A Peer-Based Distributed Password Manager

Andreas Stensig Jensen

Kongens Lyngby 2020

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract (English)

This master's thesis explore the use of data decentralization to create a password manager in a peer-to-peer network, with the purpose of strengthening the password manager's resistance to passive and active attacks; doing so through a set of requirements establishing a strict security model. It presents an analysis of the most popular password managers on the market, as well as theoretical solutions proposed in academic papers, to show that no existing solution implements a decentralized model that satisfy the requirements. It proposes a design and presents a prototype implementation that satisfy the strict security model and its requirements using a peer-based distributed system with decentralized data. It concludes with an evaluation of the implementation and reflection on the consequences of the design and how these may be improved in future works.

Abstract (Danish)

Denne kandidatafhandling udforsker brugen af data decentralisering, til at opbygge en adgangskode manager i et peer-to-peer netværk. Målet er at styrke adgangskode managerens modstand til passive og aktive angreb, ved at opsætte en række krav der etablere en striks sikkerhedsmodel. Afhandlingen præsenterer en analyse af de mest populære adgangskode managere på markedet, samt teoretiske løsninger fra akademiske journaler, for at vise en mangle på en løsning der anvender en decentraliseret model og opfylder kravene. Den præsenterer derefter et design og en prototype implementering, som opfylder den strikse sikkerhedsmodel og dens krav, ved brug af et peer-baseret distribueret system med data decentralisering. Afhandlingen slutter med en evaluering af implementeringen, og reflekterer på konsekvenserne af designet og hvordan det kan forbedres med videre fremtidig arbejde.

Preface

This thesis was prepared at the Technical University of Denmark (DTU), at the department of Applied Mathematics and Computer Science (Compute), in fulfillment of the requirements for acquiring a M.Sc. degree in Engineering.

The thesis deals with the security of password managers and how decentralizing data in a distributed system can from the basis of password manager with increased security. The reader is expected to have at least undergraduate-level software engineering knowledge, as a lot of terms, concepts, and principles are mentioned but not otherwise explained due to presumed familiarity.

The thesis consists of the definition of a strict requirements model, the analysis of the password manager domain—both industry solutions and academic models—comparing them to the requirements model, and the design and implementation of a new password manager system that satisfies these requirements.

Lyngby, 01-January-2020

Andreas Stensig Jensen

Acknowledgements

I would like to thank my supervisor Nicola Dragoni for his help, feedback, and guidance during the process of this project.

I would also like to thank my family and colleagues for their support and interest in the project's topic.

Finally, I would like to thank my girlfriend for her continued support and patience through these long months.

Contents

Chapter 1

Introduction

It is a common practice in most digital application nowadays to require the application user to authenticate themselves and prove their identity. Whether for need of not exposing secure data to unauthorized eyes, or hiding application features based on user privileges, these are just two of the many reasons applications may need user authentication.

By far the most widespread (almost universal) implementation of this authentication scheme is the use of unique credentials: a combination of some domain/application unique user ID and a secret password/phrase that only the user knows. Despite its popularity this simple method has shown to have many weaknesses, not least of which is the tendency for users to re-use passwords across many different applications, as shown in [Das+14]. This is an issue when applications are broken and user passwords are leaked, giving malicious attackers access to huge datasets of user credentials.

For this reason it is recommended that users always use unique passwords for each application, but this impose a complex need on the human mind, which has limited memory capabilities and a tendency to forget things. For an average user with several dozen applications used daily, the task of remembering unique passwords for each one becomes impractical and the need for a way to store these credentials are required.

Another issue with the widespread use of passwords is their inherent lack of complexity and entropy. With the raw computing power available today it is trivial to brute-force a guess at any conceivable password that a human can remember. For this reason may applications are beginning to require more and more complex structures for a password, with minimum lengths, requirement of special characters, case-sensitive letters, numbers etc. This also adds to the user's burden of having to remember these complex passwords.

[HV12] presents an analysis of the persistence of this flawed authentication model, and the failure of widespread adoption for better and more secure means of user authentication. Instead of adopting a better authentication mechanism, the industry has adopted the use of password managers - applications that help store the user's credentials for all their other applications.

1.1 Principle of a Password Manager

Password managers are common-place applications in today's digital world, and many people use them whether they are aware or it or not—as standalone programs, browser plug-ins, or even built-in browser features.

Common for all the different types of password managers are their purpose: to remember a set of user log-in credentials so that the user does not have to remember them. This is the bare-bone purpose of the application, but many products offer additional features such as synchronize the data between different devices, storing additional information in each entry, backup the data in the cloud, etc.

Having all your secret passwords in one centralized place is not without concern however. If an unauthorized person gets access to the password manager all of the user's credentials are suddenly available to this person. For this reason they are usually protected in some way that requires the user to authenticate themselves and prove that they should have access to the secret data. This usually takes place as a proof of knowledge through the user typing in a master password like the normal password authentication scheme.

Thus the security of a password manager is focused on its means of protecting the secret data in its vault (how it is stored and accessible outside of the application), and its means of protecting access to the secret data through the application.

That is the focus of this thesis.

1.2 Thesis Objectives

The goal of the thesis is to design and implement a prototype scheme for a password manager application with new and improved security measures. Through analysis and testing it will aim to achieve this by developing a new security scheme with the core functionality of having the sensitive vault data distributed across a number of different hosts (peers). The goal is to add strengthened security to the scheme by *not* maintaining the whole vault of the password manager upon one host, as this would leave it open to be compromised by means of brute-forcing, sniffing, or other methods of attack.

The scheme will aim to decentralize the data and functionality across different hosts removing a single point of failure/vulnerability and leveraging different distributed algorithms for peer-based decision making.

1.3 Methodology

The project will consist of three main phases: *analysis*, *design*, and *implementation* with tests of the implementation and evaluation of the design. Prior to the analysis phase a strict requirements model will be defined, encapsulating the security properties and features the project will focus on and deem necessary for a password manager to be considered complete.

The analysis phase will form the foundation of the project, by investigating existing solutions in the password manager domain, both practical and theoretical. The focus of the analysis and the works studied in this phase will be on the technical security and how the different solutions address the issue of securing access to the user's data (both internally in the application, and from an external agent/process). User-friendly features, market popularity, and other none-security factors will not weigh heavily in this analysis. By comparing the different solutions this phase will reflect on the list of requirements and the thesis objectives to show their support (or lack thereof) for data decentralization as a means to increase the security of the solution. This will form the motivation for the continued design and implementation work in this thesis.

The design phase will set out to develop a new scheme based on the knowledge acquired in the analysis phase and through additional technical research directed by the requirements. This phase will produce a new password manager design that takes on a new approach to securing the user's secret data through decentralization.

Finally, the implementation phase will develop a prototype of the design developed in the previous phase. This prototype and the design will be evaluated against the requirements derived in the analysis phase in order to conclude the project and reflect on the degree of success in achieving improved security.

1.4 Thesis Structure

This thesis is best consumed by reading it in its chronological order. This brings the viewer through the thesis' process from the start, with the domain introduction and analysis of existing solutions, through the design and implementation of the thesis's prototype solution, and finishes with the evaluation and conclusion of the thesis' process, system prototype, and this thesis report.

The individual parts of the report may also be read in seclusion or in any order that is suitable for the reader. The topic of each part is summarized below.

Chapter 1 contains the introduction, both to the project and to the thesis report. It briefly introduces the thesis' core domain and establishes the requirement model that is used in later chapters. It also describes the methodology for the project.

Chapter 2 contains the domain analysis, which presents some of the existing industry and academic solutions within the password manager domain. These solutions are evaluated against the requirements model. This part of the thesis forms the motivation for developing and implementing the system proposed in this thesis, as it documents the lack of solutions that satisfy the requirements model.

Chapter 3 propose the design of a new password manager that satisfy the requirements model. This includes a threat model, which generalizes the threats the design must protect against by modelling an attacker. The threat model is used to evaluate the design choices and make sure it satisfy the security requirements. The design includes architecture, concrete system modelling with Unified Modelling Language (UML) diagrams, security modeling, and distributed data flows.

Chapter 4 documents the project's prototype implementation of the design, including the implementation language and external libraries. The documentation is primarily handled through UML diagrams, as well as a detailed discussion on the security implementations through different cryptographic schemes.

Chapter 5 contains the evaluation of the design and implementation. This includes tests that document the prototype's fulfillment of the requirements model, but also reflection on the product of these phases and how they can be improved in further works.

Chapter 6 is the conclusion of this thesis and the project.

1.5 Decentralizing Data

This thesis will explore the possibilities of decentralizing the sensitive data stored in these password managers in order to increase their security and still protect the user's data in the event an attacker gains access to the device containing the password manager.

Data decentralization is a key component in distributed systems, where data is distributed between different components in the system rather than having them stored in a central location. Primarily, the decentralization can be broken down into two core types.

One type of decentralizing data is through duplication of the system's state, making it available in several components in the system. As a result of this the system often becomes more resistant to attacks on the system's availability, since the data is no longer focused on *one* component that can be broken down and denied availability (e.g. through a Denial of Service (DOS) attack).

Another type of data decentralization affects the secrecy of data rather than availability. In these cases the data is never available in its complete state on any given component in the system; instead a component much retrieve parts of the complete data from other components in the system. The result of such a decentralization, where unique part of the system state is distributed between different components, protects against compromised components such that they do not reveal the complete system state. Instead an attacker has to compromise a subset (or possibly the whole set) of the system's components in order to gather the complete system state.

This thesis is primarily focused on this latter type of data decentralization and is not immediately concerned with the availability feature of the first type.

1.6 Requirements

This thesis defines a strict set of requirements that constitute a secure password manager through its basic minimal feature model and strict security model. For a password manager to be considered complete in regards to the objectives of this thesis it must be able to satisfy all the requirements in this model.

These requirements form the basis of evaluation in the analysis of the existing password manager domain (section 2.3), as well as the evaluation of the project's design and prototype implementation (section 5.1.2).

The requirements are split up into two different categories: feature requirements, and security requirements. This separation allows for easier comparison with existing solutions on the market. Additionally, this separation is chosen over the usual functional and non-functional separation since this project will not focus on any non-functional aspects of the scheme. Instead it aims to introduce a general proof-of-concept that can be further designed and developed through future work. The goal of this thesis is not to design a password manager with revolutionary user features and as such the feature requirements will be kept to a bare minimum viable product (MVP).

1.6.1 Features

The basic purpose of a password manager is to store user login credentials in a secure digital vault; this must be the core feature of the scheme. For a better user experience and ease-of-use, the scheme must also support searching in the data that is stored within the vault.

Finally, for the vault to be of any use it must support creation, deletion, and modification of the vault entries.

The feature requirements are summarized below.

FR1 The vault must store entries consisting of at least a password, but may also include additional information (username, URL, etc).

FR2 The vault must support searching among entries.

FR3 The vault must support creation, deletion, and modification of entries.

1.6.2 Security

In order to add strengthened security to this scheme the fundamental requirement for the password manager is that it must not store its data on a single host; instead being separated into segments that are distributed across different host devices that operate within the scheme (e.g. a PC, smartphone, tablet etc.). These segments must be stored (persisted) on the devices and should be encrypted, as per usual industry convention.

