update the panel.

Process isolation is a set of mechanisms shielding the component from each other and from the web. Usually when Javascript is loaded from the web the authority of the script is limited to the origin from where the script is loaded [BFSB10]. Since the scripts used by the extensions are loaded from the file system, they do not have an origin in the same sense, and thus need to be assigned one. This is done by including a public key in the url of the extension, allowing a packaged extension

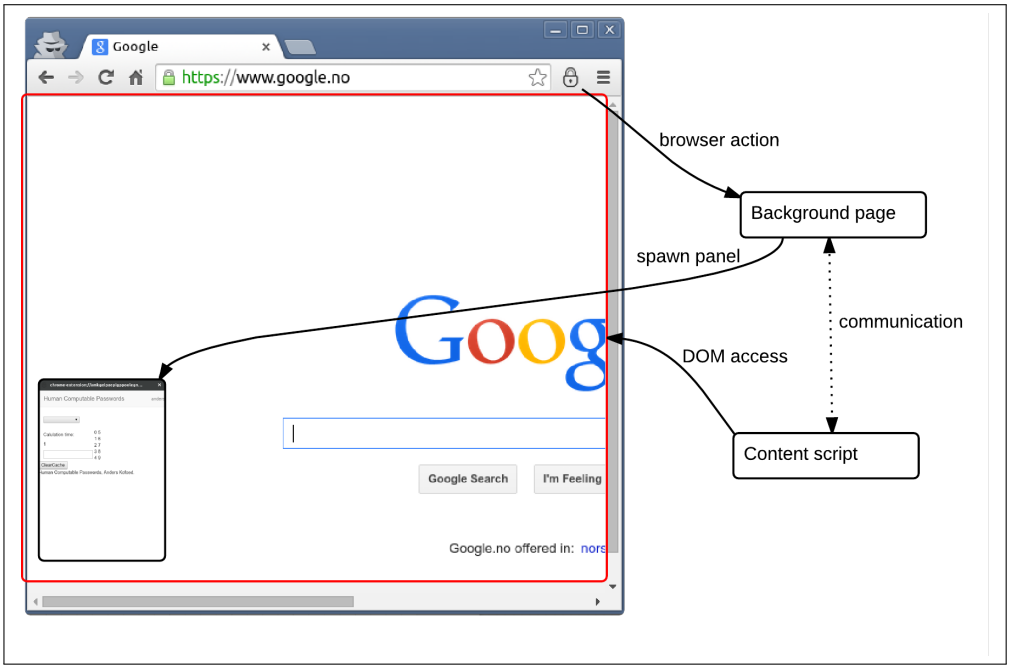


Figure 4.2: Chrome extensions browser action and content scripts.

to sign itself, freeing it from any naming authority or similar. The public key also enables usage of persistent data storage, since the origin of the extension can stay the same throughout updates and patches. This would not be possible otherwise since the Chrome local storage API relies on origin. Each extension has a private/public key pair, a hash of the public key is used as ID and each updated version uploaded will be sign with the private key. By doing this the ID stays the same throughout versions and can be verified by when uploading and by Chrome storage after releasing a new version.

The different components also run in different processes. The content scripts are injected and ran in the same process as the active web page, while the core run in its own process started when the extension is initiated. This protects against Javascript injections from malicious web sites [BZW13]. Since the content script executes in the same environment as the active web page, users may visit websites hosting content meant to exploit extensions [CFW12], possibly stealing sensitive information or issue fake requests.

Finally content scripts are ran in a separate Javascript environment isolating it from the possibly insecure environment of the web site. The environment of the content scripts are called isolated world, which in practice is a separate set of

Javascript objects reflecting the ones of the underlying DOM of the web page. This means that the content script can read and edit the DOM of the page it is injected into, but not access variables or Javascript functions present in the web page. Both the page and the content scripts sees no other Javascript executing in their own isolated world, but they share the same DOM[4].

4.2 Human Computable Passwords Chrome Extension

Chrome extensions are very useful in that they can be run from any computer with Google Chrome installed, thus on any operating system and on any computer. An extension makes it possible to run applications while browsing the web, which in the case of a password management scheme is very useful. An application meant to help the user recall complex passwords should preferably be visible simultaneously with the password field. Popular password management software today are usually web applications, mobile applications or native desktop applications[5], some of these might include plug-ins in from of browser extensions. All of these password managers are reliably storing all the passwords as described in section 2.2.2, then they are either auto-filled into the login fields or through copy pasting manually, which introduce several security issues [BFSB10, BZW13, BCJ+14, CFW12, LZC+, SJB+14]

This section presents the design and prototype implementation of the HCP scheme as presented in chapter 3. The design evolves around the fact that the scheme does not have to store anything securely, though it is important to make it as easy as possible for the user. The architecture is similar to the one explained in section 4.1 using content scripts to monitor the password fields of the active browser session. The user interface is presented through a “panel” in the browser. Panels are windows that stay in focus while interacting with other windows or applications [6].

The application implementing the HCP scheme is an extension helping users with storage and management of challenges for their accounts. The generation of secret mappings is not part of the application, this should be done through a separate program on the users’ local computers. Such a program would follow algorithm 3.1. The requirements and key components of the design are described next.

4.2.1 System Requirements.

- Provide users interface making it easy for the user to calculate their password.

[4]Content Scripts - https://developer.chrome.com/extensions/content_scripts - accessed: 2015-03-05

[5]Five Best Password Managers - http://lifehacker.com/5529133/five-best-password-managers

[6]"Panels"- The Chromium Project. https://www.chromium.org/developers/design-documents/extensions/proposed-changes/apis-under-development/panels

- The application should keep track of the active site, displaying challenges for the correct site without user interaction.
- Add new sites to the system easily.
- The displayed challenge should update seamlessly while typing the password.
- Users should be able to type their passwords directly in the password field of the active site.
- When adding a new site to the system, users should be able to classify the account (see section 2.1.4), affecting the number of single digit challenges created.

4.2.2 Key Components

AngularJS.

The front-end is built and updated using AngularJS[7], which is an open source, client-side Javascript framework. Angular is built using a variation of the model-view-controller architecture [Dea09], though the creators of Angular state that Angular is a model-view-whatever framework[8], the point being that what the architecture is called is not important.

The *view*s in Angular are templates written entirely in HTML, making it easy to read and update. The *controller* contains all the business logic used by the view. The views and controllers are connected using a shared object called *$scope*, variables or functions on this object is usually defined in the controller and accessed by the view using double curly brackets (e.g. {{name}} to access the name variable on *$scope*). Figure 4.3 shows how scope is used to share variables between the controller and the view. See "Angular Essentials - Rodrigo Branas"[Bra14] for a step-by-step introduction to AngularJS.

The main benefits of using Angular in this project is the data-binding which makes it easy to update the HTML shown to the user. The extension takes advantage of this when updating the challenges seamlessly while the user calculates his password. The content of the extension changes according to the data filled in the password field without reloading the extension.

Chrome Storage.

Local storage[9] is a way for applications to store data persistently and securely in the browser of clients. It is meant as an improvement of cookies which is the usual way

[7] AngularJS - https://angularjs.org/
[8] Model-View-Whatever - https://plus.google.com/+AngularJS/posts/aZNVhj355G2
[9] Local storage - http://diveintohtml5.info/storage.html

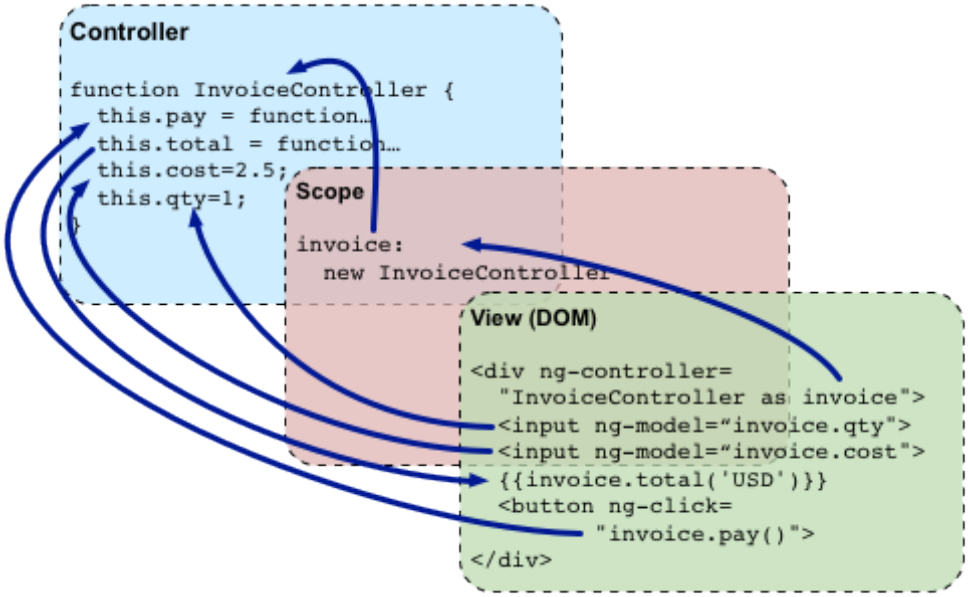


Figure 4.3: Angular data binding with controller, view and scope. Figure from AngularJs developer guide.

of storing user data across sessions. The biggest problem with cookies is that they are sent with every HTTP request and thus slow down the applications using them. Local storage allows applications to store (key, value)-pairs in the browser.

Chrome storage[10] is close to the same as local storage, differences being that Chrome local storage allows applications to store data in what is called "chrome.storage.sync". This specific storage saves data locally, but also syncs it with the currently active Chrome account, allowing users to log into their accounts in any chrome browser and access the same application data. Chrome storage also allows storage of objects compared to local storage which only allows storing strings. This project is using Chrome storage to persistently store challenges across sessions, and also provide backup. This way each user is able to keep challenges for all their accounts within the storage of their Google account. This is a important feature since it allows access to the challenges from remote locations as well.

Content Scripts.

Content scripts are Javascript files running in the context of the active web page, as browsed by the user. These scripts have direct access to the content of the active site and can thus monitor and attach event listeners to the content of the page. The

[10]chrome.storage - https://developer.chrome.com/extensions/storage

content scripts are isolated from the extension itself as describe in section 4.1.1, thus protecting the extension from possibly harmful sites trying to exploit weaknesses in the content script. Because of this the content scripts has to communicate with the extension through Google Chrome's built-in message passing system. Chrome message passing[11] allows scripts to listen for and respond to messages. One side sets up an event listener listening for messages, when a message is sent from the other side this event triggers and the message can be received and parsed.

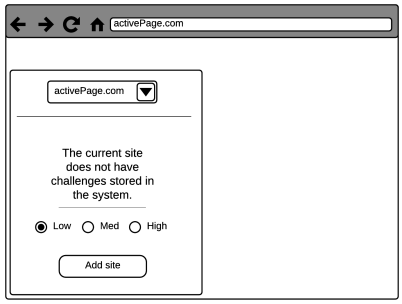
The content scripts and the message passing system are likely points of attack for adversaries. The content script can monitor the value of the password field and thus, in theory, steal passwords if a malicious script was able to trick it into leaking them. The communication channel is not particularly prone to attacks since even if an adversary eavesdropped all data sent on it, the only information leaked would be the current length of the password. It is though important that the design is like this since the mistake of sending the whole password string, which might seem like a solution, would be potentially dangerous.

Typical usage of content scripts in this application is to attach event listeners to the password fields of the pages visited by users, and message the extension when the value of the password field changes. When a change happens, the content script sends a message containing the current length of the typed password, so that the extension can display the correct challenge. In example if the user has entered 4 characters of his password the extension should display the 5th challenge etc. The content script also keeps the extension updated on the URL of the current page, by sending a message every time a page is loaded. The extension then displays the challenges corresponding to the password of that site. If the site does not have challenges stored by the application, users can generate new ones and store these in the system through clicking a button.

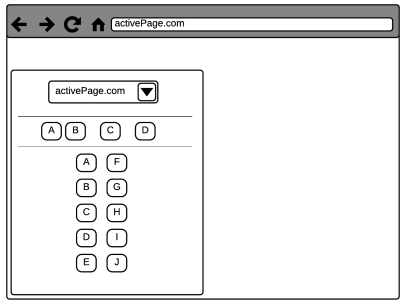
4.2.3 User Interface

The user interface is very simple, two possible screens are displayed to the user. Either, challenges associated with the current active page, or a dialog asking users to generate new challenges. When adding a new site, users are asked to categorize the site as low, medium or highly sensitive. Depending on this choice a set amount of challenges are generated and stored. The most important feature of the user interface is that it automatically updates depending on what site the user is currently browsing, and displays the correct challenge while typing passwords. Wireframes illustrating the page schematics of the extension are shown in figures 4.4a and 4.4b.

[11] Chrome message passing - https://developer.chrome.com/extensions/messaging



(a) Screen seen by the user when launching the extension while visiting a page that is not stored in the system.



(b) Screen seen by the user when loading a site with challenges stored in the system.

Figure 4.4: Wireframes illustrating the page schematics of the extension.

4.2.4 Implementation

The implementation follows the design described in the previous section, this section describes how the extension is implemented and demonstrates the actual application in action.