If one such segment is decrypted or otherwise compromised, this segment should not be able to expose any meaningful secret data stored in the vault, whether directly or by being inferred from the compromised data.

When operating the password manager, this virtual vault comprised of all the distributed segments must be constructed ad-hoc on a host by communicating with the other hosts through secure channels, which much protect against eavesdropping, common replay, and Man-in-the-Middle (MITM) attacks. Such an action should be possible from any of the hosts in the scheme.

In order to start this communication and constructing the virtual vault the user must be authenticated, and all hosts in the scheme must approve the authentication.

The security requirements are summarized below.

SR1 The vault must only physically exist as unique segments distributed across a group of host devices.

SR2 Segments must be persisted on their devices.

SR3 Segments must be encrypted.

SR4 An individual (decrypted) segment must not compromise the data in the vault.

SR5 Communication between hosts must be done through a secure channel protecting against eavesdropping, replay, and MITM attacks.

SR6 Initiating a session must require user authentication.

SR7 User authentication must be verified by all hosts.

SR8 Any host must be able to initiate an authentication session.

Chapter 2

Domain Analysis

There exist many password managers on the market, and even more academic models have been proposed in numerous articles, conference papers, journals etc. This chapter sets out to analyse some of the most common, well-known, and most used password managers on the market. It will also briefly present and analyse academic models that propose password manager designs with increased security features. Due to the objectives of this thesis, this analysis focuses on the security aspects of the solutions and not the user-oriented features.

By analysing these solutions and models the goal is to present the security schemes that are found in today's password managers and highlight their shortcomings and vulnerabilities in respect to data location and the general security of the user's data, whether through the use of the application of external access to the data.

At the end of this chapter the analysed works will be mapped against the requirements from section 1.6 in order to motivate the further work done in this thesis, and thus present the need for a password manager built on a decentralized peer-based model that offers new security benefits.

For a detailed analysis of the most common attack vectors and threat models on password managers in general, [Li+15; Sil+14] have documented many useful findings.

2.1 Existing Solutions

The analysis of existing solutions on the market will focus on two main areas of password managers: solutions that are built into popular browsers and standalone applications. The latter is further separated into the categories *cloud solutions* and *local solutions*, as each has key defining features both for the user and for the password manager's security schemes.

2.1.1 Browser Solutions

Many people may use a password manager without even realising. The first place many people will encounter a password manager for the first time is in modern web browsers. When a user enters their login credentials on a website the browser will often show a small pop-up window asking if the login data should be saved. These are password managers.

Despite the existence of these built-in tools there is still a big market for standalone password managers and it stands to reason that one may question the usefulness, effectiveness, and most importantly the security of these browser built-in password managers.

This analysis focuses on the most popular web browsers and their implementation of password managers, the browsers being *Chrome*, *Safari*, and *Firefox*. As shown in fig. 2.1 these three brands dominate the market and cover over 82%. This data is aggregated from 10 billion page views on 2 million websites, per Statcounter's facsheet.

Chrome

Google's privacy whitepaper [1] states that the passwords are stored locally on the device, unless account sync is enabled and the user is logged in with their Google account in the browser; in such a scenario the passwords are synchronized in the cloud. Therefore Chrome's password manager can function as both a local and a cloud solution.

The whitepaper makes no mention of how the passwords are encrypted, and this information does not appear to be generally available. Instead, the research done by [ZY13] outlines Chrome's use of the underlying Operating System (OS)

[1] `https://www.google.com/chrome/privacy/whitepaper.html` Dec. 2019

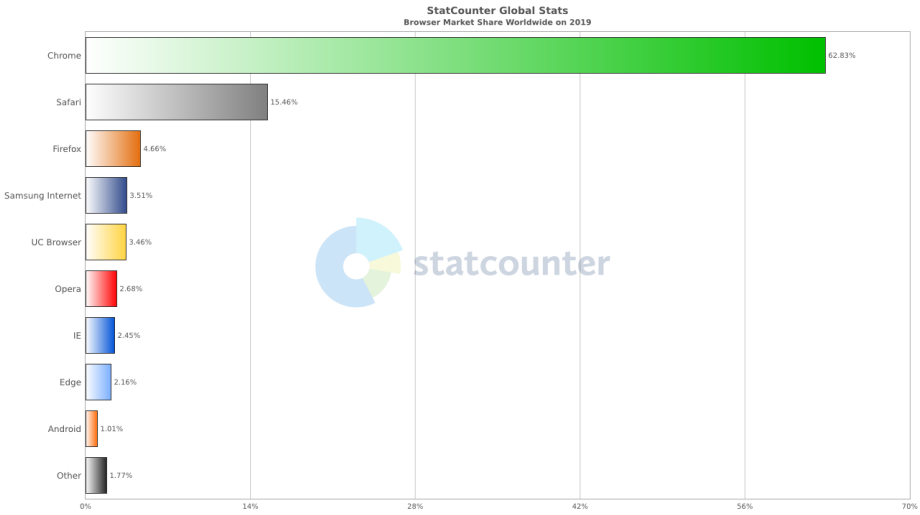

Figure 2.1: Global market share among browsers as of August 2019. Source: GlobalStats Statcounter

and its cryptographic Application Programming Interface (API) to encrypt the password entries in the manager's local files. This feature effectively mimics a master-password for the encryption scheme since only the system user that created the password entry can decrypt it, meaning the system user's password is now (by symmetric properties) the defacto master password for the manager's vault.

Though the research paper is a few years old by now, this encryption scheme appears to still be in effect, in so far as can be verified through testing and observing changes to the local storage files.

Though it may be convenient for the user that they need not remember a separate master password to access the data in the password manager [ZY13] outlines the security concerns with this approach. If an attack already has access to the system user (i.e. can authenticate as the user) the data in the vault is compromised. Such access may be gained externally from the system such as using the device in a public settings where other people can access the unlocked device, or it may be gained through other malicious code executing on the device.

Additionally, they did not find any indication that Chrome made use of features that add entropy to the encryption process (e.g. through salting).

Firefox

Firefox, like Chrome, also support both local and cloud operation, but its security features are drastically different, as found by [ZY13]. Instead of using the cryptographic API of the underlying OS Firefox bundles an implementation of the Triple DES (TDES) encryption algorithm. The shortcomings of this implementation is the storage of the generated secret keys which can be found on the device. Therefore, an attacker with access to the device's filesystem can simply locate the secure vault and the generated keys and subsequently decrypt the data without any problems.

To remedy this issue Firefox comes with an optional feature which allows the user to chose a master password that will be used to encrypt the TDES keys before they are saved unto the device. With this feature, any use of the keys will first trigger a prompt for the master password from the user, without which the data from the secure vault cannot be accessed.

With this approach the focus of any attacker now becomes the master password and how it may easily be brute-forced or acquired e.g. through phishing attacks such as Cross-Site Scripting (XSS).

Safari

Unlike the two other browser solutions Safari does not bundle its own password manager. Instead it integrates with a secure vault app built into the various Mac OS versions named Keychain[2].

This secure vault has an optional feature to synchronize with the cloud, enabling sharing of the secured data between devices, and like Chrome it relies on the authentication of the system user (their password) as security measure for decrypting the stored data.

Apple does not disclose the details of the encryption scheme underlying the Keychain app, but their developer documentation offers some insight into the architecture and the design choices[3]. Specifically, Keychain acts as an abstraction layer tasked with fully separating the vault from services that need the sensitive data. All cryptographic primitives required for handling the secure

[2] `https://support.apple.com/en-gb/guide/keychain-access/kyca1083/mac` Dec. 2019

[3] `https://developer.apple.com/library/archive/documentation/Security/Conceptual/Security_Overview/Architecture/Architecture.html#//apple_ref/doc/uid/TP30000976-CH202-TPXREF101` Dec. 2019

data is taken care of by the Keychain app, and services that require data from it does so with unique IDs referencing entries in the vault.

This level of separation does achieve some security through isolating the sensitive cryptographic primitives into the Keychain app, and limiting their exposure to the external services that require the data. However, as with the Chrome password manager, this scheme is still vulnerable to an attack that can authenticate as the system user.

As shown, it is a common theme to all three built-in password managers that they may be executing as a local or cloud solutions, but either case involves the secure vault being stored in its entirety on at least one single device. The different implementations introduces different vulnerabilities to how this singular vault may be compromised.

2.1.2 Standalone Applications

Password managers as standalone applications is a large market in of itself, with many different solutions and as many different features. The analysis will focus on some of the most popular solutions and investigate their security schemes. Many applications share the same schemes and instead compete on their different user-experience and none-security features, all of which will be ignored here as these are not relevant to the analysis.

2.1.2.1 Cloud Solutions

Dashlane is a popular and well-known password manager service, serving over 11 million users as of August 2019[4]. The platform has many services, one of them being their password manager. Though initially a local solution where the vault is stored on the user's device (both mobile and desktop support), Dashlane offers multi-device features, enabling the user to synchronize their data across several devices hosting their password manager application, turning it into a cloud solution. In this configuration Dashlane's servers are simply used as intermediary storage of the encrypted data, which the user's different devices can synchronize with.

Dashlane's security whitepaper [Das18] goes into details of their scheme's implementation within the password manager application itself. The encryption scheme relies on four primary secrets.

[4] `https://www.dashlane.com/about` Dec. 2019

- Master password selected by the user.
- Intermediate key encrypted by the master password.
- Device key generated for each device and used to authenticate it with the Dashlane servers.
- Dashlane secret key used to establish secure channels for communicating with plugins.

By default the master password is not stored on the device and it is *never* transmitted outside the boundary of the device (e.g. to the internet or any Dashlane server). Optionally the user may choose to have the application remember the master password, in which case it is stored on the device. The system components and locations of the cryptographic primitives are shown in fig. 2.2, where it can be seen that the components communicate over secure channels, both to the browser plugins and the cloud.

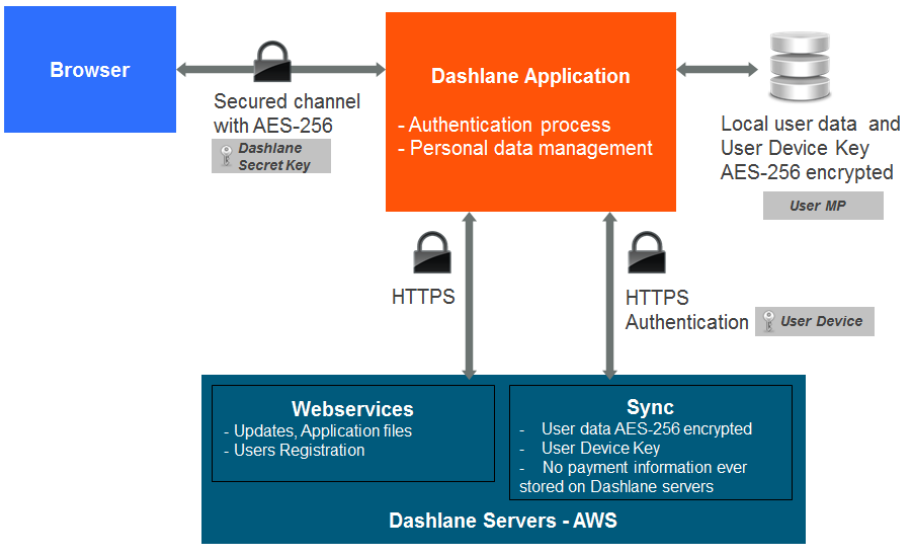

Figure 2.2: Dashlane components. Source: [Das18]

The purpose of the mater password is only to decrypt the intermediate key, which is the primitive used to handle all other encryption and decryption for the application and its secret data. This is a symmetric AES 256-bit key that is generated using the OpenSSL library. The algorithm and parameter details are further detailed in [Das18].

Additional security can optionally be added by enabling Two-Factor Authentication (2FA), in which case all local and cloud data is encrypted by a composite key combining the master password secret and a newly generated secondary key. This secondary key is *only* stored on the Dashlane servers, and handed to the application upon being authenticated using the generated 2FA one-time password.

This additional security ensures an attacker has to have knowledge of the master password and access to the 2FA ad-hoc generated one-time password in order to come into possession of the primitives required for decrypting the secret data. Using this scheme, phishing attacks become much less powerful since knowing the master password is no longer enough to decrypt the secret data.

The overall security scheme in Dashlane is very robust and the encryption primitives and implementations are up to date. However, the scheme lacks any security through decentralization. The only purpose of the cloud is enabling data synchronization between devices and remote authentication with 2FA enabled. If any device with the password manager is compromised the entire vault and its secret data is exposed.

LastPass is another big competitor on the market, marketing themselves as the #1 password manger with over 13 million users as of August 2019[5]. Unlike Dashlane, LastPass uses their cloud services as the focal point of the vault data, albeit still in a securely encrypted state. The vault is then synchronized to devices, which also maintain a local copy after synchronization [Las]. Thus, for this application the primary location of the vault data is in the cloud, and accessing the local device copy is only a backup in the event the cloud service cannot be connected to by the password manager.

The details of LastPass' security model is detailed in their whitepaper [Las] and an overview of the encryption scheme and secure communication channels is shown in fig. 2.3.

A master password is used to generate an authentication hash that is stored on the LastPass servers, as well as an AES 256-bit key used to encrypt and decrypt data; this key is *never* transmitted to the servers.

Like Dashlane, LastPass also supports 2FA in order to add additional security to the authentication of the user when accessing the password manager application. However, the whitepaper does not specify any additional cryptographic primitives that may be added to the encryption scheme by 2FA. An attacker

[5] `https://www.lastpass.com/password-vault` Dec. 2019

that can bypass the authentication and has knowledge of the master password may therefore still be able to compromise the secure vault, unlike in Dashlane where this feature added an additional primitive needed for encryption/decryption steps.

LastPass also suffers from same vulnerability as Dashlane, in that its cloud platform simply hold complete replications of the secret data vaults and does not utilize any security benefits from decentralizing the data.

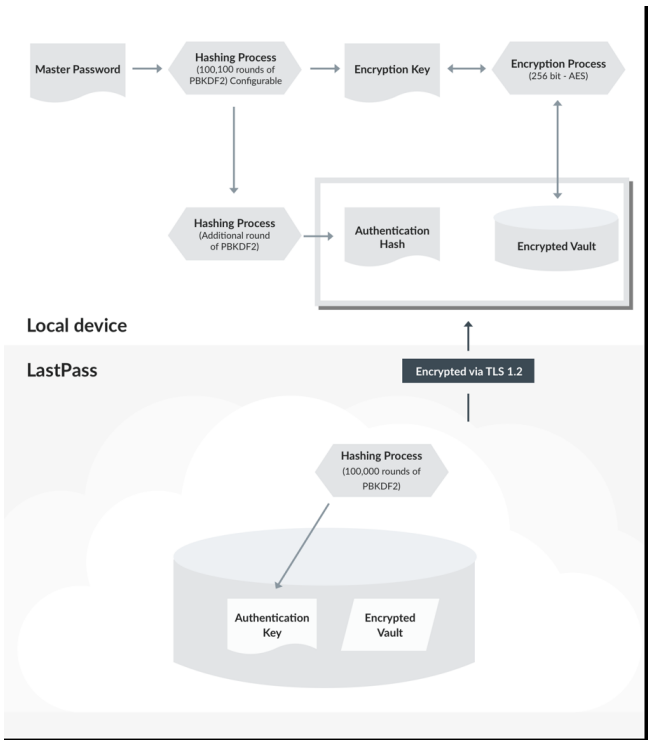

Figure 2.3: LastPass encryption scheme overview. Source: [Las]

Common themes for cloud-based solutions is the need to rely on the security of the cloud architecture. The replicated data that is handled in the cloud is commonly encrypted on the client side *prior* to reaching the cloud, and the encryption keys are never handled by the cloud. Yet it offers a point of vulnerability if the security schemes within the cloud architecture are suddenly vulnerable to attacks. Moreover, if the encryption scheme turns out to be vulnerable to brute-force attacks, the transmission of the secret (encrypted) data over a network introduces the risk of it being intercepted by an attacker. The decentralization has no (positive) security impact when the different nodes in

the system all have complete replicas of the data, and transmits the data in full when they communicate.

Despite this, the cloud services are a necessity to support most user-oriented features that make the products competitive, and rarely are they a means to strengthening the security.

2.1.2.2 Local Solutions

KeePass is an free password manager project maintained as open-source[6]. With no mandatory cloud integration, this solution maintains its vault in its entirety on the local device on which the password manager operates, using an encrypted database [Reib].

The security details for KeePass is maintained on their website [Reia] and can be summarized as follow:

- Whole database is encrypted, using either AES or ChaCha20 schemes (as of KeePass 2.x).
- A SHA-256 hash is generated from a composite of a master password, key file, and/or system user credentials (not password).
- The secure key is derived from the hash.
- Secret cryptographic primitives are kept encrypted in memory.

Unlike some solutions, KeePass has chosen to keep all the data in the local device database encrypted, meaning even username, URLs, etc cannot be viewed if the database is accessed. In order to decrypt all of this data the application uses a key derivation feature based on the composite of the user's master password, a file containing an additional (stronger) secret, and optionally some auto-generated system user keys that are associated with the system user account. Together, all three of these primitives require an attacker to have knowledge (password), possession (file), and identity (system user), in order to decrypt the database.

Keeping only the minimal amount of data in memory and keeping secret primitives encrypted until they are required is a somewhat common practice and helps protecting against an attack reading secret data from memory dumps,

[6] `https://keepass.info` Dec. 2019

whether these happen maliciously or through system paging (temporarily writing memory to the disk).

Because this solution is entirely local there are no decentralization (whole or partial) and no components to communicate with. The vulnerability of having all of the secret data stored on one device is somewhat mitigated by the multi-factor keys used to secure the data. Yet, in the case of an attacker who has gained access to the authenticated system user, it is only a matter of locating the key files and brute-forcing the password in order to compromise the vault.

Bitwarden is a local solution different from KeePass in that it allows for the same features and infrastructure as a cloud-based solution, but doing so with a local on-premise installation instead[7]. This means using a self-hosted server the user is fully in control of, rather than exposing the server to the cloud environment.

Bitwarden's security details are also only available on their site [Sol], and not in any published whitepaper. Like KeePass, it uses an AES 256-bit encryption scheme and derives the secret key from a hash of the user's master password; but only the master password—there are no mentions of a composite key.

The solution also supports 2FA, but as with LastPass this feature does not appear to have any effect on the cryptographic primitives, only the authentication flow which results in acquiring the secret key hash of the user's master password.

Despite functioning in a self-hosted local cloud solution, Bitwarden still shares the same vulnerabilities as the other cloud solutions studied here. The decentralization only occurs on replication of data, not actual fragmentation of it, and thus no advantages in security is achieved.

These local solutions make up for the network vulnerability of the cloud-based solutions, but are often much more feature limited because of it. However, the core issue with the security of these local platforms is still a single point of failure. Instead of residing in the cloud, the attack vector is now solely focused on the device that host the secured vault, and compromising this device means compromising the vault. The strong composite key schemes helps protect against brute-force attacks, but local solutions like those presented here all suffer from the vulnerability of all secret data being found on one device.

[7] `https://bitwarden.com/` Dec. 2019

2.2 Academic Research

In this section the analysis shifts its focus to theoretical models that have been proposed in academic literature but not adopted into functioning products on the market.

2.2.1 Methodology and Scope

In order to discover and acquire the academic models needed for this analysis a literature review has been conducted in the context of data security and password managers.

The tools used for the reviewing has been threefold:

1. Online academic search engines from DTU[8] and Google Scholar[9] used for accessing and acquiring literature.
2. Contextual search queries for filtering the literature to relevant password manager topics.
3. The snowballing model proposed by [Woh14] for evaluating the findings and discovering related literature through references and citations.

For a detailed account of the literature review including the search queries and the snowballing iterations, see appendix A.

The number of academic journals, articles, and conferences on data security models are numerous, and many models have been proposed which share some similarities with the goal of this thesis, but may be developed in other contexts than password managers. An example of such a model is DroidVault proposed by [Li+14], which propose a model for handling sensitive data files and code execution from a server in an untrusted Android stack. Another example is the model proposed by [YDC17], which use Visual Cryptography (VC) and Optical Character Recognition (OCR) to enhance password validation for authentication against a server while minimizing the vulnerability to offline attacks.

Though such work is significant and without a doubt interesting, the difference in context of such works – not being related to password managers – renders

[8] `https://findit.dtu.dk/` Dec. 2019
[9] `https://scholar.google.com/` Dec. 2019

them outside the scope of this analysis. In order to stay on topic in this analysis the scope for the literature review was restricted to *password-manager* contexts.

Nevertheless, components or methodologies of such out-of-scope models may still be referenced in later model development in this thesis.

2.2.2 Solutions

SplitPass

Perhaps the work most closely related to the goal of this thesis is the work done by [Liu+18]. This paper introduce *SplitPass*, a password manager that segments its passwords and distributes these segments between two distrusting parties: a password manager client (one a phone) and an assistant proxy server in the cloud.

SplitPass functions as a seamless intermediary application and protocol stack for user authentication against online services (e.g. social media websites) rather than self-contained isolated look-up application. This is on par with the browser plugins many existing standalone solutions come with, which enables the password manager to fill in a login-form without the user having to search the manager's vault for the explicit credentials.

The model makes heavy use of modifications on the transport layer of the TCP/IP stack, as shown in fig. 2.4. When a login request is made through HyperText Transfer Protocol (HTTP), the client component prepares the request by inserting its segment of the password followed by a placeholder into the request Universal Resource Language (URL). The model then prepares three separate Secure Socket Layer (SSL) packets; the first, which contains the request data prior to the placeholder (and thus includes the client's password segment), the second, which holds the placeholder and custom metadata, and the third which holds the rest of the request data after the placeholder. The first and third packets are sent to the target server, while the second packet is sent to the assistant server. The assistant server sends its segment of the password data to the target server on behalf of the client, and finally the model combines all three packets on the server to generate the complete request.

This collaboration between the two components, and the communication to the server, is done through a secure channel as specified by the SSL protocol.

The model obtains decentralization of the password data and ensures neither

component comes in possession of a full password. Thus the model provides additional security against theft and similar attack models; if one component is compromised and its encryptions are broken it does not result in the secure data being compromised.