Figure 4.5 shows the complete architecture of the system. The content script attaches an event listener to the password field of the active site. If the content of the password field changes, the script sends a message using Chrome message passing containing the current length of the password. The controller receives the new password length from the content script, and updates the challenges displayed. When users visit a new site, the URL is sent to the controller, which then updates the view with new challenges associated with that site.

The classes seen in figure 4.5 are all represented in the implemented version of the extension. The construction and responsibilities of the classes are summarized in the following list. The code of the implementation for each component is explained in more detail in appendix A.

- **Controller** is responsible for the business logic related to the information shown to the users in the extension. The controller first tries to load the user data stored in Chrome storage, if no user has been created, typically if it is the first time the extension is loaded, a new user is created. The controller keeps track of the URL of the current active page and the current length of the password field. It listens for messages from the content scripts and updates

the URL and password length variables with the info received. The controller class used in the extension is described in appendix A.2.

- **ChromeStorage**[12] is a utility resource used to access Chrome local storage (see section 4.2.2). This module is essentially a wrapper making it easier to retrieve data from Chrome storage, while also providing useful debugging and cache management functions.

- **Content script** is responsible for attaching event listeners to the password field of the currently active page.. It also listens to the onload event[13], sending URL updates to the controller every time a new page is loaded. The content script of the extension built in this project can be seen appendix A.1.

- **main.html** is the view of the application, responsible for the user interface. The view receives updates from the controller through the $scope variable. When the controller updates in example the current password length, this is reflected in the view where different objects are shown. If the current URL is not in the user's list of saved sites, the view shows a dialog allowing the user to add it. When adding a new site the user should update the password of that site by calculating the response to the challenges. A snippet of the view is shown in appendix A.4.

- **app.js** is the focal point of the application, responsible for initializing the Angular app and routing of views. In this project this file also includes some helper functions which can be seen in appendix A.3.

Random numbers are an important component of the application since all the security relies on the fact that the secret mapping and the challenges are chosen at random. As for this application, the only concern is the challenges, since the secret mapping would be generated using a separate program on each client's machines. Javascript provide a function called *Math.random* which is not considered cryptographically secure [SHB09] and should thus not be used. New browsers now support a new method which are considered to be the best suited random source for cryptographic purposes, the method is called *window.crypto.getRandomValues()*[14]. The application uses this function as the source of randomness to generate challenges. The random number generator used can be seen in appendix A.3.

Mapping objects. The application as implemented and demonstrated in this chapter uses the alphabet as mapping objects. As explained in chapter 3, these

[12] angular-chrome-storage - https://github.com/infomofo/angular-chrome-storage

[13] Event triggered when a new page is loaded in the browser.

[14] RandomSource.getRandomValues() - https://developer.mozilla.org/en-US/docs/Web/API/RandomSource/getRandomValues

objects could be anything and are fixed in this presentation for simplicity. There is a folder in the implementation where all the pictures are kept, these could easily be exchanged with other pictures to better suite the user.

Launching the application is done by clicking an icon in the browser toolbar which launches the extension in a panel floating on top of the other browser windows and tabs. This behavior is specified in the *background page*[15]. The background page is the "launcher" of the extension, it waits for users to click the extension icon, firing a browser action event. On catching this event, the script spawns a panel, with the Angular application as content. Listing 4.1 shows the background page of this extension. After spawning the panel, the background page is standby doing nothing, everything now happens through the Angular application running inside the panel.

```
1 chrome.browserAction.onClicked.addListener(function() {
2     chrome.windows.create({
3         url: chrome.extension.getURL('main.html'),
4         type: 'panel',
5         focused: true,
6         height: 520,
7         width: 400,
8     });
9 });
```

Listing 4.1: Background page.

4.2.5 Demonstration

After launching the extension, as previously explained, users are presented with the window shown in figure 4.6. In this example the active site does not have a record in the user's challenge database, thus the "add new" dialog. By clicking the button, new challenges are generated and stored in the database as a new site. The site is automatically added with the current site domain as key together with randomly generated challenges. Before adding the site, users can specify which category they rate the site in (low, medium or high), depending on this choice, 5, 10 or 15 challenges are generated. Next time the user visits the same page, challenges will appear. After adding a new site, it is the users responsibility to change the password of this site so it matches the challenges.

Figures 4.7a-d shows how users would use the extension when logging in to a site. When launching it, the system receives the current sites domain and loads the first challenge associated wit that site. Users then calculates the first character using

[15]Background pages - https://developer.chrome.com/extensions/background_pages

the challenges displayed. When this is entered in the password field, new challenges appear until the whole password is calculated.

Data Flow

Figure 4.8 illustrates the life cycle of the application. When users visit a site, an event is triggered and caught by the content script, which in turn informs the controller about the newly loaded site. The controller then tries to load challenges for that site from the storage; if there is a record, the view is updated with the first challenge, if not the "add new"-dialog is loaded. Next event is triggered when the user enters a character in the password field, on receiving notice from the content script the controller updates the view with a new challenge depending on the length of the password. This then goes on until the entire password is calculated, eventually closing the extension panel.

4.2.6 Discussion

The approach of this project when designing the HCP extension was to make a clean and intuitive application, making the password management scheme feasible to use. The extension does not require any user interaction, except from when adding new sites, this was done so users could focus on doing the calculation correctly without having to keep track of the challenges. The scheme itself is also quit complicated so it is important for users to focus on as few operations as possible. (I.e. the simplest approach would be to have a "next"-button for users to click between every character calculated.)

The choice of a browser extension was made with the same mind set, making the application helpful and not distracting. It should be easy to start using the application, requiring as little configuration and managing as possible. The design and construction make it so that the only configuration required is the changing of passwords when adding sites, which essentially can not be circumvent. Panels is another very useful feature, since the panels float on top of the other browser windows and keep focus while typing. Without panels the user would have to switch between windows when calculating a character and typing it in the password field, which would be a huge drawback both in terms of time and usability.

One apparent problem with using extensions is that they are not supported in mobile browsers, and probably will never be[16]. The solution to this would be to make an additional application for mobile, and sync the data to another service available on mobile as well. This could be done quite easily since the data is stored as text, the only change needed would be to substitute the storage module with some other

[16]Multidevice FAQ - https://developer.chrome.com/multidevice/faq

cloud storage service. The problem if the application was built for mobile would be the lack of space to display challenges while typing the password.

As mentioned in section 2.1.1, some sites may enforce password policies [SBSB07], forcing users to include different types of characters. HCP does only produce digits which might be a problem on some sites. A solution to this is to either have an option in the application to mark accounts where policies are enforced and append dummy characters to the password produces by the scheme. E.g if the application displayed challenges producing the password “234554675687856”, and the site requires a symbol and upper/lower case letters, a user would append “Xa*” to the calculated password to circumvent the policy.

The decision to make a browser extension was made on behalf of the mentioned pros and cons, with the most important parameter being disturbance. Only extensions allow a seamless, non-interfering user interface. The other options, namely web application and mobile application, would require switching between windows or at least switching of focus, as well as requiring the user to manually chose the correct site and browse through the different challenges. This is all done automatically with the construction presented in this chapter. The main goal of the design was to include the least amount of unnecessary features, with a clean and unobstructive interface. A summary of the strengths of the chosen design is listed next.

- Easy to use, require little to no user interaction after configuration.
- Panel that stays on top while entering the password allows for quick and easy calculation without stopping to update and manage challenges.
- Chrome.storage.sync automatically synchronize the stored challenges to the users’ Google accounts allowing persistent usage across devices and accounts.
- All data is stored as text strings, this makes it easy to integrate with other services in the future. Users could also store the challenges elsewhere if need be, e.g. as text locally or even print the challenges on paper in the extreme case.

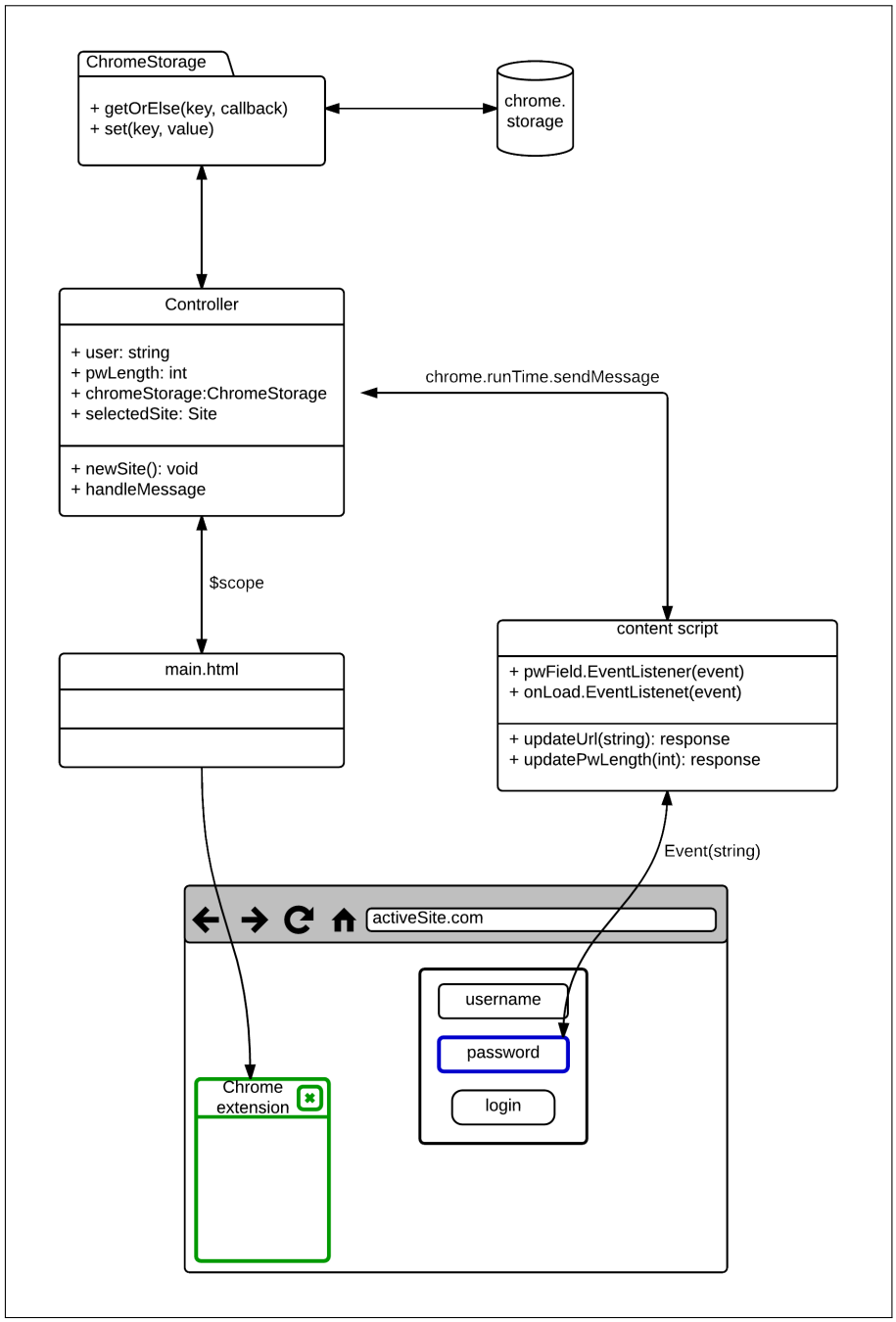


Figure 4.5: Class diagram of the HCP Chrome extension.