The key differences between this work an the goals of this thesis is primarily rooted in the difference in application of the system. SplitPass aims to be a transparent library-like application and protocol stack that has a client and cloud-based server assistant that collaborate to construct valid authentication requests against web services. The goal of this thesis is to make an isolated application[10] with peers that collaborate to construct a fully ad-hoc vault in which sensitive information can be extracted.

There are also some shortcomings of the SplitPass model. Its prototype implementation only functions with URL based login requests (where the credentials are displayed in the request URL), whereas many modern authentication request puts the credentials in the request body.

From a security perspective it is worth noting the plain determinism of the passwords always being segmented in two equal halves, one preceding the other. Introducing randomness into the segmentation process of the passwords would provide more security against an attacker's ability to brute-force the missing segment if the attacker has come in possession of one of the components and compromised their secure vault. Additionally, the model makes no mention of encryption schemes outside of the transport layers (SSL), so no assumptions can be made on whether the password fragments are stored in an encrypted state.

Kamouflage

Kamouflage is a password manager model proposed by [Boj+10]. In this special model the focus is put on securing the vault against misuse in the event that the vault is compromised, and doing so by using decoy passwords. In this approach, for every 1 real password in the vault there will be up to 10,000 decoy passwords for the same log-in credentials (username/email), obscuring the real password to an attacker that has managed to decrypt the secret vault.

This form of security forces an attacker to use online attacks—testing the passwords against the services for which they are stored, and verifying whether the login proceeds or fails. This is a tedious step for an attacker; furthermore, many websites nowadays implements a lock-out feature which either blocks or

[10] Not integrating with plugins or other third-party processes

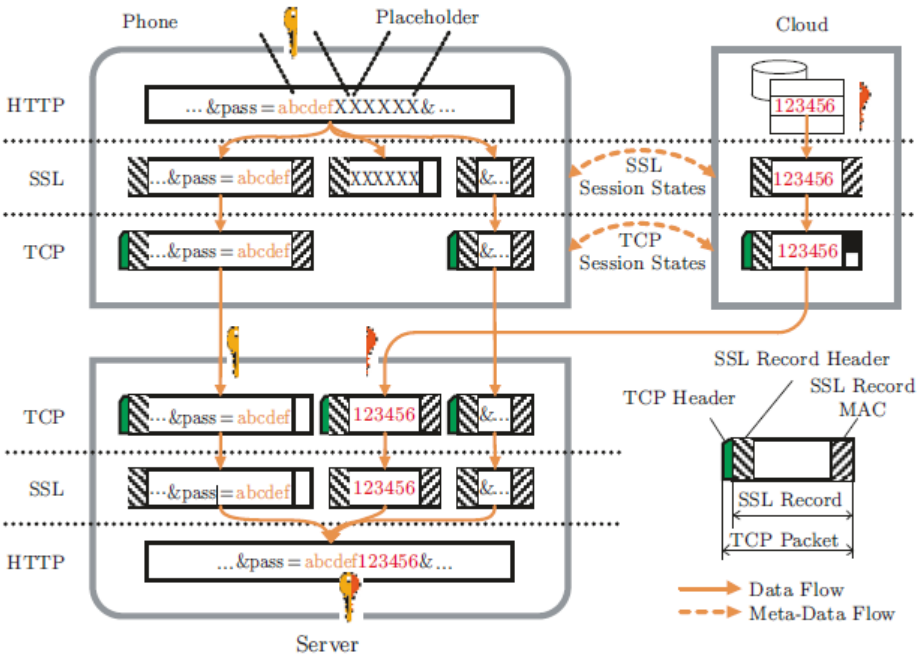

Figure 2.4: SplitPass overview of interactions between components on TCP/IP protocol level. Source: [Liu+18]

times-out an account after a number of failed attempts to login (typically 3 or 5 attempts in a row).

For the decoy passwords to be effective they must closely resemble the real password in structure in order to make them look like human-choices and not computer-generated text strings. This model achieves this through an extension of the context-free grammar proposed by [Wei+09]—a technique usually reserved for cracking password databases, not strengthening it.

The vault itself is stored in its entirety upon the device hosting the password manager and an abstract example of the vault's database storage is shown in fig. 2.5. In order to access the vault the user is authenticated using a master password which is cryptographically transformed into an index that fetches the real password set rather than a decoy set.

In the initial model the vault and its data is stored in the clear, relying on the obfuscation of the decoys to hide the secret data with no encryption. In an extended version the model proposed in [Boj+10], encryption is added by

deriving a secret key from the user's master password, which is then used for encrypting the real passwords and thus making the model a Password-Based Encryption (PBE) scheme. This secret key only works for decrypting the real passwords; for each of the decoy password sets a decoy master password is generated for use in encrypting this set.

As mentioned in the paper, this model is particularly susceptible to an attacker that may have sociological knowledge of the person owning the secure vault. Since passwords are often made using words, phrases, and names the user likes, an attacker with knowledge of the user's sociological domain will have an easier time differentiating real passwords from decoys.

Furthermore, it is not all online services that implement a lock-out feature. In the event the vault contains an entry for such a service an attacker could utilize this vulnerability to exhaust all passwords sets until the real entry is found and thereby exposing the cryptographic secrets that are needed to comprise the real password set in its entirety.

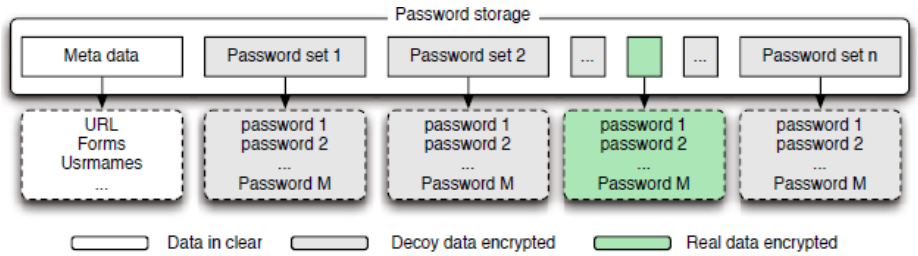

Figure 2.5: Kamouflage storage overview, for M real password entries and n - *1* decoy password sets. Source: [Boj+10]

DFF-PM

Another academic model for a secure password manager that is of interest is the Decentralized File Format Password Manager (DFF-PM) proposed by [Agh+16]. Like the previous Kamouflage scheme, this model focuses on adding security to a password manager through obscurity and its threat model is primarily concerned with protection against an attacker using sensitive data that has already been compromised. The model achieves this through two concepts: faux passwords and a decentralized file format.

This DFF-PM model also use the principle of decoy passwords, though by the name of faux passwords. These are used to pollute the model's password table

and obscures the legitimate user password entries that are stored by the password manager. By ensuring the faux password representations are close enough to the legitimate entries, an attacker who has compromised the password table will not be able to guess the legitimate passwords from the faux ones. Equally, attempts to use the faux passwords for authentication may lead the attacker to believe the decryption of the passwords were not correct.

This model achieve a close proximity to the legitimate passwords using a Modified Levenshtein Distance algorithm for generating the faux passwords. This algorithm ensures a maximum number of operations needed to change a faux password into its legitimate counterpart, and thus ensuring they do not differ so much as to an attacker easily being able to identify the legitimate password.

Additional protection against an attacker that compromise the secure vault is added by the model through the use of a decentralized file format, which stores sensitive data in separate files on the the device, rather than opting for the common one-file database structure which is common in many password managers.

By storing passwords in a file separate from usernames, compromising one file does not give the attacker access to a complete credential set needed for use in logins. Additionally, by decoupling the two files an attacker would not gain knowledge of the association between different passwords and different usernames even in the event that both files became compromised.

The implementation of the model uses AES to encrypt the different decentralized files, and the use of the password manager required user authentication with a master password.

This model is of interest primarily for its decentralization theme, although the faux password component is a noteworthy addition to the security model. The decentralization in this work takes on step towards the goal of this thesis by performing local decentralization of the sensitive data, but it lacks the full scope of decentralization beyond the boundaries of the device hosting the password manager. Because of this the secure vault with all of its files is still vulnerable to an attacker that gains access to – and compromise – the device.

As with the Kamouflage model, this security of the faux passwords relies on the assumption that attempts to use compromised credentials for login on services will block the attempts after some number of failed attempts. If this is not the case the attacker can freely utilize a service to discover the exhaust the full set of passwords to discover the legitimate one.

Additionally, the extra data provided by these faux passwords may accidentally

aid in attempts to crack the password encryption in the first place if the generated faux passwords ends up being common entries that are often used for offline brute-force attacks such as rainbow-table/dictionary attacks. This is of more concern in this model where the same cryptographic primitives are used to encrypt the entire password file, as opposed to the Kamouflage model that used different primitives for each password set.

NoCrack

The NoCrack model proposed by [Cha+15] takes the principle of the decoy and faux passwords one step further and consists of two primary schemes: first it applies a transformation model on the password set in the vault, followed by a standard PBE model.

The transformation model uses a Distribution-Transforming Encoder (DTE) to encode the password set, a principle which is based on an extended model of the honey encryption proposed by [JR14]. The transformation model used in [Cha+15] consist of two models which are documented in the paper:

- A normal DTE model for encoding computer-generated passwords.
- A special Natural Language Encoder (NLE) model for encoding user-generated passwords.

The purpose of the DTE models is to transform the password set into a modified data set such that when it has been encrypted and is subsequently decrypted, only the correct cryptographic primitives will derive the original password set through decryption and subsequent decoding. Using any other incorrect master password for the decryption will result in a decoding that presents a plausible decoy password set.

This stands in contrast to the Kamouflage and DFF-PM solutions. In case of the former, specific master passwords had to be used to decrypt the decoy password sets and thus it is possible to supply a none-mapped master password which would fail the decryption and could therefore be ruled out as a possible valid master password. This exclusion of possibilities shrinks the set of plausible master passwords eligible for use to decrypt the vault (albeit they have to be discovered through trial and error) and makes it easier to compromise the vault through brute force. In the case of the latter, the same cryptographic primitives are used to secure the entire password set, therefore the set of plausible master passwords is reduced to only 1 and makes it even easier to single out.

For the NoCrack model, any wrong master password used for decryption will lead to a decoy set that – to the attacker – will look like a plausible solution; in order to verify its legitimacy the attacker would have to test the passwords for authentication online against their registered services. This means the attacker cannot rule out any combination of a master password before he has verified that the decrypted and decoded passwords fail in online authentication.

With this model the security strength of the decoy technique no longer lies in the number of decoys and how well they obscure the legitimate passwords, but rather in the strength/entropy of the master password—the higher the entropy the harder it will become to determine the correct input to generate the legitimate password set from the DTE.

The transformation is not enough in itself, however, and requires a standard PBE model to be applied to the encoded password set. This is done using AES CTR encryption with a secret key derived from the user's master password using SHA-256 PBKDF2 hashing with a salt. The encrypted vault is stored on the mobile device using the password manager, while a cloud sever is used to store a backup copy which is also used for synchronization with multiple devices, with communication between these two components being done using HTTPS.

One vulnerability with this model – highlighted in the paper – is that of updates in the vault. If an attacker has access to two version of the vault, one before and one after an update, then only the correct master password will result in two decrypted vaults that are similar (except for where changes happened), and thus it becomes easy to rule out wrong master passwords which produce too many variations in the two vaults. In this case its security falls back to that of a simple PBE scheme and relies purely on the strength of the encryption scheme.