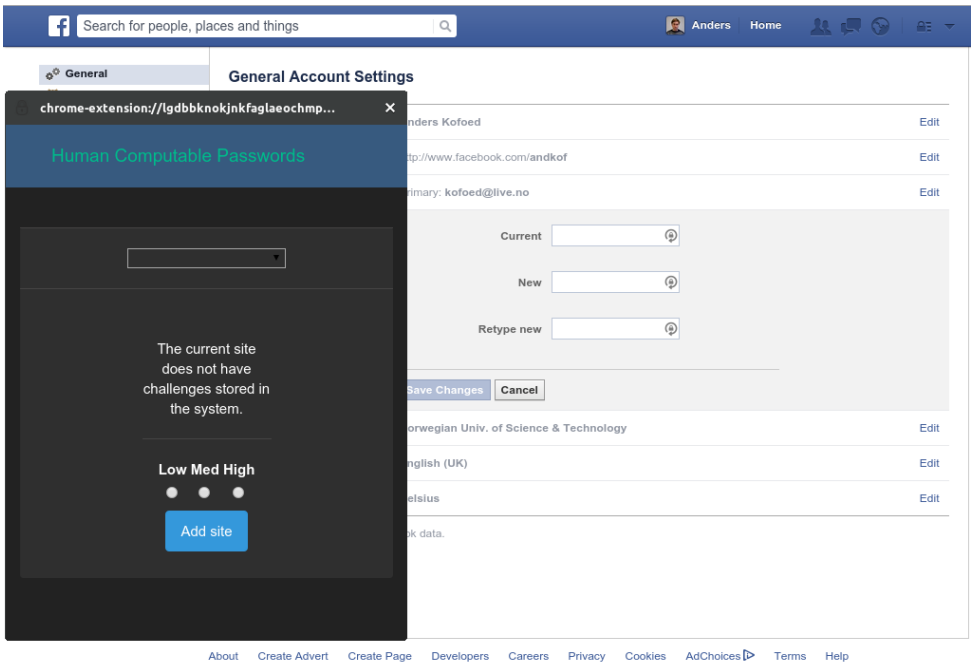
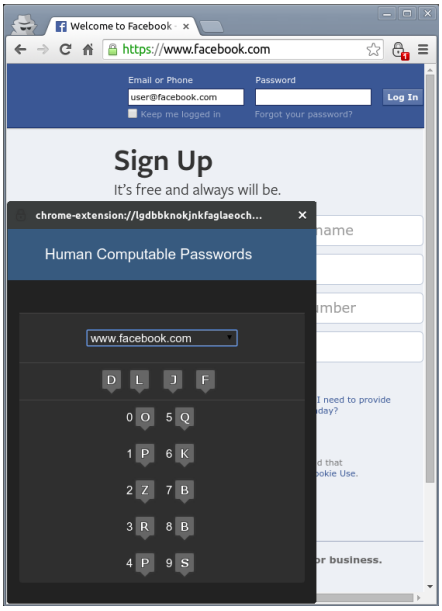
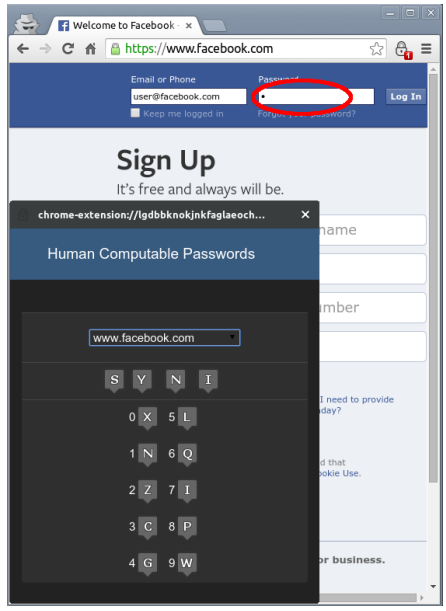


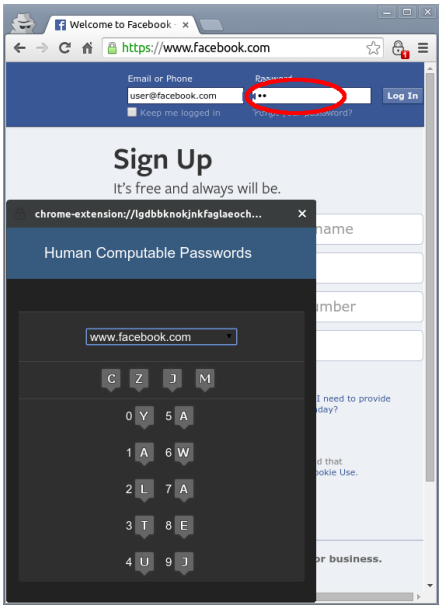
Figure 4.6: Screen as seen by the user after launching the extensions while on a page without stored challenges. After clicking the "Add site"-button, the view updates, showing the newly generate challenges. Users should then calculate the response and change the password of the site to match it. Next time the same user wants to log in to the site, they will calculate the response again as seen in figure 4.7.



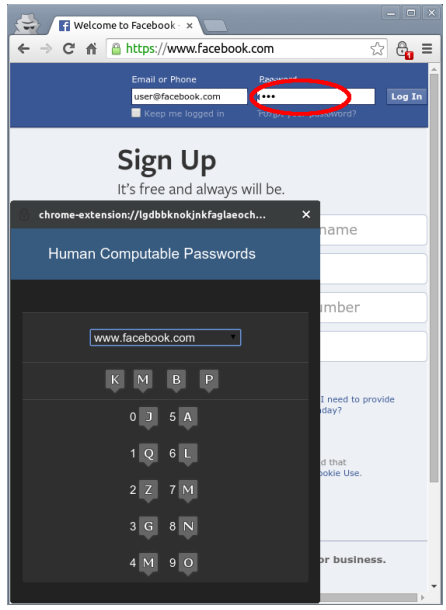
(a)



(b)



(c)



(d)

Figure 4.7: Challenge screens as seen by the user while entering password. The challenges update when the user enter a new character in the password field.

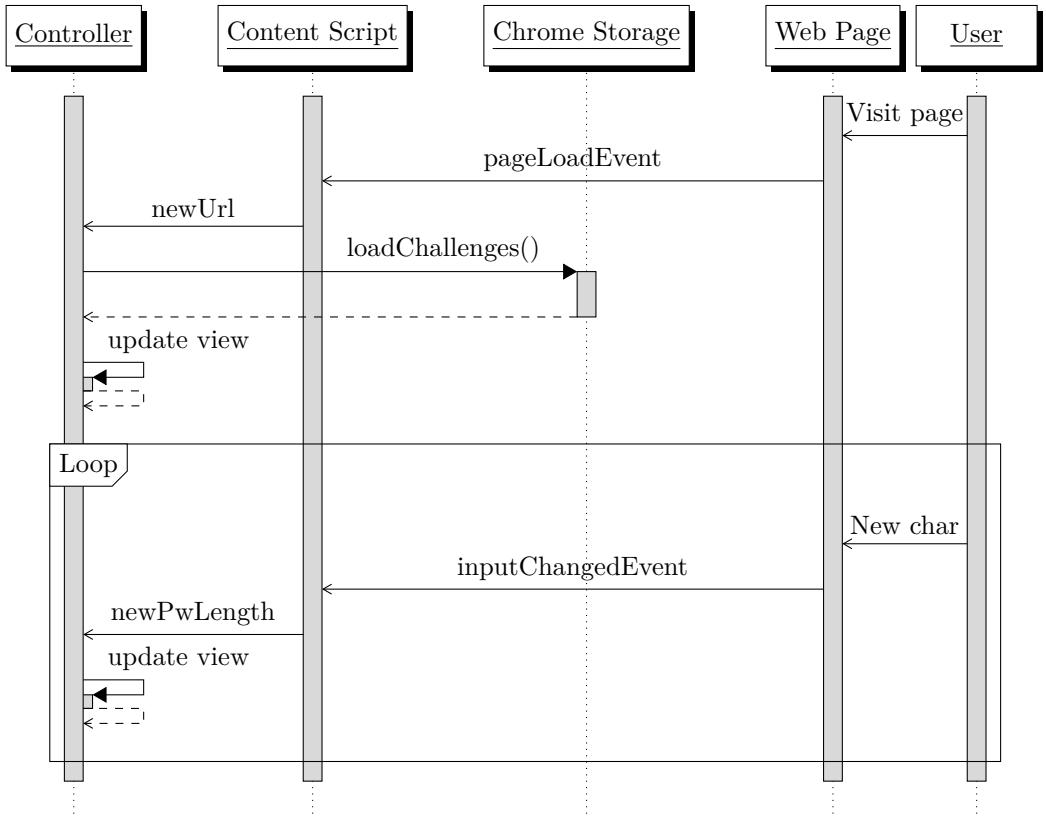


Figure 4.8: Sequence diagram showing the flow in the system when a page is loaded and characters typed in a password field.

Chapter 5
Usability Experiment

This chapter presents the design and execution of an experiment trying to measure the performance of the HCP scheme. The experiment is designed as a web application implementing the scheme, allowing participants to test how the scheme would work in practice while measuring how fast and reliable the computation is performed. The application acts as both a demonstration app and a tool for gathering performance data. It has four sections designed to help the participants understand the scheme and get familiar with the computation technique. First, a demonstration video is shown explaining how to compute a password from a challenge, then users are asked to enter some demographic data. Next is the practice section, where users are supposed to practice doing the calculation until feeling comfortable solving challenges without error. Finally is the experiment part, where the calculations are timed and correctness monitored. After finishing the calculations user submit the data and can choose to continue doing more experiments now, or later.

5.1 Experiment Objective

The goal of the experiment is to measure how hard it is for a user to learn the scheme, i.e. how fast and reliable users do the calculations. The experiment measures the calculation times of the participants, and also monitor their progression after several trials. Users should also be able to calculate passwords without too many mistakes. The failure rate is maybe the most important variable to measure; if a user averages more than one mistake for each password calculated the scheme would not work in practice, since most passwords calculated would be wrong.

It is important to note that this project does not see the HCP scheme as a replacement for the widely used standard password managers. It is regarded as a solution for users interested in a secure and reliable way of keeping track of strong passwords, without having to trust a password management service or application. This experiment tests if it is even possible, for users willing to go through the trouble

of learning a secret mapping, to use the scheme as an everyday solution.

The objectives of the experiment can be summarized as the following.

- Measure the average calculation times, both for each participant and for the whole population.
- Measure how much users improve after several trials.
- Measure the average failure rate for each participant.

5.2 Method

The experiment does not try to test any preset hypothesis as it is not clear what to expect regarding either of the tested values. It is not known how hard it is do to the calculations, neither regarding calculation times or failure rates. The results of the experiment does not give an answer, but rather a basis for further data collection through larger scale experiments, for example using crowd sourcing or social medias.

Exploratory data analysis (EDA) was introduced by John W. Turkey [Tuk77] and involves analysis of data without a prior hypothesis to test. The technique promotes exploration of data to possible find characteristics not previously considered, and essentially suggesting hypotheses to be tested in later surveys or experiments. Velleman and Hoaglin [HD79] describe EDA as a contrast to the formal scientific method involving stating a hypothesis, collecting data and applying a statistical test of the hypothesis. EDA often involves making graphical representations of the data, and then trying to find interesting characteristics and relations. EDA does not conclude with a hypothesis test based on the collected data, but is the first step of an iterative process trying to reveal facts.

Remark 5.1.
The project does not store any personal information about the participants, and is thus not subject to notification to the "Norwegian Social Science Data Service"[1]. The experiment only records an anonymous id plus the experiment results consisting of computation time and correctness of the calculations done. Some demographic data is also recorded (age, area of study), but the data can not be connected to person and are thus not regarded as personal information.

After conducting the experiment, the data is examined and significant characteristics discussed. There are some obvious parameters to explore, including means

[1] Is my research project subject to notification? - http://www.nsd.uib.no/nsd/english/pvo.html

and standard deviation of the calculation times and failure rate, as well as how these change with practice.

The usability of the scheme directly relies on the calculation time and failure rate as discussed in section 3.3. This can be discussed further after obtaining some numbers giving a picture of what is normal and possible in terms of speed and reliability.

A result showing that more than 95% of all calculations are correct would be promising, since it would mean that approximately 60% of the passwords calculated would be correct, given length 10 passwords. If the failure rate is significantly worse, the scheme would more often than not be useless since most users would obtain a faulty password when trying to log in. The conclusion of the experiment presented in this project will either way have to be tested more thoroughly, possibly using the same experiment setup or preferably with a throughly random mapping as well.

5.3 Experiment Setup

The experiment presents and demonstrates the calculation technique to the users, which then is given a chance to practice until fairly familiar with the mechanics. Finally users are asked to calculate a complete password challenge, from which the time spent on each single digit challenge is recorded, as well as if the calculation was correct or not. The practice section allows users to learn through trial and error, using backspace to go back and forth between challenges while also given feedback on the correctness. The experiment view on the other hand does not give any feedback and does not allow user to go back after entering a character. This is done to make sure all mistakes are recorded, even if it is only a miss click.

5.3.1 Secret Mapping

The biggest decision made in regards to the experiment design was how to simulate the operation “recall” i.e. recalling from memory. It would not be feasible to ask all the participants to memorize a secret mapping beforehand as this would make it very hard to find volunteers. The chosen solution to this problem is to include a “cheat sheet” in addition to the displayed challenges, i.e. a list of object to digit mappings shown separately but in the same view as the challenge.

After some testing it became apparent that there was a big difference between recalling from memory and actually “reading” from a list, which this approach eventually is. To make the operation more similar to the desired recall operation, the mapping was changed from random alphabet positions (e.g A=1, B=2, C=3). This way one does not always have to read in the table to know the mapping. Most users

will be able to know instantly what the mapping for at least the first and eventually, after some practice, all the letters, much easier than with a random mapping. The is the closest way of mimicking the actual operations of the scheme without having the participants memorize an actual mapping. It is not in any way certain that this shortcut reflect the real world act of recalling from memory, but it is assumed to be "close enough" for the experiment.

Remark 5.2.
The author of this project did memorize a mapping of both 10 and later 20 mappings without significantly different calculation times compared to using the alphabet positions.

5.3.2 Participants

The participants for the experiment were chosen mostly from people known by the author, this limits the number of participants somewhat. This was done to ensure the quality of the data samples. The experiment could have been distributed through crowd sourcing services or social media, probably increasing the number of participants drastically. The problem with this approach, and the reason for not doing it, is that the experiment requires absolute concentration which can not be assured from "casual" participants accessing the experiment through a link posted on facebook. Since every participant calculates 10 single digit challenges for each trial of the experiment, there is still a decent amount of data samples, even with relatively few participants.

Even if the experiment might be limited by the number of participants, the results are still considered to be interesting. Since the scheme tested is not supposed to be a widely-deployed password manager, it might be sufficient if the experiment can show that some users are able achieve what is considered sufficient usability. It should also be noted that most of the participants are regarded above average in mathematics. This might be a strength or a weakness of the experiment as it does not represent a wide range of the population, but the calculation times should at least be stronger than average.

5.4 Web Application

The experiment application is similar in design to the Chrome extension presented in chapter 4, but implemented as a web application. It is implemented using the AngularJS framework (as described in chapter 4) and a mongoDB[2] database to store the results, a cookie in the users' browser is used to keep track of users across trials.

[2]mongoDB - https://www.mongodb.org/

The application consists of four sections to flip through, with the last one being the actual experiment. First, users are presented with a video demonstration of the scheme and instructions on how to calculate passwords. The slides used to instruct users can be seen in Appendix B. After watching the demo, users are asked to enter some demographic data (age and occupation). Next, is a section which looks the same as the experiment, but without the actual recording of data, this is the practice section. The purpose of this section is to give users a chance to verify that they have understood the scheme and are actually able to calculate passwords. When users are ready to start the experiment, they can continue on to the actual experiment, which is the final view. The experiment is exactly like the practice section, without response on correctness and redo capabilities, with backspace deactivated. The first sample of the experiment is not counted towards the results, allowing users to get ready and start when they feel like it.

Section 3.3.3 discusses how a single digit challenge can be presented to users in a logical way, possibly making it more efficient to calculate. The experiment uses a similar layout to the one shown in Table 3.4 and also as in the Chrome extension in chapter 4. The challenges update in the same way as in the chrome extension so users should be able to calculate the responses continuously. The calculation time is recorded for each single digit challenge computed, together with a boolean value representing if the result was correct or not.

Figure 5.1 shows the screen after completing a trial of the experiment, next users will push the submit button and the sample would be saved in the database together with a random identification stored in a cookie in the users' browsers. Users can then redo the experiment several times keeping the same id, making it possible to track each participant's development.

As for the storage of results a noSQL database called MongoDB is used. After completing a trial of the experiment a Javascript object is stored in the database, Table 5.1 shows an example of this object. Note the *hcp_id* field which is fetched from a cookie in the participants' browsers, making it persist across several trials.

The views of the web application can be seen in Appendix C or by visiting the hosted experiment site at hcp.sp1nakr.com.

5.5 Results

This section presents the results of the conducted experiment and investigate the characteristics of the data recorded. The focus is on the calculation times and failure rates, which both are important factors in the resulting usability of the scheme, as discussed in section 3.3.

hcp_id	1b7aa17ea0
occupation	Technology student
age	27
results	$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$
calcTimes	$[12.047, 7.115, 8.022, 9.283, 10.544, 8.807, 9.925, 9.78, 7.11, 8.187]$

Table 5.1: Experiment object after completing a trial, as stored in the database.

The observations made are not necessarily representative for all users, because of the relatively small number of samples (467 single digit challenges), but it is still interesting to investigate the consequences of the results. Even if the results show that the scheme achieves a lower level of usability than what are considered acceptable, it might still function well for some users with the right amount of practice. The limitations of the experiment is discussed, in addition to results and the consequences of these.

5.5.1 Calculation Times.

How fast a user is able to calculate the response to a single digit challenge is a concrete measure of the usability of the scheme. Figure 5.2 shows the distribution of calculation times of all the experiment samples, the calculated average off all the 467 trials are 10.296 seconds. The median value is 9.406 with standard deviation of 3.64 which also can be observed in the figure.

Recall conjecture 3.1 from section 3.1.2. The function f (see definition 3.3) used in this project is $(P, 9, 3)$-computable. The conjecture then defines the variable γ_H for user H as $\gamma_H = \frac{\hat{t}}{9}$.

The average γ of all the participants are $\tilde{\gamma} = \frac{10.104}{9} = 1.227$. This is slightly higher than the 7.5 seconds ($\gamma_H \leq 1$) Blocki [BBD14] predicts that users with moderate mathematical backgrounds should achieve. It is still not unreasonably high, and should not be a direct hindrance. Calculating a 10 character password would for example take $10 \cdot 10 = 100$ seconds, just above 1.5 minutes, to calculate, 15 character would take 2.5 minutes. On average spending 2-3 minutes calculating a password might seem like a lot, but users concerned about the security of their accounts, might be willing to make this trade-off. Especially if users are worried about putting passwords in the hands of a online service to store persistently (e.g. *lastpass.com*), might be willing to put quite some time into calculating passwords instead.

It is also relevant to study the evolution of each user in terms calculation times. Figure 5.3 illustrates the average calculation time for each ith trail per user. $i = 1$ represent the average of the first calculation done by each user, $i = 2$ the second

calculation etc. The figure clearly illustrates that the averages decrease with practice. This could have been expected, since users will be more and more familiar with the calculation procedure for each trial. The consequence of this observation is that the average calculation time, and consequently γ, would be significantly lower if the participants got more practice with the scheme before their results was recorded. Each participant averaged 33 samples each, so it is not feasible to e.g. calculate the average after removing the 20 first calculations from each participant, as this would leave very few trials. The important goal of the experiment is though not to find an accurate average calculation time, but to verify that $\gamma_H \leq 1$ is achievable. It seems that the average most likely is somewhere between 9 and 11 seconds, depending on how long users have been using the scheme, which is not severely limiting.

Figure 5.4 and table 5.2 shows the 7 participants with the most samples, the same effect can be observed, all except from one participant have a downward going trend (see SCT in table 5.2).

SMP	MCT	SDCT	SCT	MFR
72	9.02	2.38	−0.015	0.0695
36	10.06	3.24	0.038	0.1944
72	9.62	2.79	−0.373	0.0278
45	10.33	3.33	−0.115	0.044
45	10.06	3.31	−0.068	0.089
36	9.89	3.34	−0.143	0.083
45	9.88	3.33	−0.023	0.022

Table 5.2: Table of the 7 best participants. SMP=sample size, MCT=mean calculation time, SDCT=standard deviation of calculation time, SCT=slope calculation time, MFR=mean failure rate.

5.5.2 Failure rate.

Failure rate is a key characteristic of the scheme. As mentioned it is essential that a user is able to calculate passwords correctly for the scheme to function. How high the failure rate can be depends on how long passwords are used and how often a user can enter the wrong password without locking the account. A typical password protected site will allow a minimum of three strikes before the account is locked [BS03]. It is thus important that the probability of being "locked out" is small enough for it not to happen often, how often is of course subject to discussion.

Observation 5.1.
Average failure rate for all samples recorded in the experiment was measured to be

$\tilde{\lambda} = 0.0584795321637$, approximately every one out of 17 single digit challenge was calculated wrong.

This observation might not seem very unusual, but since a password consists of a sequence of calculations, it is required that users calculate a given number of challenges consecutively without failure. It is thus more interesting to evaluate the probability of having at least one mistake in a complete password calculation sequence. It is also not unusual to enforce a password policy using a three-strike policy which locks the account if three consecutive mistakes are made. Next, these probabilities are presented and discussed.

The probability of having at least one mistake in a length l password given the failure rate λ is given by

$$P(fail) = 1 - (1 - \lambda)^l \tag{5.1}$$

Next, the probability of getting the account locked given a three-strike policy with the same password and failure rate is

$$P(lock) = (1 - (1 - \lambda)^l)^3 \tag{5.2}$$

Password length l	3	5	10	15
$P(fail)$	0.1654	0.2601	0.4526	0.5959
$P(lock)$	0.0045	0.0176	0.0927	0.2107

Table 5.3: Probability of having at least one mistake in a length l password given failure rate $\lambda = 0.0584795321637$ for each single digit challenge.

Observation 5.3 shows the probabilities of calculating a password wrong once and three times in a row. If users want to use a password of 15 characters, which is reasonable to assume since the scheme only produce digits, they will compute a faulty password nearly 60% of the time. This limits the usability severely, since users will more often than not, be unable to log in.

The probability of actually locking the account by miscalculating three passwords in a row, is lower but not significantly. With the same failure rate and password length, users would break the three-strike rule approximately one out of five login attempts.

To illustrate this consequence in a extreme case, consider the *very active* user from table 3.2 in section 3.3 who visits 10 different accounts every day. Such an user would eventually lock two accounts every day with password lengths of 15 characters, which of course is not acceptable.

As discussed in section 3.3.3 (and as implemented in chapter 4), the password length needed for different accounts may vary. By using shorter password for noncritical accounts, and only generating long passwords for the most critical, the average password length will be much lower than in the examples above. The scheme could then be tweaked by users to fit specific needs, instead of requiring 10 or 15 characters in all passwords generated. For less sensitive accounts users may have passwords of lengths 5, which would make for a mistake in approximately every 4th password, and only locking an account every 57th login attempt. This way users can calculate passwords with significantly less effort and lower failure rates.

The experiment did not find any significant correlation between failure rates and calculation times. Users calculating fast do not have a higher failure rate than slower users, which some might have expected.

Remark 5.3.
The author remarks that the findings related to failure rates might be too harsh since the participants was asked to calculate the challenges "as fast as possible", which might be the wrong approach. In a real world scenario it would be more important to calculate correctly. Though, the results clearly show that the failure rate is a relevant attribute worth investigating closer, as it might limit the reliability of the scheme in real usecases.

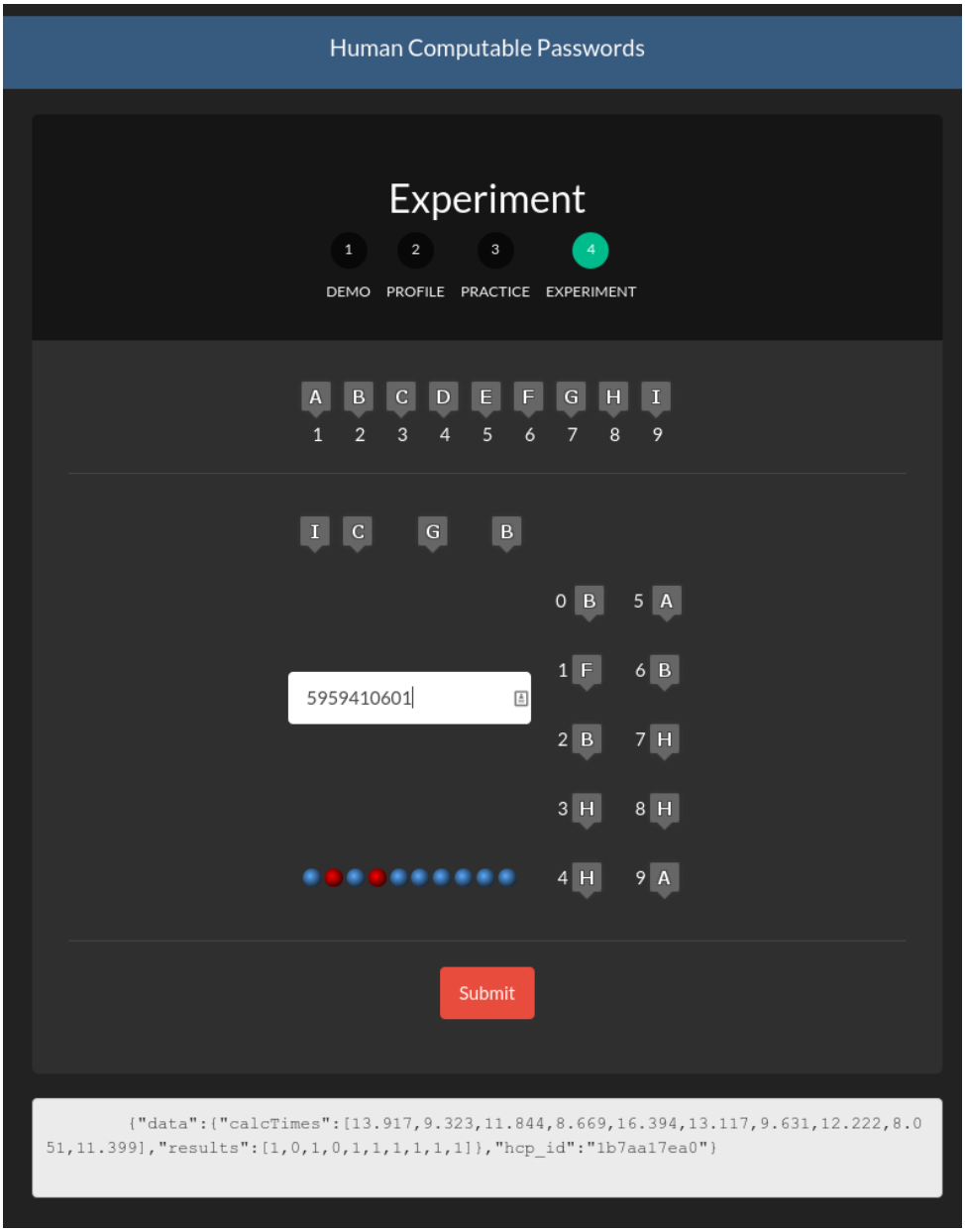


Figure 5.1: The experiment screen seen after completing an experiment sample, ready to be submitted.