With the cloud server storing full copies of the encrypted vault, this architecture leaves multiple nodes that are susceptible to attacks with the purpose of exploiting the above-mentioned vulnerability. Compromising any of these components and acquiring multiple versions of the vault undermines the enhanced security features of this model—a problem that could be addressed by decentralizing the data in the vault, without excluding the features of the DTE model.

BluePass

The model BluePass proposed by [LWS19] takes a different approach to securing the data vault and does so through some measure of decentralization—not of the data but of the cryptographic primitives. This model consists of three main components:

- The user's pone serves as a transparent and autonomous[11] storage device of the encrypted secure vault.
- A computer is the user's entry-point to using the system.
- A BluePass server is the storage of the cryptographic secrets used to process the secure vault.

When the user interacts with the password manager the computer communicates with the server and the mobile device, as shown in fig. 2.6. The user's master password is used to authenticate the user against the server, which in turn returns a private key k_2. This communication is done over HTTPS.

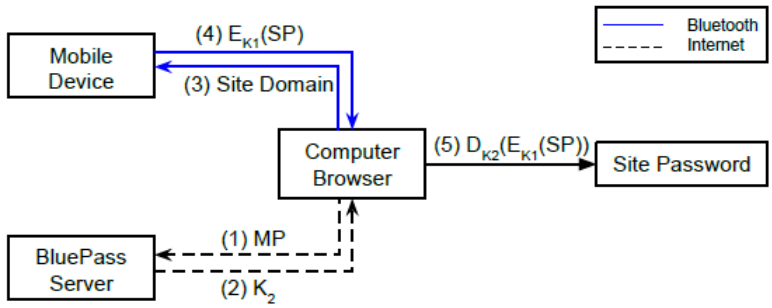

Figure 2.6: BluePass data flow. Source: [LWS19]

Afterwards, the computer fetches the encrypted password for a requested site domain from the mobile device; this password is encrypted using the public key K_1—the counterpart to the private asymmetric key K_2. This is done over a secure Bluetooth channel and also servers as a form of 2FA due to the limited range of Bluetooth requiring the user has possession of the phone and that the phone is in proximity to the user's computer (with the other of the two factora being the user's knowledge of the master password).

Finally the computer use the server's private key K_2 to decrypt the password and auto-fills it into the log-in form that required the password.

The model dictates two version of the computer: it can either be trusted or untrusted. A trusted computer may choose to save the private key K_2 for some extended period, but an untrusted computer will only store the key for the duration of the browser's session, meaning it must authenticate against the server with the user's master password for each new browser session.

[11] The user does not interact with the phone

The encryption used in the mode is a 2048-bit RSA scheme, and the asymmetric key-pair is generated on a per-phone basis. Thus a vault that is copied and stored on two different phones will not have the same cipher text as two different key-pairs will have been used between the devices. The phones themselves are identified and mapped by the server using the MAC address of the phone's Bluetooth device.

All of these aspects of the model makes it somewhat resistant to theft and data breach due to its partial decentralization. If the phone is compromised the attacker only has access to the encrypted vault not the cryptographic primitives needed to expose the data; doing so would require also compromising the server or acquire the user's master passwords. Furthermore, because the vault is not encrypted using the master password the attacker can not launch an offline brute-force attack against the vault in an attempt to acquire the master password; in order to get it the attacker would have to attack the server.

If an attacker compromise the trusted computer he will gain easy access to the private key K_2, which is of some concern. But even in this scenario the attacker would not have access to the vault, which is decentralized on the phone. The attacker would have to compromise the secure Bluetooth channel in order to eavesdrop on the communication and acquire the secured password, or compromise the device to gain access to the whole vault.

Tapas

Tapas is a distributed password manager model proposed by [McC+12]. The key security feature of their work is what they call *dual possession authentication*, i.e. authentication is validated by possessing two devices (unlike 2FA where you need two separate factors, e.g. knowledge and possession). By splitting the functionality of the classic password manager use case between two devices this scheme offers additional security against theft, be it of devices or data.

The model consists of two main components operating on different devices: a *manager* and a *wallet*. The manager encrypts and decrypts sensitive data and the wallet stores encrypted data. The two components communicate with a secure channel.

With decentralization of the two core cryptographic entities (the key and the sensitive data) an attacker must be in possession of both devices in order to gain access to the sensitive data. Having access to only one of the devices leaves the attacker in possession of either the keys or the encrypted data, but not both.

In the prototype developed in [McC+12] the implementation uses a Firefox browser plugin as the manager and an Android app as the wallet, but the architecture is not limited to this platform choice.

The data itself, which includes site information and login credentials (username and password), is encrypted using a 128-bit AES-GCM encryption scheme with the cryptographic primitives only stored on the manager. When the data is required the wallet transmits the encrypted data to the manager, which then decrypts and serve the data. The GCM mode of the encryption scheme includes integrity checks to ensure the encrypted cipher texts cannot be maliciously modified which may occur in certain types of attack models such as padding oracle attacks.

Because the model makes no use of a user master password for authentication or even key initialization the scheme is free to use more strongly randomized keys with better protection. Additionally, the user does not have the burden of remembering a strong master password that – if broken – will compromise the sensitive data.

This model does, however, still suffer the vulnerability of having the secured data stored in its entirety on *one* device. Even if the keys for decrypting the data is not readily accessible, in the event an attacker has possession of the wallet the encrypted data is susceptible to offline brute-force attacks, given an attacker with enough resources. The model also lacks a concept of user authentication. Although the use cases require a user to interact (to initiate and confirm actions) with both the wallet and manager device, these interactions can very well be instigated by a malicious agent.

Similar solutions that focus on the same principle with decentralization of secret keys and encrypted data has also been proposed in other works such as [Fuk+16].

SPHINX

The work done by [Shi+17] introduces a password manager model that focuses on high-entropy passwords to offer more security against offline dictionary attacks, phishing, and data breaches on the services that actually use the passwords. This model is called SPHINX and is based on the Device-Enhanced Password Authenticated Key Exchange (DE-PAKE) model proposed by [Jar+16].

It achieves its goal by dealing with two different concepts of passwords: *pwd* is a user-generated password, and *rwd* is a high-entropy computer-generated password. The user only deals with *pwd* and use this to authenticate with the

password manager as he/she would in any other model, but instead of using *pwd* for the online services SPHINX maps pwd into a much stronger *rwd*, which is used in the online services. This is demonstrated in fig. 2.7.

The model contains two primary components: the client which is the computer the user interacts with and authenticates against, and a mobile device that stores the cryptographic primitives set k, which is used to transform $pwd \rightarrow rwd$. An interesting property of this transformation is that only the primitive k and the user-supplied *pwd* is required for the process, meaning neither the device nor the client has to store either of *pwd* and *rwd*–only k is stored in the system; this makes the model very robust against device and data theft, minimizing the avenues an attacker can utilize.

As shown in fig. 2.7, the transformation is done using the user's password and the domain for the requested password, with a unique k being stored for each such domain. Because this pair is a unique identifier, the user *can* choose to use the same value *pwd* for all entries in the vault, effectively making it a master password. But the optimal solution is to have the user use a separate password for each of the entries. This also means the user is implicitly authenticated by both the mobile device and client since an un-mapped *pwd* would not result in any data being exposed.

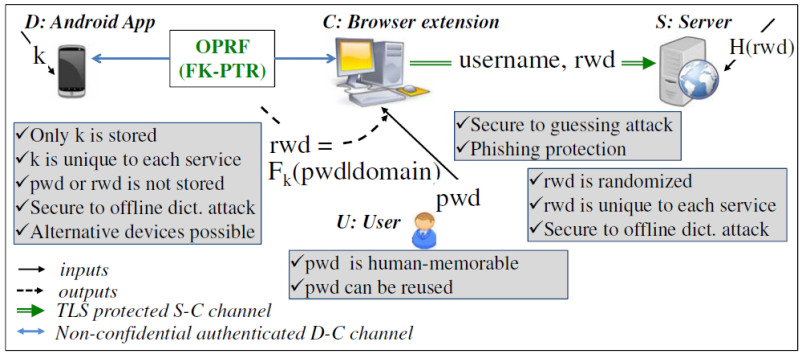

Figure 2.7: SPHINX authentication overview involving (U) user, (C) client, (D) device, (S) server. Source: [Shi+17]

The transformation itself, which happens during the communication between the client and the server, is done using Oblivious Pseudo Random Function (OPRF), a modification of the protocol initially coined PTR in [Jar+16]. This protocol enables secure exchange of information without having to expose the secrets used to derive the information; thus the client can derive the mapping $rwd = F_k(pwd|domain)$ without having to learn the secret k from the phone device. The implementation of the protocol uses elliptic-curve hashing and

SHA-256 to add additional entropy to *rwd* before it is handed over to the online service over HTTPS.

While this model has some strong security features, it does also have a couple downsides. Because the model transform the user's password before it is being used by online services it required the model to be part of the user process from the beginning, since it must be used to generate the high-entropy password that will be registered on the service. This means the presence of the password manager is not transparent, as many models aim to be, and may deter a user from utilizing such a system.

Additionally, as mentioned previously, the user *can* elect to use the same *pwd* for all entries in the vault, effectively making it a master password. In such a scenario the model lose much of its added security strengths as it now relies on the strength of the master password—if this is compromised there is no additional security preventing an attacker from learning the transformed *rwd* values. It should be noted, however, that in this case the attacker would still have to extract this information on a per-request basis, as there is no storage to compromise and the model never transmit more than the requested password, for a given domain. Thus the attacker would have to cycle through probable domains to discover the passwords.

Amnesia

The final academic model covered in this analysis is that of Amnesia, proposed by [WLS16]. Like several other models it adds security through the use of a separate devices that decentralize the cryptographic primitives needed to use the secret vault and its data. This offers a sense of 2FA where the attacker has to compromise more than one factor of the process in order to expose the secure vault.

As shown in fig. 2.8 the model consists of four primary components.

- A computer the user interacts with and that navigates to a service that requires login.
- An Amnesia server that stores cryptographic secrets set K_s.
- The user's phone, storing cryptographic secrets set K_p.
- A rendezvous server used to facilitate communication between server and phone.

When a user navigates to a site and requires login (1) the model contacts the Amnesia server over HTTPS to acquire the login (2) and authenticate with the user's master password (not shown). Using the rendezvous server, the Amnesia server sends a request R to the phone, containing the username μ, the domain d, and a 256-bit random seed σ (3). The phone returns a password token T generated by transforming R using the K_p (4). Finally the server computes the password P using T and K_s and returns it to the computer(5). Both T and R are hashed using SHA-256, while P is hashed using SHA-512.