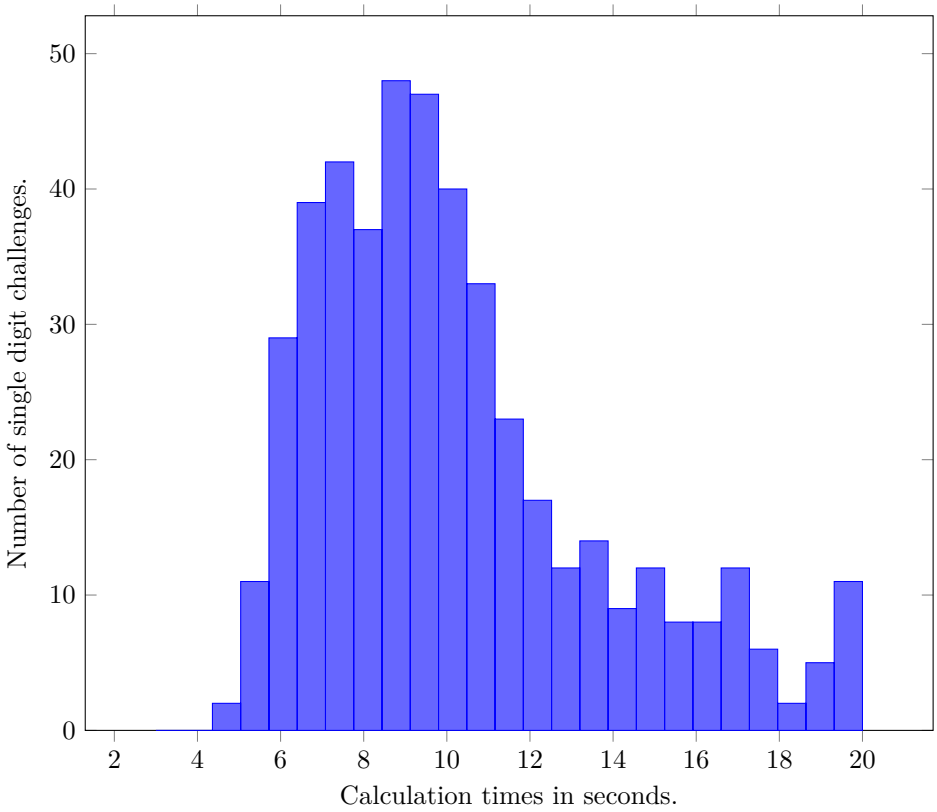


Figure 5.2: Histogram showing the distribution of calculation times of all the recorded experiments. Sample size 467 single digit challenges.

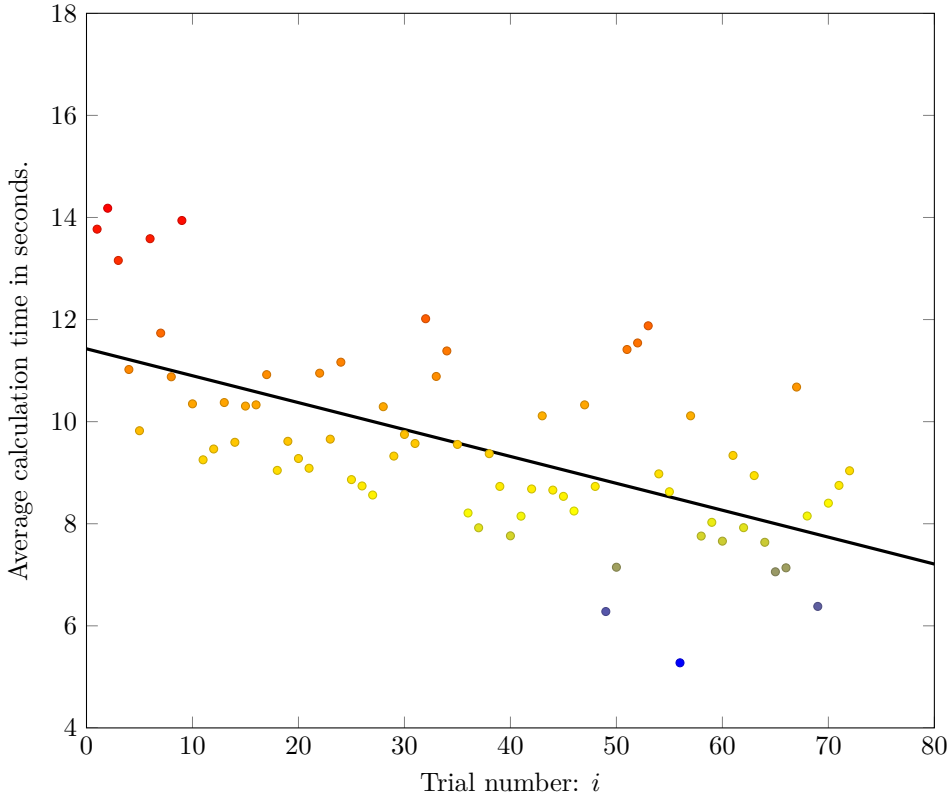


Figure 5.3: Average calculation time of all participants' $i'th$ calculation sample, and the regression line of the averages. Sample size 467, average samples per participant 33.

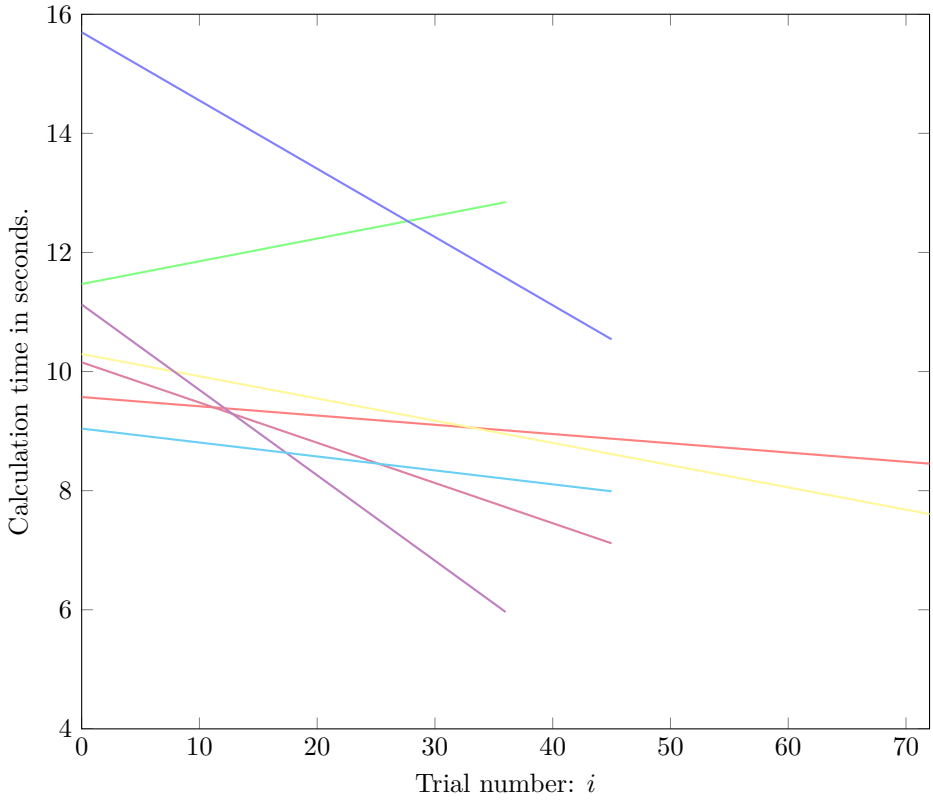


Figure 5.4: Regression lines for the 7 participants with the most samples. A clear downward sloping trend in terms of calculation time is observed.

Chapter 6
Concluding Remarks and Further Work

This project has presented the human computable password management scheme by Blocki et al. [BBD14], as well as the design and construction of two applications implementing the HCP scheme with different purposes. How different characteristics and parameters affect the usability of the scheme was discussed, as well as how password length and number of secret mappings affect the security. Failure rate was introduced as an important factor affecting the usability of the scheme.

6.1 Application and Usability Remarks.

The first application is a Google Chrome browser extension, making it easy for users to employ the scheme without too much trouble. It is a fact that the scheme is quite complex and requires a lot dedication from the users for it to work. It is thus important to have a tool minimizing extra work required, including management and generation of challenges. Without a tool to take care of these obstacles, the scheme would be very hard to use in practice. With the application created in this project, users only have to worry about memorizing and rehearsing the secret mapping, all the overhead related to generation, storing and fetching of challenges is handled by the extension. The one choice users have to make is what category to put their accounts in, either low, medium or high sensitivity. This measure was introduced to increase the usability by lowering the average number of single digit challenges, while still keeping the important accounts secured.

The application is available in the browser, making it very accessible in situations where passwords are needed. The extension monitors the active page browsed by the users and fetches information from DOM allowing it to display the correct challenge at all times, both in regards to which page users are visiting and how many characters are entered in the password field. E.g. if users visit google.com the extension will get notice about this and bring up the challenges associated with google.com. If the site is not part of the system, users will get the chance to add

it. When calculating passwords, the challenge on display updates for each entered character in the password field.

Blocki et al. [BBD14] define usability as a combination of calculation time, effort spent memorizing σ and extra rehearsal in addition to natural usage. This project has introduced failure rate as another influential factor. It is apparent that it does not matter how fast users calculate if the calculations are not correct. It can also be observed in the experiment that failures do happen, and thus should be part of the usability analysis.

6.2 Experiment and Findings.

The second application is a web application functioning as a demonstration platform as well as an experiment, gathering usage data. The experiment gathers data related to calculation time and failure rates. It was created to investigate how the scheme performs, with usability characteristics in focus. The participants in the experiment get a short introduction and the chance to practice until fairly familiar with the scheme. They are then asked to calculate a complete password consisting of 10 single digit challenges, while getting timed.

The experiment measured a average calculation time of 10.296 seconds, with a standard deviation of 3.64. This is considered to be reasonably good, as it should not limit the usability of the scheme severely. It could also be observed that users improved over time so that average calculation time would probably be even lower after some practice.

A more unanticipated result was the failure rate, which was measured to be 0.058, meaning that every 17 single digit challenge would be calculated wrong. This result should not be interpreted as a dismissal of the scheme, since different users will achieve different failure rates, and many will be able to calculate correctly most of the time. What should be emphasized though, is that the failure rate should be investigated closer, to verify that a general user average a low enough failure rate.

6.3 Further Work

6.3.1 Additional Applications.

This project showed how browser extensions can be used to create an easy to use password management application. This choice was made with a goal of making a quite complex scheme usable without too much overhead and practice. A possible way to make the scheme even more accessible, would be by creating an accompanying web application and possibly a mobile application. These applications could use

the same storage, while still syncing to chrome storage for the browser extension and to mobile storage for the mobile application. It would be useful to have a user management page to overview user data and challenges. The construction presented here does not allow users to manually configure the account which some users might dislike.

6.3.2 Larger Scale Experiments

It is, as mentioned, clear that the failure rate of the scheme is important for it to function in practice. The experiment conducted in this project did not aim to verify a hypothesis, which should be the next step. A typical setup would could be similar to the one presented here, but in a larger scale, possibly using a crowd sourcing service or social media, asking the participants to calculate challenges. It would be important to get more samples from each participant, making it easier to calculate different averages after more practice.

It would also be interesting to conduct a survey gathering data about how users rate their account. Typically asking how many they regard as “highly sensitive”, “don’t-care” etc. This would make it possible calculate an average password length which again could be used to measure failure rates and calculation times more thoroughly. By knowing how many accounts users have of each type, it would be possible to say something about how many mappings would be sufficient to have in σ.

References

[ABK13] Tolga Acar, Mira Belenkiy, and Alptekin Küpçü. Single password authentication. *Computer Networks*, 57(13):2597–2614, 2013.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[AMV12] Syed Akram, Mohammed Misbahuddin, and G Varaprasad. A Usable and Secure Two-Factor Authentication Scheme. *Information Security Journal: A Global Perspective*, 21(4):169–182, 2012.

[Bad97] Alan D Baddeley. *Human memory: Theory and practice*. Psychology Press, 1997.

[BBD13] Jeremiah Blocki, Manuel Blum, and Anupam Datta. LNCS 8270 - Naturally Rehearsing Passwords. pages 361–380, 2013.

[BBD14] Jeremiah Blocki, Manuel Blum, and Anupam Datta. Human Computable Passwords. *CoRR*, abs/1404.0, 2014.

[BCJ+14] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, and Yuan Tian. Analyzing the dangers posed by Chrome extensions. In *IEEE Conference on Communications and Network Security, CNS 2014, San Francisco, CA, USA, October 29-31, 2014*, pages 184–192, 2014.

[BFSB10] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[Blo14] Jeremiah Blocki. *Usable Human Authentication: A Quantitative Treatment*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2014.

[Bra14] Rodrigo Branas. *AngularJS Essentials*. Packt Publishing Ltd, 2014.

[Bro11] Gerald Brose. Rainbow Tables. In Henk C A van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1021–1022. Springer, 2011.

[BS03] Sacha Brostoff and M Angela Sasse. “Ten strikes and you’re out”: Increasing the number of login attempts can improve password usability. 2003.

[BSNP96] Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk. Keyed hash functions. In *Cryptography: Policy and Algorithms*, pages 201–214. Springer, 1996.

[BZW13] A Barua, M Zulkernine, and K Weldemariam. Protecting Web Browser Extensions from JavaScript Injection Attacks. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 188–197, 2013.

[CFW12] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *USENIX Security Symposium*, pages 97–111, 2012.

[Dea09] John Deacon. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[DG09] Mohan Dhawan and Vinod Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009*, pages 382–391, 2009.

[dM14] Xavier de Carné de Carnavalet and Mohammad Mannan. From Very Weak to Very Strong: Analyzing Password-Strength Meters. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*. The Internet Society, 2014.

[DMR09] Matteo Dell’Amico, Pietro Michiardi, and Yves Roudier. Measuring Password Strength: An Empirical Analysis. *CoRR*, abs/0907.3, 2009.

[FH07] Dinei A F Florêncio and Cormac Herley. A large-scale study of web password habits. In Carey L Williamson, Mary Ellen Zurko, Peter F Patel-Schneider, and Prashant J Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, {WWW} 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 657–666. ACM, 2007.

[FHC07] Dinei Florêncio, Cormac Herley, and Baris Coskun. Do Strong Web Passwords Accomplish Anything? In *2nd USENIX Workshop on Hot Topics in Security, HotSec’07, Boston, MA, USA, August 7, 2007*. USENIX Association, 2007.

[FHV14] Dinei Florêncio, Cormac Herley, and Paul C Van Oorschot. An administrator’s guide to internet password research. In *Proc. USENIX LISA*, 2014.

[GF06] Shirley Gaw and Edward W Felten. Password management strategies for online accounts. In *Proceedings of the 2nd Symposium on Usable Privacy and Security, SOUPS 2006, Pittsburgh, Pennsylvania, USA, July 12-14, 2006*, pages 44–55, 2006.

[HD79] Frederick Hartwig and Brian E Dearing. *Exploratory data analysis*. Sage Publications Beverly Hills, 1979.

[IWS04] Blake Ives, Kenneth R Walsh, and Helmut Schneider. The domino effect of password reuse. *Commun. ACM*, 47(4):75–78, 2004.

[LLV08] Mike Ter Louw, Jin Soon Lim, and V N Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[LZC+] Lei Liu, Xinwen Zhang, Vuclip Inc Huawei R&D Center, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures.

[Mil56] George A Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[SBSB07] Richard Shay, Abhilasha Bhargav-Spantzel, and Elisa Bertino. Password Policy Simulation and Analysis. In *Proceedings of the 2007 ACM Workshop on Digital Identity Management*, DIM '07, pages 1–10, New York, NY, USA, 2007. ACM.

[SHB09] Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric cryptography in javascript. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 373–381. IEEE, 2009.

[SJB+14] David Silver, Suman Jana, Dan Boneh, Eric Yawei Chen, and Collin Jackson. Password Managers: Attacks and Defenses. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 449–464. USENIX Association, 2014.

[Squ89] Larry R Squire. On the course of forgetting in very long-term memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(2):241, 1989.

[SS09] Karen Scarfone and Murugiah Souppaya. Guide to enterprise password management (draft). *NIST Special Publication*, 800:118, 2009.

[Tuk77] John W Tukey. Exploratory data analysis. *Reading, Ma*, 231:32, 1977.

[UKK+12] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 65–80, 2012.

[vABL04] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, 2004.

[YBAG00] Jianxin Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. The memorability and security of passwords: some empirical results. *Technical Report-University Of Cambridge Computer Laboratory*, page 1, 2000.

[ZYS13] Rui Zhao, Chuan Yue, and Kun Sun. A Security Analysis of Two Commercial Browser and Cloud Based Password Managers. In *Social Computing (SocialCom), 2013 International Conference on*, pages 448–453. IEEE, 2013.

Appendix A
Extension Class Files

A.1 Content Script

```
1  window.onload = setTimeout(updateUrl(),500);
2  var passwords = getPwdInputs();
3  var pwField = passwords[0];
4
5  if(pwField){
6      pwField.addEventListener('input',
7      function (callback){
8
9          console.log("input content: " + pwField.value.length);
10         chrome.runtime.sendMessage({pwValue: pwField.value.length+1},
11         function(respons){
12             console.log("Respons pw changed: " + respons);
13         });
14
15     });
16 }
17
18 function getPwdInputs() {
19       var ary = [];
20         var inputs = document.getElementsByTagName("input");
21         for (var i=0; i<inputs.length; i++) {
22             if (inputs[i].type.toLowerCase() === "password") {
23                       ary.push(inputs[i]);
24             }
25         }
26           return ary;
27 }
28
29 function updateUrl(){
30     chrome.runtime.sendMessage({newUrl: window.location.hostname},
31     function(respons){
32         console.log("URL changed to: "+ window.location.hostname);
33     });
34 }
```

Listing A.1: Content script file.

The content script listens for the onload event triggered by the window object when a new page is loaded. When receiving this event the updateUrl function(line 29) is called, which sends an update containing *window.location.hostname* which is

the hostname of the currently active page. Hostname is used since login forms may be located at different locations at different domains.

Next the script searches the DOM for input fields of type "password" using the *getPwdInputs* function(line 18). This function iterates through all the input fields looking for password fields. If a password field is found, an event listener is attached to the field, listening for events of type "input" which are sent when the field changes[1]. When the password field changes a message containing the new length of the password is sent to the controller.

A.2 Controller

```
1  angular.module('human-computable-pws.controllers', [])
2  .controller("MainController",
3      function($timeout, $scope, chromeStorage){
4      //initial values
5      $scope.user = "";
6      $scope.pw=0;
7      $scope.selectedSite=null;
8      $scope.siteClass ='10';
9
10     //function adding new site to the user
11     $scope.newSite = function(){
12         var newSite = new Site($scope.url, $scope.siteClass);
13         $scope.user.sites.push(newSite);
14         $scope.selectedSite = newSite;
15         chromeStorage.set("user", $scope.user);
16         $scope.newSiteButton = false;
17     }
18
19     //try to load the user object from storage,
20     //if no user is present, create new.
21     try{
22         chromeStorage.getOrElse("user",
23         function(){
24             var newUser = {
25                 name: "user",
26                 sites: [
27                     new Site("accounts.google.com"),
28                     ]
29             };
30             chromeStorage.set("user", newUser);
31             return newUser;
32         }).then(function(keyValue){
33             $scope.user = keyValue;
34         });
35     }
36     catch(err){
37         console.log("Not run as extension")
38     }
39
40     //receive messages sent from the content script
41     //if url or password length changes
42     var tmp;
43     function handleMessage(){
44         if(tmp.pwValue){
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

```
45              $scope.pw = tmp.pwValue-1;
46              console.log("pw: %o", $scope.pw);
47              console.log("ombjects:   %o", $scope.user)
48          }
49          if(tmp.newUrl){
50              $scope.url = tmp.newUrl;
51              var site = searchList($scope.url, $scope.user.sites);
52              if(!site){
53                  $scope.newSiteButton=true;
54                  $scope.selectedSite = null ;
55              }else{
56                  $scope.newSiteButton=false;
57                  $scope.selectedSite = site;
58              }
59          }
60      }
61      //add eventhandler listening for messages from content script
62      try{
63          chrome.runtime.onMessage.addListener(
64          function(message,sender){
65              tmp = message;
66              $timeout(handleMessage);
67          });
68      }
69      catch(err){
70          console.log("not extension")
71      }
72  });
```

Listing A.2: Angular controller.

The controller is responsible for all the business logic, keeping the storage updated with new users and new sites. The *newSite* method on line 11 is called when then "new site"-button is clicked by the user, it then generates a new object of type Site with the domain name and selected site class (essentially the number of single digit challenges to be generated). The site is then added to the user object which is stored in the database to keep the storage persistent in case of disconnection.

When the controller is loaded the first action executed is trying to load the user object from the storage (line 21), if no user object is found, the controller creates a new one. In the code presented here, a new user is initiated with a dummy site for demonstration purposes. All the storage specific methods are called from the *chromeStorage* object, which makes it easy to do operations accessing the Chrome storage. The *getOrElse* is used, which checks for a user object and creates a new if none is present.

The controller also listens for messages from the content script, this event listener can be seen at line 62. When a message is received the *handleMessage* is called to process the data, distinguishing between a changes in the URL and changes in password length. When data is received, the corresponding variable in the controller is updated. If a new site was added, the view is also updated, hiding the "new

site"-button.

A.3 App.js File

```
var hcp = angular.module('human-computable-pws', [
    'human-computable-pws.controllers',
    'ngAnimate',
    'chromeStorage',
    'ngRoute',
    ]);

hcp.config( ['$compileProvider',
    function( $compileProvider  ) {
        var currentImgSrcSanitizationWhitelist =
            $compileProvider.imgSrcSanitizationWhitelist();
        var newImgSrcSanitizationWhiteList =
            currentImgSrcSanitizationWhitelist.toString().slice(0,-1)
            + '|chrome-extension:' +currentImgSrcSanitizationWhitelist.
            toString().slice(-1);

        console.log("Changing imgSrcSanitizationWhiteList from "+
            currentImgSrcSanitizationWhitelist+" to "+
            newImgSrcSanitizationWhiteList);
        $compileProvider.imgSrcSanitizationWhitelist(
            newImgSrcSanitizationWhiteList);
    }
]);

hcp.config(['$routeProvider',
function($routeProvider){
    $routeProvider.
        when('/',{
            templateUrl: 'partials/content.html',
            controller: 'MainController'
        }).
        otherwise({
            redirectTo: '/'
        });
}]);

function searchList(site, sites){
    if(sites == undefined) return false;
    for(var i = 0; i<sites.length; i++){
        if(sites[i].name == site){
            return sites[i];
        }
    }
    return false;
}



function Site(name, siteclass){
    this.name = name;
    this.challenges = [];

    for(var j=0; j<siteclass; j++){
        this.challenges.push(randomChallenge());
    }
}

```

```
58 function getRandInt(){
59     var intArr = new Uint8Array(1);
60     window.crypto.getRandomValues(intArr);
61     if(intArr[0]>=Math.floor(256/26) * 26)
62         return getRandInt();
63     return (intArr[0] % 26);
64 }
65
66 function randomChallenge(){
67     var letters = "abcdefghijklmnopqrstuvwxyz";
68     var ch = [];
69     for(var z=0; z<14; z++){
70         ch.push(letters.charAt(getRandInt()));
71     }
72     return ch;
73 }
```

Listing A.3: Angular launcher file.

The app.js file initiates the application, specifying the modules used, as well as some helper functions, including the function responsible for generating the random challenges(line 66).

A.4 View File (partial file)

```
1  <div class="container-fluid">
2  <div id="form-container">
3
4      <hr class="divider">
5
6      <div align="center">
7          <select ng-model="selectedSite"
8                  ng-options="site.name for site in user.sites"
9                  class="dropDownColor"  >
10         </select>
11     </div>
12     <hr class="divider" style="margin-bottom: 0;">
13     <div id="challenges">
14     <div align ="center" class="newSiteButton" ng-show="newSiteButton">
15         <div class="infoBox" >
16             The current site does not have challenges stored in the system.
17             <hr>
18         <form name="siteClass" class="siteClass">
19                 <label for="radio1">Low
20                     <input type="radio"
21                            ng-model="siteClass"
22                            value=6
23                            id="radio1">
24                 </label>
25                 <label for="radio2">Med
26                     <input type="radio"
27                            ng-model="siteClass"
28                            value=10
29                            id="radio2">
30                 </label>
31         <label for="radio3">High
32                     <input type="radio"
33                            ng-model="siteClass"
34                            value=15
```