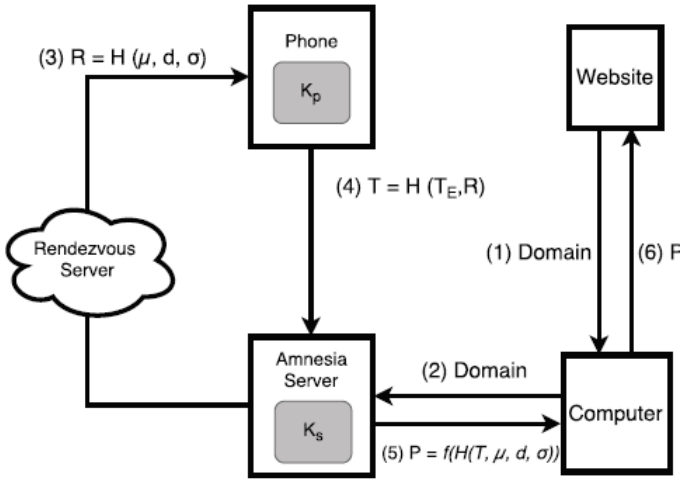

Figure 2.8: Amnesia data flow. Source: [WLS16]

What is special in this model is its cryptographic secrets K_p and K_s used to transform the password request into a password. The secret on the phone contains a table of 512-bit random values which is used for generating the token T; it does so by splitting the request token R (which is a fixed-length hash unique for a given $< \mu, d, \sigma >$ set) into fixed segments which are used to look-up into the table, each such look-up resulting in a value that is used in conjunction to generate the token value.

The secret K_s contain a table of printable characters that can be mapped into a password (alphanumerical characters) and it also split the hashed token T into segments, which are then used to look-up the printable characters in the table and thus generating the requested password. This whole process thus ends up with the transformation $< \mu, d, \sigma > \rightarrow P$ using two different hashing stages and a translation of the hashes using two separate secrets.

As with the SPHINX model this solution does not include explicit storage of the secure vault's data in the conventional way like many other password man-

agers. Instead it utilize its algorithms and secret tables to transform a unique password request into the desired password. This means the attacker cannot compromise the vault by breaking an encryption scheme on one entity of data. Compromising either of the phone or server only reveals the tables used to carry out the transformation and does not explicitly compromise the whole vault; instead the attacker would have to eavesdrop on subsequent requests to discover one password at a time.

Therefore one can describe this model as simply "hiding" the vault in plain sight, since the secrets are not themselves encrypted. All the data of the passwords are encompassed in the table in K_s but one needs to know which entries to fetch in order to compose a given password—information which is only disclosed from the phone with its password token T. Since the domain and username information can easily be inferred in the request R, the seed σ serves to add entropy to the hash.

2.3 Requirement Mapping

As presented in this chapter, the popular password manager solutions that exist on the market, whether cloud-based, local installations, or built-in browser engines, do not leverage the security found in decentralizing the sensitive data stored in the vault of password managers. Instead they focus on user-friendly features and entrust proven encryption schemes and policies to remedy the single point of failure (vulnerability).

Different academic solutions have been studied that leverage these added securities of decentralization, each in their own way and to varying degrees of success. Yet, the implementation and use case for these proposals differ from the goals of this thesis.

Table 2.1 gives an overview of how the different solutions satisfy the requirements outline in section 1.6. As seen, there is no *one* solution that checks all of the requirements.

Based on the analysis of these different solution proposals, whether existing or academic, this thesis is motivated with the goal of developing a model and prototype that satisfies all of these requirements, as shall be presented in the following chapters.

	Browser Solutions	Dashlane	LastPass	KeePass	Bitwarden	SplitPass	Kamouflage	DFF-PM	NoCrack	BluePass	Tapas	SPHINX	Amnesia
Features													
FR1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
FR2	✓	✓	✓	✓	✓	x	✓	?	x	x	x	x	?
FR3	✓	✓	✓	✓	✓	?	✓	?	✓	✓	?	✓	✓
Security													
SR1	x	x	x	x	x	✓	x	x	x	x	x	x	x
SR2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	?	?
SR3	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	x	?
SR4	x	x	x	x	x	✓	x	?	x	x	x	✓	✓
SR5	-	✓	✓	-	?	✓	-	-	✓	✓	✓	?	✓
SR6	?	✓	✓	✓	✓	x	✓	✓	✓	✓	x	✓	✓
SR7	?	✓	✓	✓	✓	x	✓	✓	x	x	x	✓	x
SR8	x	x	x	✓	x	x	✓	✓	x	x	x	x	x

Table 2.1: Existing solutions' fulfillment of the requirements; (✓) full, (?) partial/unknown, (x) none, (-) not applicable.

Chapter 3

Design

The analysis of the existing solutions in the password manager domain showed in section 2.3 demonstrate the lack of a password manger that satisfy the strict requirement model for this thesis—one that is build on the architectural foundation of data decentralization in a Peer-to-Peer (P2P) network.

This chapter propose a new design for a Distributed Password Manager (DPM), meant to fulfill the requirements outlined in section 1.6. In order to validate the design choices made in this chapter—and analyse possible alternatives—a threat model is formulated based on the requirements.

Using the threat model to inform the design decision, the complete design is presented. It is documented in terms of its architecture, the primary data flows that link the architectural components, and the main security schemes needed to counteract the threat model's attacker. Finally, all these specifications and choices are combined to form a coherent system model documented using UML diagrams.

3.1 Threat model

When designing a solution it is important to ensure it complies with the security requirements outlined in section 1.6. In order to do so a threat model is created to to capture the threats the system must protect against and the vulnerabilities it must mitigate. Threats that are not captured in such a model are considered out-of-scope and are not required to be addressed by the system (though it may still be). The modelling process was inspired by the application threat modelling process proposed by Open Web Application Security Project (OWASP)[1].

Using the model, every design decision is made with an evaluation of how the effects of such a design choice would affect the threat model, and to what end it might unintentionally weaken the security goal of the system.

The requirements already outline an initial abstract overview of the system, and this overview is visualized in the threat model's diagram shown in fig. 3.1. This diagram capture the abstract architecture and their data flows.

It is important to note that the threat model is symmetric for all the nodes. Security requirement SR8 dictates that any host node must be able to initiate a session with user authentication, and this means any node may initiate authentication data flows and must also respond equally to such data flows. It should also be noted that the particular number of nodes shown in fig. 3.1 does not hold any significance as the requirements do not mention a specific number of nodes, simply a node network.

The following sections outline the different aspects of the threat model and go into details with the vulnerabilities the different aspects must protect against, and those that are considered out of scope.

3.1.1 Components

The core components of the model are the secure data vault fragments and the nodes in the P2P network that facilitate the data communication and processing. As depicted in fig. 3.1 the model considers each node with an associated vault fragment, for which the node has the responsibility of handling storage and processing. Such a node-fragment pair is located on a single physical device.

One or more nodes in the system may be subject to malicious intent, but the model does not allow for the whole network to become malicious. For the

[1] `https://www.owasp.org/index.php/Application_Threat_Modeling` Dec. 2019

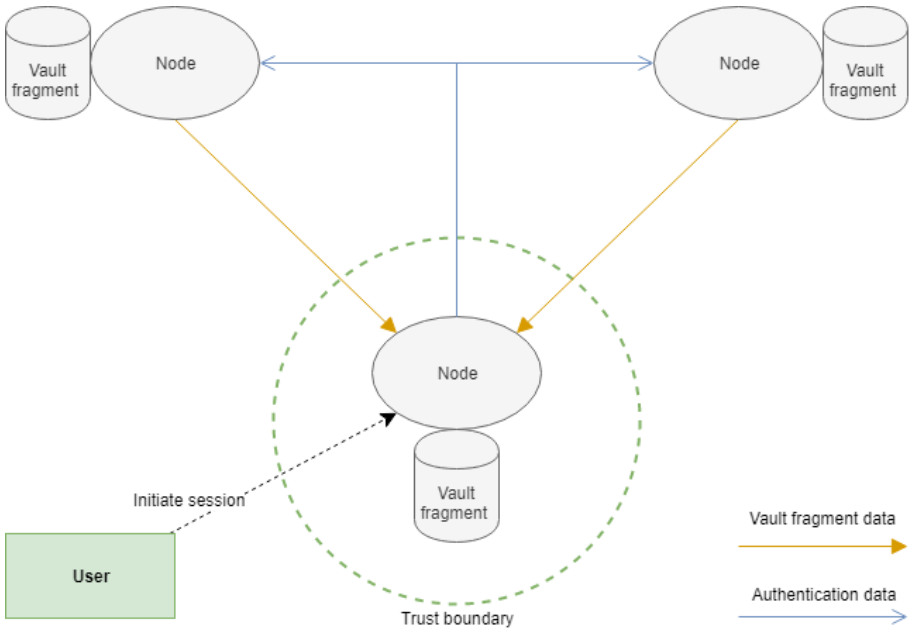

Figure 3.1: Threat model - architecture and dataflow

purpose of simplicity, it is acceptable that the model ends in a locked state in the event that malicious nodes occur. This means the vault will not be usable by the user, at which point it is assumed the user will take reactionary actions to cull the malicious node and restore the network to a working state.

Furthermore, a vault fragment may be compromised, but doing so (as per the requirements) must not expose any meaningful secret data from the vault. While the secrecy of the data may be compromised, it should be infeasible to maliciously altering the data.

3.1.2 Trust Boundary

The trust boundary for each node only encompass the individual node and its vault fragment. Due to the distributed nature of the system there cannot be any inferred initial trust between nodes in the system; such a design could enable one malicious node to compromise the entire system. This implies a node must prove its trust to another node before its vault fragment can be used. The authentication data flows make up this proof of identity and establishing of temporary trust between nodes.

Similarly, the user is also outside the trust boundary of a node and thus he/she must be authenticated to prove their trust in the system. This goes for both the node that initiates a session and all other nodes in the system during the session.

3.1.3 Data Flow

Because of the small trust boundary, it is vital that the communication between the nodes is secured with proper communication channels that do not compromise the data in transit or interfere with the correct execution of the features in the distributed system. Establishing such secure channels also require trust between the nodes, and thus the authentication flow is of vital importance in this model. Without it, the nodes will not be able to collaborate in merging their fragments and create an ad-hoc usable data vault.

When the secure channels have been established the trust boundary has effectively (but temporarily) been expanded to the whole node network, enabling them to collaborate and transmit their vault fragments to the initiating node. This property imposes a restriction on the model: a malicious node cannot complete authentication and gain the trust of the node network. If this was the case a malicious node could break the temporary trust, which much not be feasible.

It is important that the complete vault is neither persisted nor cached in any usable state on the node when it is outside of the temporary trust boundary; doing so could compromise the whole vault in the future if the node becomes malicious.

The model also impose another restriction on the dataflow, in that the authentication and vault fragment-exchange must be an atomic operation in the system. A node cannot become malicious after being authenticated but prior to exchanging its fragment with the initiating node. Equally, an initiating node cannot become malicious after authenticating with the node network but prior to receiving the network's vault fragments. A node can only become malicious outside of a session, meaning when it is outside of the temporary trust boundary.

3.1.4 Evaluation

In order to evaluate the threat model and verify that it captures the security requirements an attacker model is created with the properties that are mitigated by the security requirements. This enables verification of the threat model by

showing that the attacker cannot achieve their goal of gaining access to the secure data in the vault.

The model considers an attacker with the following properties:

- Can gain access to a node's vault storage.
- Can compromise the secured vault storage.
- Can emulate a node's behavior (malicious node).
- Can intercept data communicated in the node network.

Node Storage Access

If an attacker has gained access to a node's storage, he will only have gained access to a fragment of the complete vault. With no caching or storing of the complete vault on any *one* node the attacker cannot gain access to any copies of the complete vault.