```
35                              id="radio3">
36                  </label>
37              </form>
38              <button ng-click="newSite()"
39                      class="btn btn-info"
40                      ng-show="true" >Add site</button>
41          </div>
42      </div>
43  <div ng-hide="newSiteButton">
44  <table
45      align="center" class="challenges"
46      ng-init="site = user.sites[user.sites.indexOf(selectedSite)]">
47      <tr>
48          <td>
49              <img ng-src="letters/letter_{{site.challenges[pw][10]}}.png"/>
50              <img ng-src="letters/letter_{{site.challenges[pw][11]}}.png"/>
51          </td>
52          <td>
53              <img ng-src="letters/letter_{{site.challenges[pw][12]}}.png"/>
54          </td>
55          <td>
56              <img ng-src="letters/letter_{{site.challenges[pw][13]}}.png"/>
57          </td>
58          <td></td>
59      </tr>
60  </table>
61  <hr class="divider" style="margin: 0;">
62  <table align="center" class="challenges">
63      <tr>
64          <td>
65              0<img ng-src="letters/letter_{{site.challenges[pw][0]}}.png"/>
66          </td>
67          <td>
68              5<img ng-src="letters/letter_{{site.challenges[pw][5]}}.png"/>
69          </td>
70      </tr>
71      <tr>
72          <td>
73              1<img ng-src="letters/letter_{{site.challenges[pw][1]}}.png"/>
74          </td>
75          <td>
76              6<img ng-src="letters/letter_{{site.challenges[pw][6]}}.png"/>
77          </td>
78      </tr>
79      <tr>
80          <td>
```

Listing A.4: Angular view. Only included the table showing the challenges for readability.

The code shown in this listing contains the view where the challenges are presented, as well as the new site button and site classification radio buttons. Notice the *ng-show* directive on line 14, this decides if the "new site"-functionality should be shown or not. If *newSiteButton* is true it will appearer and vice versa, the same goes for the challenges which starts on line 34 with a similar directive, *ng-hide*, which hides the challenges div if new site is visible.

The challenges div is essentially a table containing an element of the challenge in each cell. Notice the double bracket notation, where each of the challenge objects are referred. *{{site.challenges[pw][i]}}* refers to the *$scope.site.challenges* which contains the challenges associated with the current active page. *pw* is at any given time the length of the password field of the active web page, if *pw* changes the challenge on display will automatically change through the two-way data binding. This way, the correct challenge will always be the one seen by the user.

Appendix B
Demonstration Slides

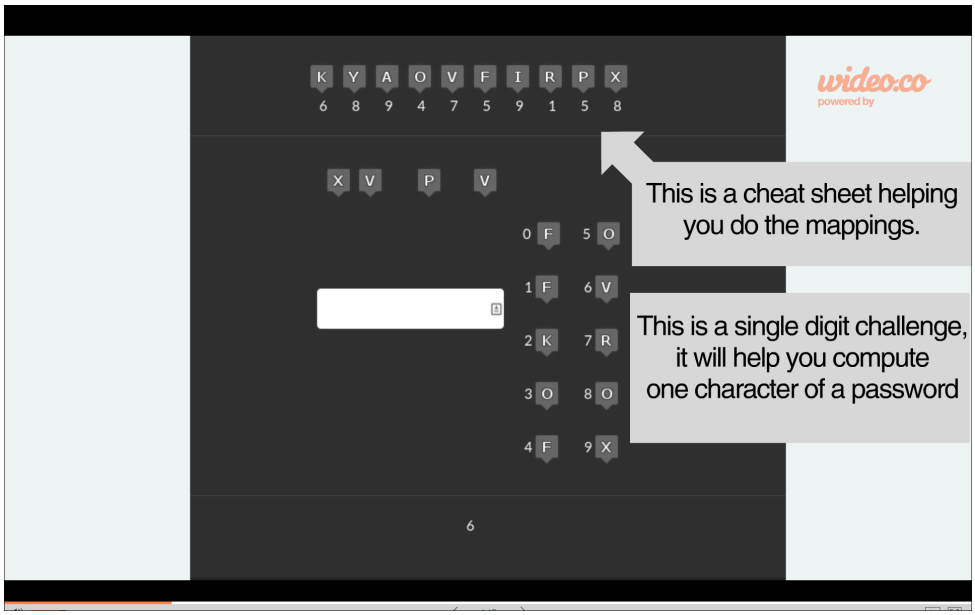


Figure B.1: Demo slide 1.

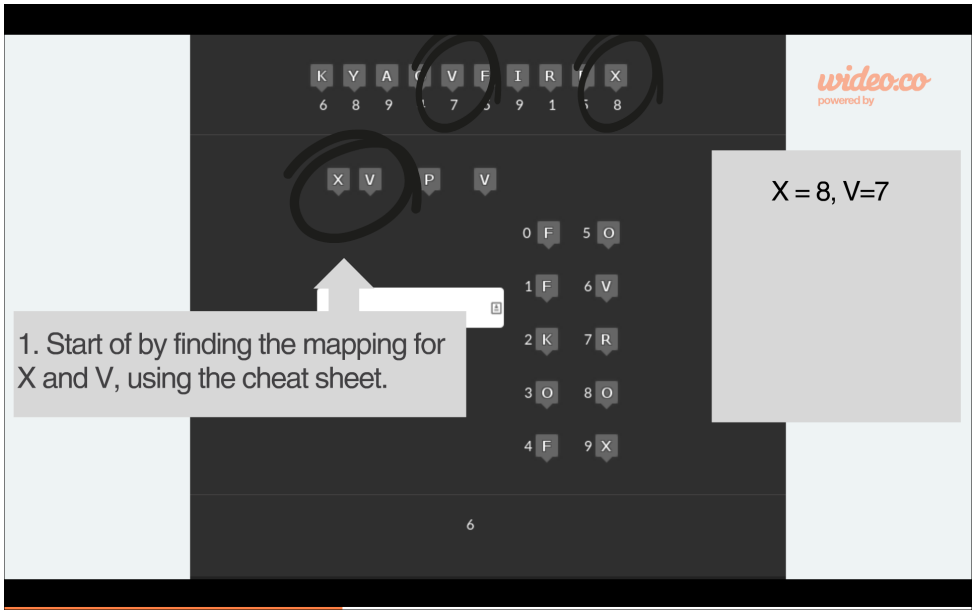


Figure B.2: Demo slide 2.

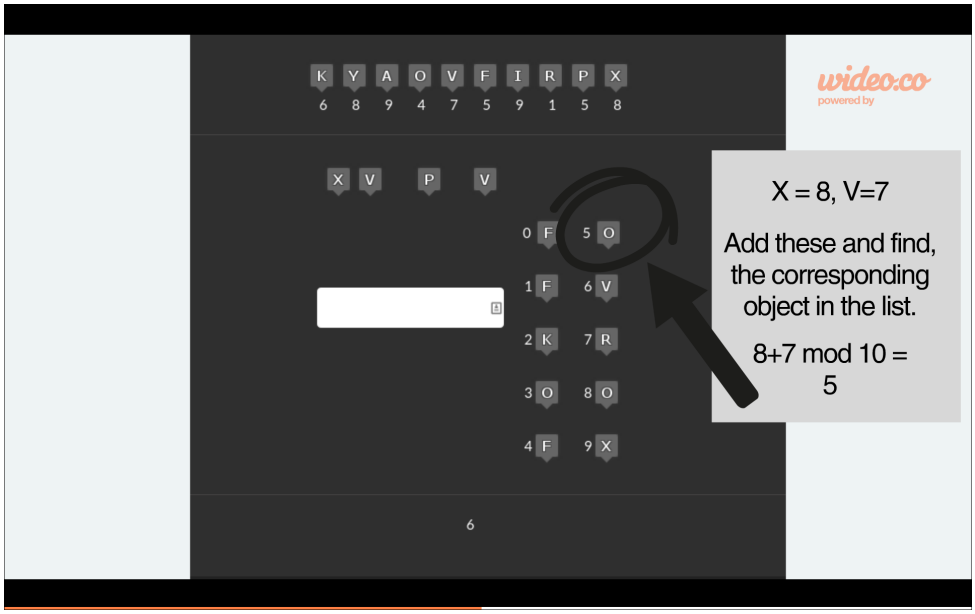


Figure B.3: Demo slide 3.

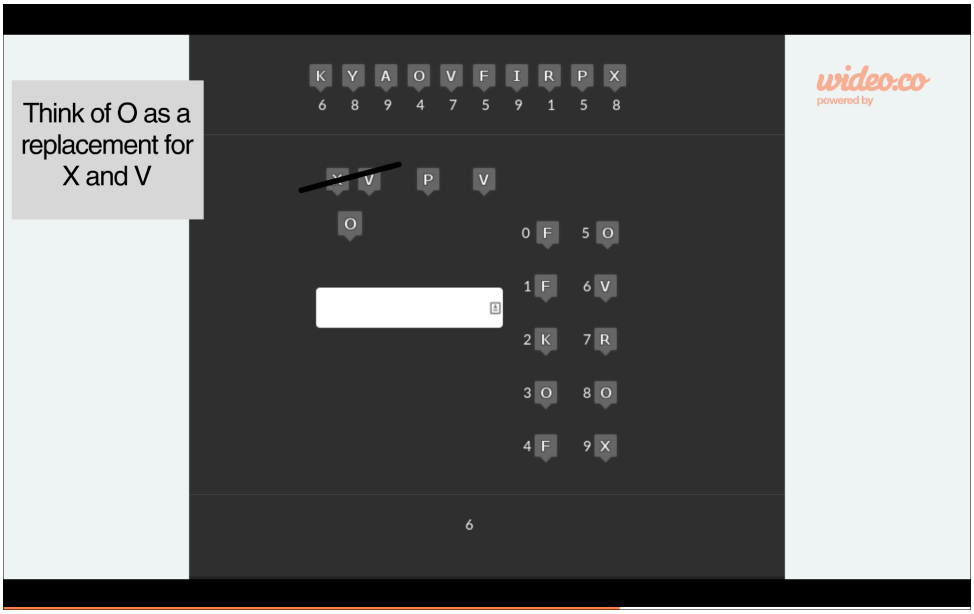


Figure B.4: Demo slide 4.

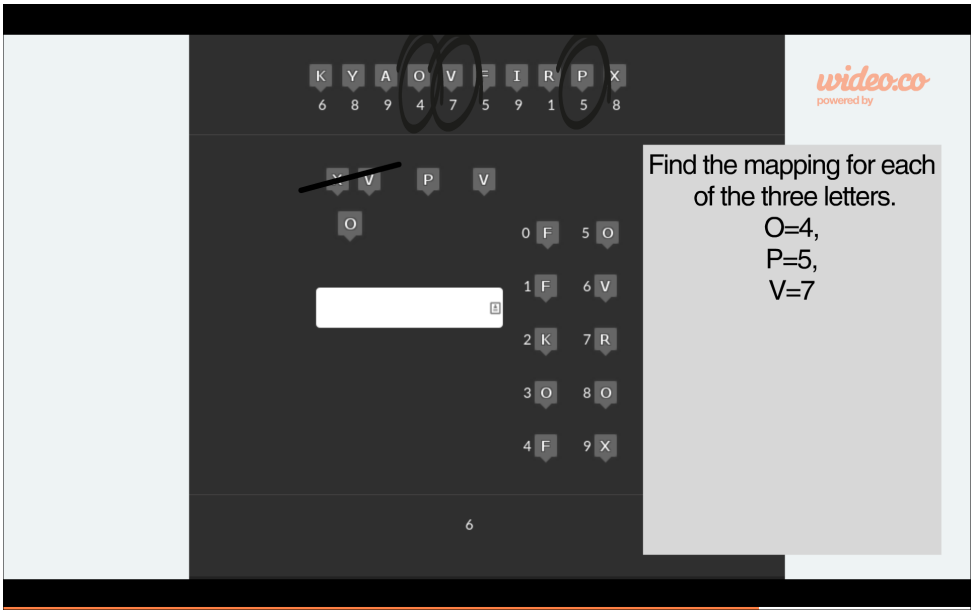


Figure B.5: Demo slide 5.

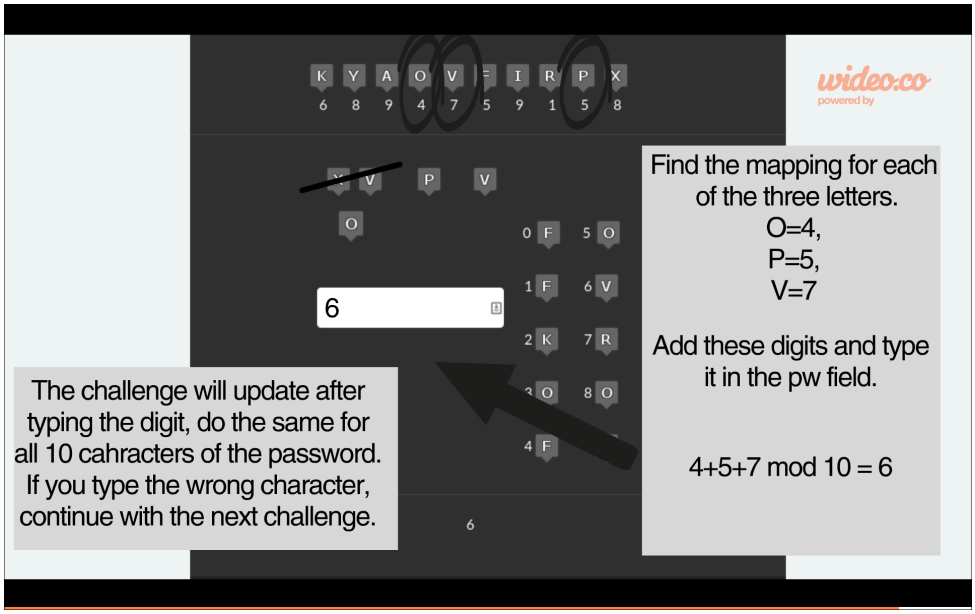


Figure B.6: Demo slide 6.

Appendix C

Experiment Application

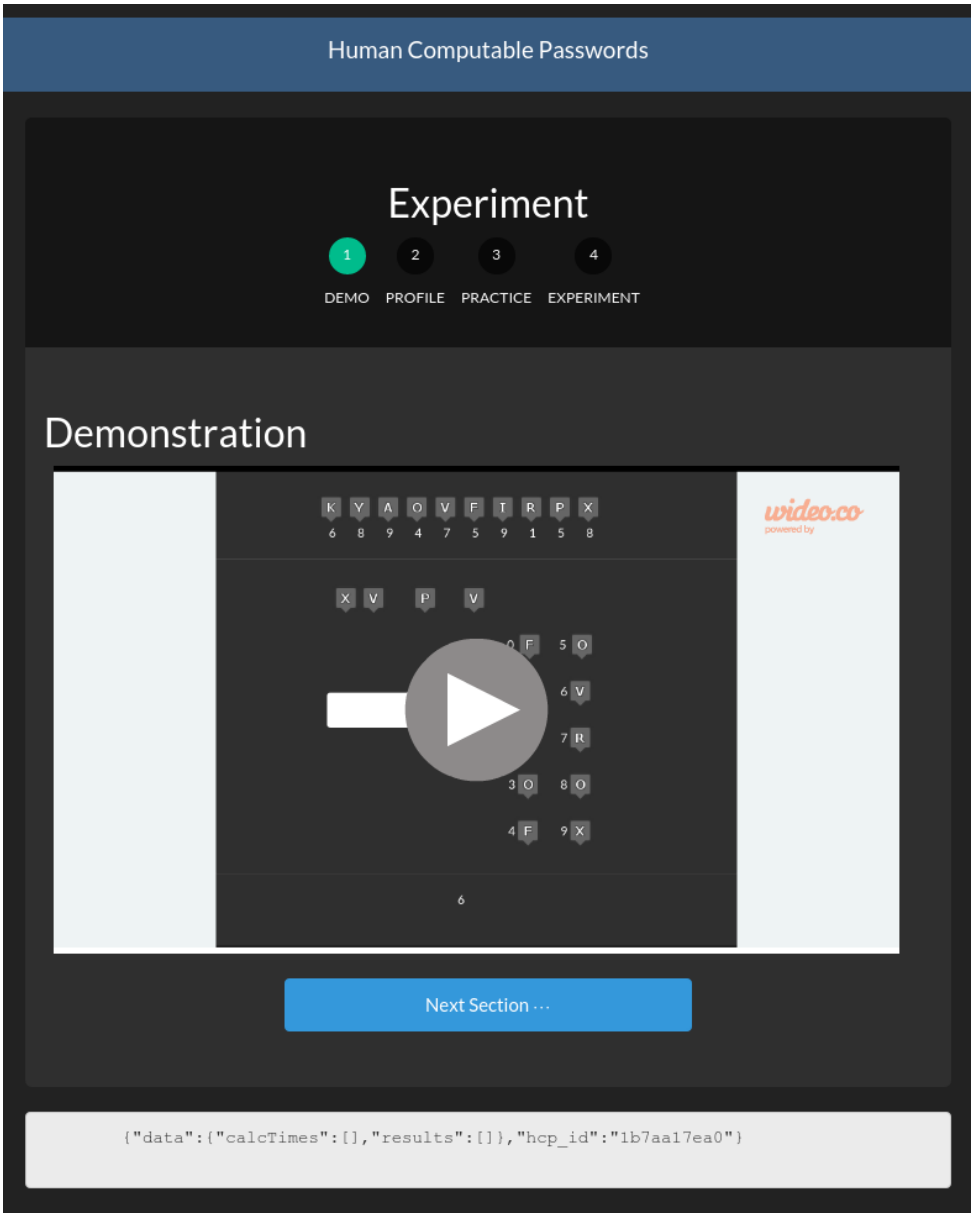


Figure C.1: First view shown to the user, containing a demonstration video. Created using wideo.co.

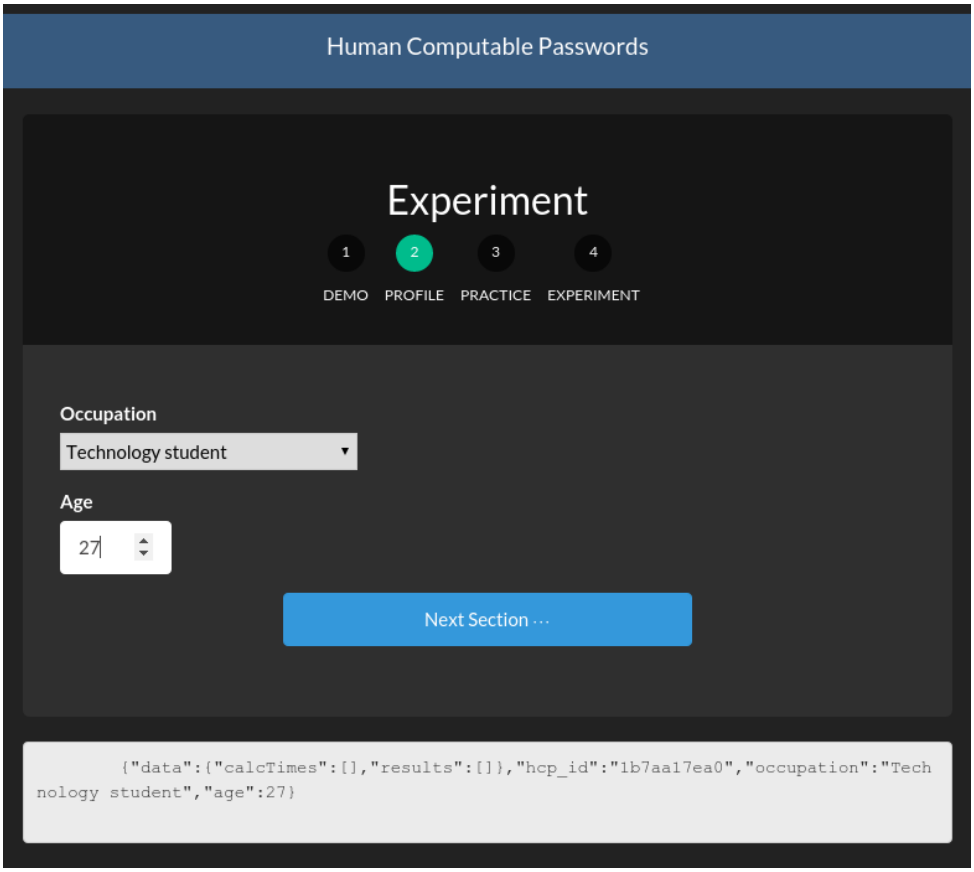


Figure C.2: Second view shown to the user, gathering some basic demographic data which might be relevant.

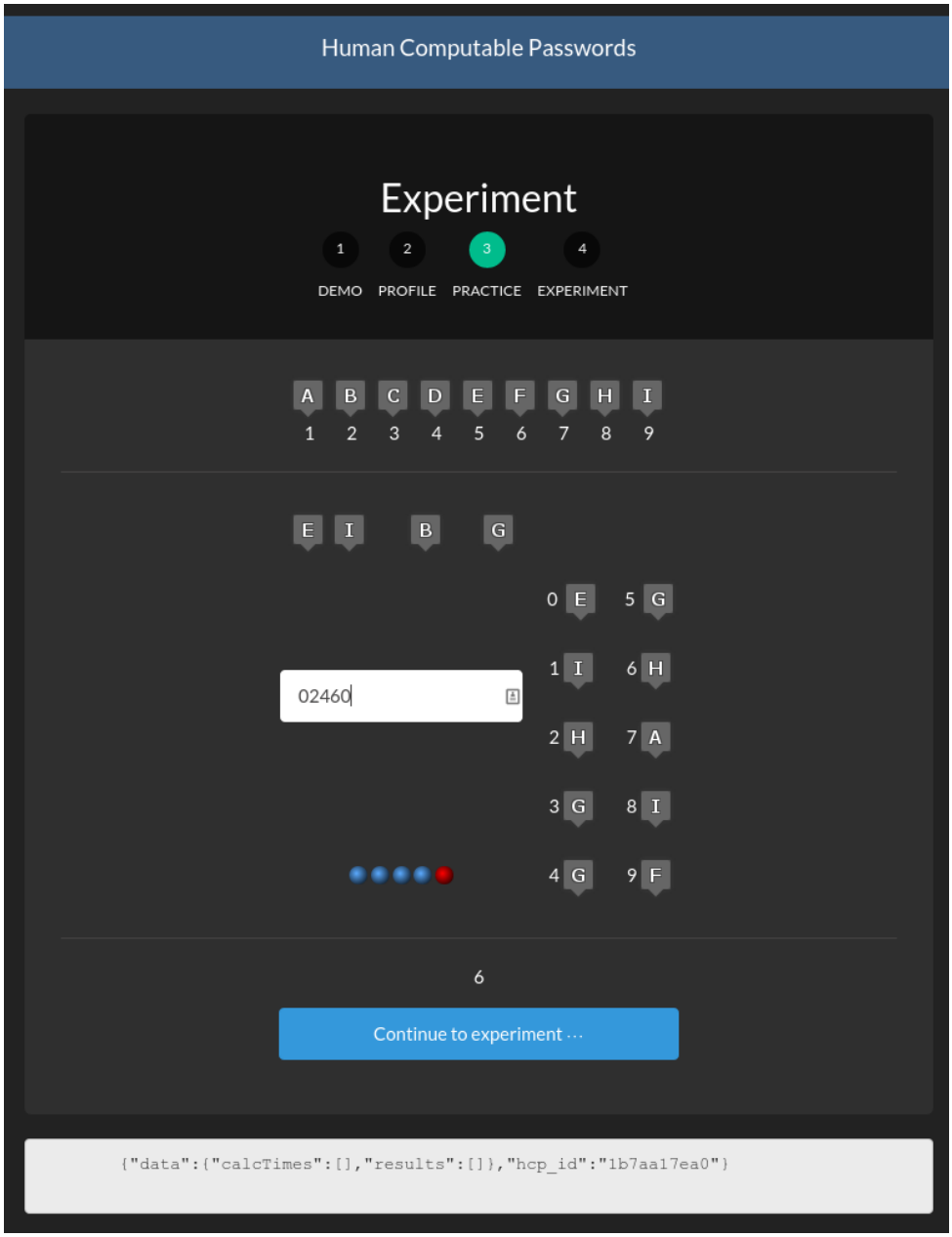


Figure C.3: Third view of the web application. Practice section used by the user before entering the actual experiment.

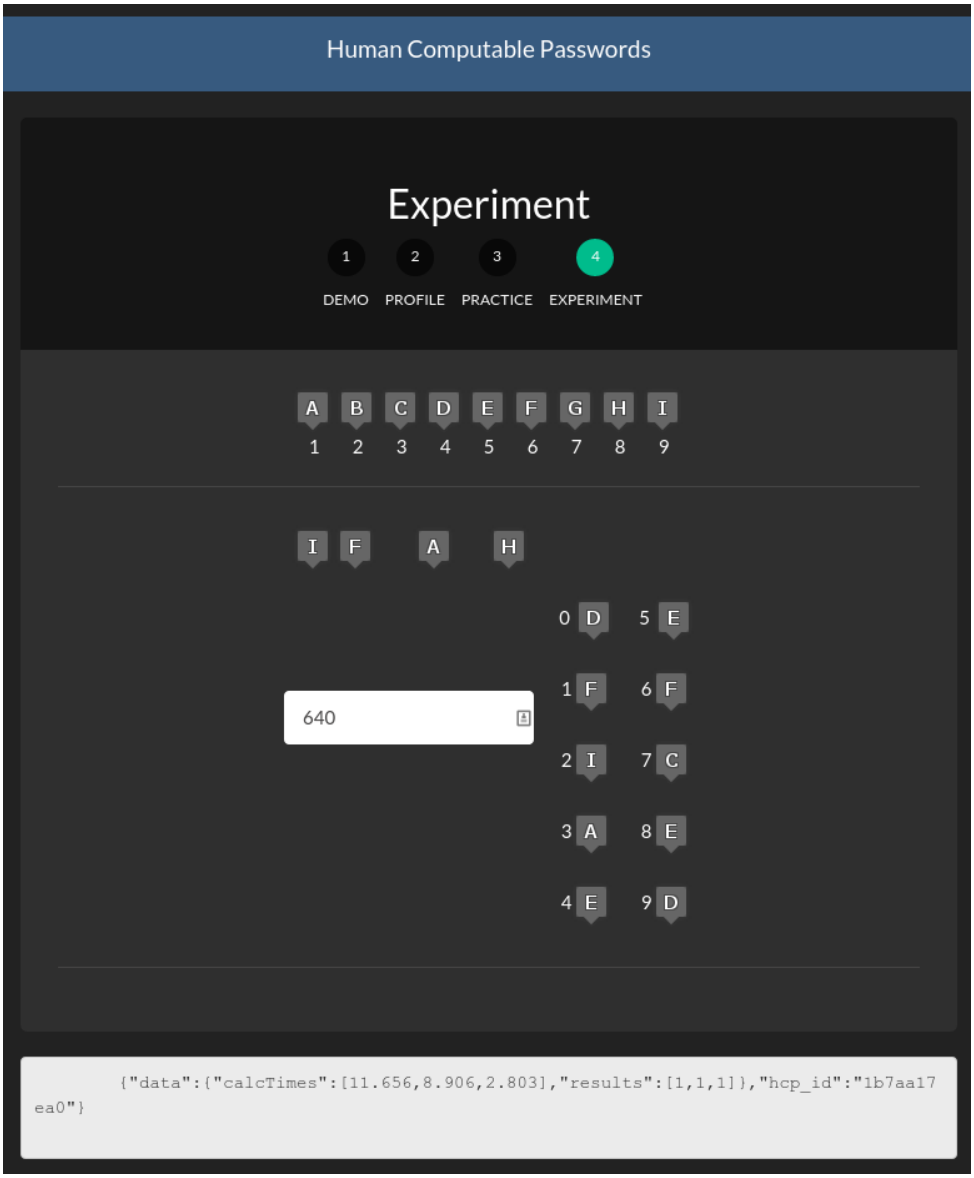


Figure C.4: Final view, containing the experiment form. The user will calculate the response to the challenge on display and enter the answer in the password field. When finished the results can be submitted. And eventually stored in the database.