Compromising Secure Storage

If the attacker has compromised the storage and the mandated encryption of the vault fragment, the fragmented nature ensures the attacker will not come in possession of any meaningful data from the secure vault. The strength of this assertion depends on the entropy of the fragmentation function.

As an example, consider a low-entropy implementation that include a network of two nodes where the password set P is fragmented such that for each password $\rho \in P$, with length l, the fragment function $f(\rho)$ produces two fragments f_ρ^0 and f_ρ^1 such that

$$f_\rho^0 : \prod_{i=1}^{\lfloor l/2 \rfloor} \rho[i] \wedge f_\rho^1 : \prod_{i=\lceil l/2 \rceil}^{l} \rho[i] \tag{3.1}$$

Here $\prod$ is a string concatenation operator, $\lfloor x \rfloor$ is the floor operator on x, $\lceil y \rceil$ is the ceiling operator on y, and $\rho[i]$ is the i^{th} printable character in ρ. In plain terms, each password is split in half and a fragment simply contains one half.

In such an implementation an attacker can with relative ease derive the full password entries when only compromising one of the vault fragments. This process would only require brute-forcing (guessing) one half of each password entry, and such operations may become even more trivial with sociological knowledge of the vault's owner.

With more than two network nodes the strength would naturally grow with the number of nodes in the network; if there were three, an attacker would have to brute-force $\frac{2}{3}$ of the password instead of just $\frac{1}{2}$. Even so, the entropy of the fragmentation is still very low due to the sequential nature of the data stored in each fragment, which by itself still offers a useful amount of meaningful data to an attacker.

Therefore the entropy of the fragmentation function should be uniformly distributed across the password and network-node domain. Furthermore, it would be more ideal to have the fragmentation function process raw bytes of data from the data stream that make up the secure vault, rather than simply processing specific character data sets stored within the vault.

Malicious node

Should the attacker emulate—or otherwise influence—a node's behavior and thus making it a malicious node, the authentication flows will prevent the attacker from abusing the system.

If the malicious node tries to initiate a new session and retrieve the other vault fragments from the rest of the node network, the other nodes will reject the invalid authentication attempt.

If the malicious node tries to respond to a legitimate authentication request the originating node will reject the malformed response and the attacker will not be able to submit a maliciously-crafted vault fragment.

Data Interception

Should the attacker be able to intercept the data communicated within the node network, the secure channel ensures the attacker cannot successfully alter any transmitted data, nor can the attacker attain any meaningful data from the channel as it is protected against eavesdropping through confidentiality.

Thus it shows that an attacker is successfully stopped by the proposed threat model and it can be used as the informed basis for developing a design for the new password manager scheme.

3.2 Architecture

The first part of the DPM design is its architecture, which is used to define the external components, their relative structure and interrelations, and their means of communication within the system. [Dra18] defines an architectural model as describing the abstract structure of a system using four basic building blocks:

- *Communicating entities* - the entities in the system that communicate with each other.
- *Communication paradigms* - the method with which the entities communicate.
- *Roles and responsibilities* - the roles and responsibilities the entities hold.
- *Placement* - mapping the entities unto physical placements in the system's infrastructure.

Since DPM will be a distributed system the focus of the communicating entities and their placements will be in the form of a discussion on the system's network topology. The design's communication channels discuss the communication paradigms; and finally the entity responsibilities are discussed as part of the system's internal node structure, but is also an inherent part of the network topology discussion.

3.2.1 Network Topology

The project requirements (section 1.6) specify a node network, which is analogous to a P2P network. This is one of the two most well-known and used network topologies, with the other being the client-server topology. For completeness, both are covered in this discussion.

Client-Server

In the classic client-server topology, a network consists of entities that have distinct roles: a *client* and a *server*. From a system user's perspective, a server is generally some remote entity that is not directly involved with the user's interactions; instead the client handles human-to-machine communication, and the machine-to-machine communication is done between the client and server entities.

Another defining feature of this topology are the interactions that define the entity roles: a client is an entity that creates requests, and a server is an entity that responds to requests. An entity may be one or both of these roles, depending on the system and its functionality.

It is a common implementation of this topology to have a much greater number of client entities than server entities, as the servers are generally centralized components in the network and have well-defined, static network locations (addresses), allowing clients a well-defined point of entry into the network. Meanwhile the clients are often ad-hoc entities that do not have static network locations and may join and leave the network at any moment.

The strength of this topology is its simple network structure. Having well-defied, centralized locations makes it easy and very efficient for nodes to communicate with the network. But the centralization of entities are also its weakness. It requires at least a partially rigid network structure, when it comes to the server entities. These entities often require much greater security than client entities as their centralized nature provides a common entry-point into the network, and compromising such an entity can often compromise the entire network. Furthermore, their centralized nature presents a bottle-neck in communication flow.

This topology is not an ideal solution for the DPM design. The analysis of existing solutions (section 2.1) highlighted several issues with designs that incorporated centralized entities, chief among which was their general tendency to weaken the security of the system in favour of user features. Furthermore, their centralized nature makes them far more vulnerable to attack on the network's availability due to their single point of failure.

P2P

In contrast to the client-server topology a P2P network does not enforce the same roles upon its network entities. In a true P2P network all entities are semantically identical in their roles and their functionality within the network's procedures; any action that can occur on *one* entity, can occur on all entities, and thus they are interchangeable.

Within the communication flow of the network, at any instance of time there may be nodes in the network that take on the role of a client, and other entities that take on the role as servers, in that an entity makes a request to the network and the other entities respond to the request. However, an important distinction between such an instance and the client-server topology is that these roles are not statically defined. In one instance one entity may be a client and another be a server, yet in the next time instance the roles may be reversed. In a classic client-server topology, a client entity will never take on the entity of a server (though the server may take the role of a client when communicating with other servers).

If the network procedures only rely on a subset of the network entities, meaning it can satisfy its functionalities without requiring all entities in the network to be online and available, the network facilitates reliability and availability. If the network duplicates its state across different entities it facilitates data redundancy. These are common strengths of this topology over the client-server type, and together they ensure a basic form of fault tolerance in the network model.

The weakness of this type of topology is its implementations of these features. Without well-defined network locations the communication flow often becomes much more laboured as all the nodes have to handle all the network maintenance: mapping and searching/lookup resources in the network (be it data or specific nodes), maintain and update the network interconnections when nodes enter or leave the network, etc. Overlay networks are a means to mitigate these implementation weaknesses by defining partially rigid network connections, roles, and procedures as an abstract imaginary network that is laid on top of the pure P2P network.

Figure 3.2 shows a visualization of the common overlay network topologies, where leaf-nodes are shaded. These leaf nodes denote the proper P2P entities that carry out the functionality of the network, while the un-shaded tree-nodes are entities that are responsible for maintaining the overlay network, i.e. routing the network communication and handle network maintenance (entities joining or leaving the network, resource routing, etc).

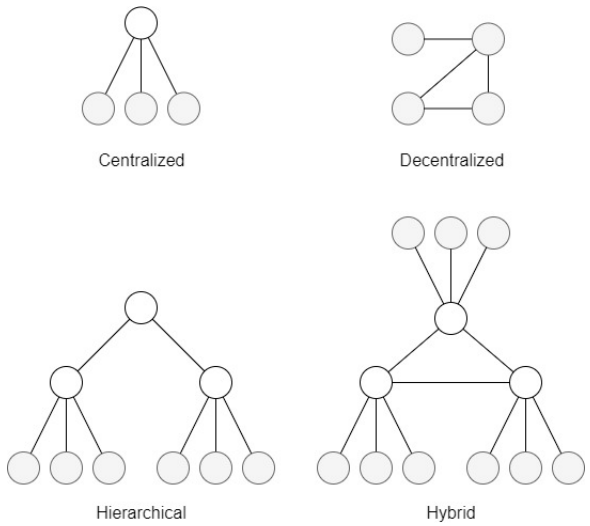

Figure 3.2: Common overlay network topologies.

The P2P topology is the clear choice for the DPM design, as indicated by the requirements of a node network. Particularly, the fully decentralized network appears to be the best choice due to the very small expected size of a network. Furthermore, the prototype is only expected to work within the context of Local Area Network (LAN), meaning it does not have to deal with complex network maintenance and routing of a large number of entities.

The node count is not expected to ever exceed more than a handful of nodes for any given node network operating a single logical password manager unit. Such a small logical network far supersedes the need for an overlay network, which would only introduce unnecessary network-managing nodes, and introduce a degree of partial centralization that would require special types of entities or physical nodes, the later of which is not very attractive for a LAN-bound system.

In the event the solution was to be expanded to function in a Wide Area Network (WAN), an overlay network would be required in order to administer the network structure more dynamically and efficiently. In such a situation a hybrid network would be more suitable. However, for this proof-of-concept design and its prototype, the decentralized node network is sufficient.

Evaluating the choice of topology against the threat model, the decentralized P2P model enables the peer-based use of nodes in the system and allows each node to function as its own entity with a limited trust boundary that can be temporarily expanded to encompass the entire node network upon authentica-

tion of the other network nodes. It also satisfy the requirement for all nodes to be involved in the authentication procedures; this would be an issue for a topology containing nodes whose only purpose was network management and would not be directly involved with the user's interactions.

3.2.2 Communication Channels

The communication channels, or more simply the means of communication—is another important design choice in the solution's architecture. In a distributed system these channels handle the direct and indirect communication that synchronize state and implement remote computing across the system's components, whether through local inter-process communication or remote communication.

There are many different paradigms for distributed communication and a useful way to compare and evaluate these differences is through two implementation requirements: *time* and *space* coupling. A channel that is time coupled requires both channel participants (the sender and receiver) to be online and available at the same time; conversely, a channel that is *not* time coupled does not require both the sender and receiver to be available at the same time. A channel that is space coupled requires the channel participants to be aware of each other's existence (known how to locate/address each other), while a channel that is *not* space coupled does not require the sender to be aware of the receiver's existence and vice versa.

With these four implementation characteristics, a communication paradigm can fall into one of four categories, which [Dra18] describe as shown in table 3.1.

	Time coupled	Not time coupled
Space coupled	Communication is directed towards receivers, that must exist at the time	Communication is directed towards receivers, but need not exist at the time
Not Space coupled	Communication is not directed towards receivers, but they must exist at the time	Communication is not directed towards receivers, and need not exist at the time

Table 3.1: Space and time coupling matrix.

For the purpose of this project, the communication channel that is most useful is

one which is not space coupled but is time coupled. Space coupling is not desirable since it implies a rigid overlay network that adds unnecessary maintenance complexity for this proof-of-concept, though it may be a more viable solution in production. Time coupling is a necessity imposed by the threat model, since all entities in the network must engaged in the authentication flows in order to establish the temporary expanded trust boundary. Such authentication cannot be guaranteed if all entities do not agree on the same life-time.

Table 3.2 maps the space-time coupling properties to some of the common communication paradigms used in most distributed systems to date. The criteria outline above show that the multicast and broadcast socket message passing paradigms are most suitable for the DPM design. These two, and the other paradigms, are discussed in the next few sections.

Paradigm	Space coupled	Time coupled
Sockets (message passing)		
Unicast	✓	✓
Multicast	x	✓
Broadcast	x	✓
Remote invocation	✓	✓
Message Queue	x	x
Tuple Space	x	x

Table 3.2: Common communication channel paradigms and their space-time coupling.

Socket

Sockets are one one of the most basic means of inter-process communication and an implementation of the abstract message passing paradigm. It defines three communication methods: unicast, where a sender transmits a directed message to a known receiver; multicast, where a sender transmits an undirected message to all unknown receivers within a recipient group; and broadcast, where a sender transmits an undirected message to all receivers in the network.

Sockets are best known for their implementation in the TCP/IP internet protocol stack, where they are an inherent part of the Internet Protocol (IP) and transportation layers, which serve to locate the host and application within a network.

This transportation layer has two primary paradigms for how to handle com-

munication delivery between sockets: Transmission Control Protocol (TCP) is a connection- and stream-oriented protocol that ensures reliable delivery, and User Datagram Packet (UDP) is a connection-less packet-oriented protocol that does not ensure reliable delivery of data.

Because TCP uses connections to send streams of data between two bound sockets it falls under the unicast paradigm, where the channel is bound by both space and time. UDP does not maintain connections and can be used for any of the three unicast, multicast, and broadcast communication paradigms.

While unicast is the ideal way of ensuring reliable communication between two well-defined nodes in the network it is not suitable for handling the dynamic discovery of the unknown (space un-coupled) receivers in the decentralized node network. Multicast is perfectly suitable for this task as it send undirected messages to a group of unknown receivers, as opposed to broadcasting which sends to every entity available on the network, even if its not part of the DPM node network.

Thus the system's communication channel design can be satisfied by using multicast to handle the discovery of the dynamic node network, and unicast to handle any subsequent communication between individual nodes after their discovery.

UDP is ideal for handling the dynamic discovery and querying in a small network because of its simple support for multicast. However, this method does not define a reliable communication channel, whereas TCP is more useful for its reliable unicast support. These paradigms also satisfy the threat model specifications and both implementations allow for the application to enforce the security model's data confidentiality and integrity on the raw data that is transmitted in the messages.

Remote Invocation

Remote invocation is an abstract, high-level communication paradigm where entities expose parts of their API for other remote entities to invoke directly, behaving as if the programming model was operating on local components in the entity rather than on remote ones. This is particularly useful in object-oriented systems where the remote invocation paradigm allows the system to treat a remote entity like any other local object and use its API to carry out tasks.

The direct invocation of a component's API enforce a very tight coupling between the interfaces of the interacting entities, and the implementation of this

paradigm often requires a lot of overhead when implementing.

It is a useful paradigm for large-scale systems that has a very comprehensive distributed communication need, but the limited operations required to satisfy this project does not warrant the configuration and implementation overhead imposed by this paradigm. Additionally, it is only suitable for direct communication between well-defined nodes in the network due to its space coupling, making it unfit for the dynamic network discovery. It is also more difficult to impose a strict security model to ensure confidentiality and integrity.

Message Queue

Message queues are an abstract, high-level communication paradigm where entities communicate indirectly with a persisted buffer/queue rather than with other entities directly. This allows the paradigm to be decoupled from both space and time.

The paradigm has no coupling between interfaces of the senders and receivers, and the non-coupling of space makes it very suitable for the dynamic network handling in the password manager's node network. The issue, however, is its implementation requirements. The persisted queue(s) must be a localized entity in regards to the network – a component every node always has access to and can locate. Such additional network maintenance nodes were already discussed in section 3.2.1 and are not justified for the small node networks that are handled in this system.

Another issue is the decoupling from time, which complicates the authentication flows and makes it difficult to satisfy the need for all network entities to be involved in the authentication of a user and node.

Tuple Space

Tuple space is an abstract, high-level communication paradigm not entirely unlike message queues. Like message queues the paradigm requires persisted storage to save the messages sent into the tuple space and decouple it from time. It is decoupled from space by messages not being addressed to any receivers; instead of functioning as a sender-receiver, an entity publishes a message (with a message type) to the tuple space and another entity polls the tuple space for messages of this type in order to receive it.

The time decoupling makes it unsuitable for the DPM design for the same reason as message queues are unsuitable.

3.2.3 Node Structure

The full system consists of a network of symmetric nodes/entities where each node must be identical in its internal components and functionality, as dictated by the system requirements (SR8). Each node much be able to function in network of size 1 (i.e. the node is the only node in the network) and a network of any size greater than 1. This means each node must be able to handle the full responsibility and role of the system features individually, and simultaneously support collaborate with a node network to distribute the responsibility, taking on the role as a client requesting to other nodes (servers) in the network.

To achieve this each node must—as a minimum—consist of internal components that handle the following responsibilities:

- *I/O* - for managing I/O operations to the local device the node is hosted on, i.e. reading from and writing to the device's storage, including encryption and decryption operations.
- *Network* - for managing communication with the other nodes in the network, both directly (unicast) and indirectly (multicast), including establishing secure connections.
- *UI* - for handling user interactions with the node, both user input and visual output.

The combination of these architectural design choices are shown in fig. 3.3, excluding the internal node structure. The fully decentralized P2P network topology is manifested through a (UDP) multicast group that is used to discover the network and then initiate (TCP) unicast communication channels between the network entities. All maintained within LAN.

It is worth emphasizing, as discussed in section 3.2.1, that the network structure is not rigid and thus the node entities do not know the other entities in the network. They neither store nor cache this information. Therefore the unicast channels must *always* be preceded by the multicast channel, in order to discover the network in its current state whenever a node entity needs to communicate with the rest of the network. This introduces minimal network maintenance responsibilities and ensure a fluent network structure where new nodes can freely join.

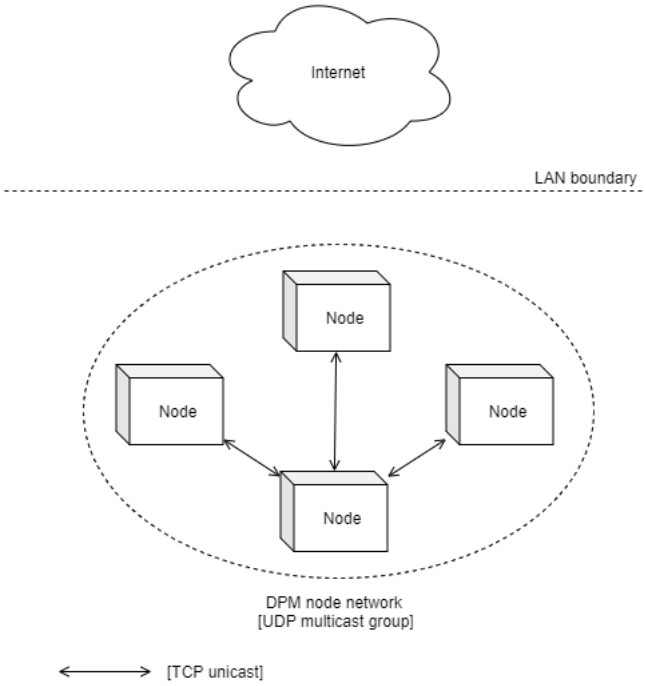

Figure 3.3: DPM architecture - combined overview of network topology, communication channels, and communicating entities.

3.3 Data Flow

The threat model from section 3.1 highlights a basis for the minimum data flows required in the DPM design. This section further specifies their minimum design considerations and requirements for both internal and distributed data flows.

3.3.1 Configuration

The configuration data flow has two important goals: (1) to establish the node network for the password manager (or join it if it exists), and (2) to ensure user authentication is possible both locally and remotely in collaboration with the node network.

An issue arise for all distributed data flows regarding filtering of messages in the event two users have a DPM instance each within the same LAN context. The

messages must not be valid across different users' instances of the application.

The system architecture specify the choice of group multicasting for handling the dynamic network discovery and initial messaging, after which unicast channels are established between entities in a DPM instance's network (section 3.2.2). This means the instance message filtering must occur as part of the initial multicast messaging to ensure only nodes within the same instance network establish unicast channels, and messages do not cross the boundary of another DPM instance in the same LAN context.

This filtering can be achieved in two different ways: either by using a unique multicast group for each DPM instance, or use the same multicast group for all instances and use a unique network identifier for each instance which can then be used to filter messages based on their target instance identifier.

Using a different multicast group for each instance has several drawbacks. There are a finite number of multicast groups in the TCP/IP stack, which could theoretically be depleted within a given LAN boundary. Additionally, the configuration flow would have to include additional messaging procedures for determining which groups are already in use in the LAN context.

Using a unique identifier for each DPM instance is the better solution, where the only real drawback is that a node must check all messages it receives, discarding those that do not have a matching instance network ID to that of the local instance.

The identifier can be uniquely generated by creating a secure one-way hash of the user's master password key-derivative and a timestamp seed. The master password key-derivative will be the same on each node in the DPM instance's network, but is not unique since different users may use the same master password in separate instances. Therefore a unique seed is needed to differentiate them. A timestamp with millisecond accuracy is sufficient, and can be generated automatically when the network is initially created, and specified by the user on new nodes that are joining the network. The probability of two users choosing the same master password and their DPM applications generating their network ID at the exact same millisecond is sufficiently improbable for this design.

Because the network ID will be stored alongside other network properties on the device (the node ID and the seed used to generate the network ID), and due to the strict attacker in the threat model, it is not safe to only use a secure hash on the user's master password directly. Instead it should be used on a secure key-derivation from the user's master password, to maximize the entropy of the hash value used in the network ID generation.

The network identifier generation is thus shown in eq. (3.2), where $KDF(k)$ is a key-derivation function on key k, mp is the user's master password, $Hex(i)$ is the hexadecimal value of integer i, t_{ms} is a timestamp with millisecond accuracy (as an integer), and | is the concatenation operator. The hexadecimal function is introduced in order to condense the string representation of the timestamp for when the user has to input it in the configuration phase of new nodes that must join an existing instance's network.

$$Network_{ID} : Hash(KDF(mp)|Hex(t_{ms})) \tag{3.2}$$

3.3.2 Authentication

A DPM node must support two authentication flows: locale and remote. Because the network ID contains a secure key-derivation of the user's master password the local authentication can be handled without having to directly store the key-derivation on the device. The node must simply re-compute the network ID using the user's input for master password and the stored network seed, and then compare the new ID with the stored network ID.

Remote authentication requires remote/distributed data flows to other nodes in the network, and as per the architecture design this must be done through socket communication channels. The local derivative of the user's master password cannot simply be transmitted between nodes for comparison since the communication channels are not initially secured and can only become so *after* the nodes have authenticated each other. Doing so would allow an eavesdropper to intercept the derivative and enable them to pass the simple authentication check.

Another reason the key-derivative cannot simply be transmitted to the other note is the threat model and the attacker's ability to maliciously control a node. If the remote node was compromised and it received the key-derivation as part of an authentication check it could simply pass the check, store the key-derivative for future us and establish a secure channel, thus breaking the temporary trust boundary in the system. Until the secure channel is established, the system is network of nodes sharing a mutual lack of trust, and thus no secret information can be transmitted as part of the authentication flow.

To achieve secure distributed authentication the system must use a Password-Authenticated Key Exchange (PAKE) scheme that achieve mutual authenticate without exchanging their secret data. Once this is done, the secret key generated

from the authentication scheme can be used to open a secure channel between the nodes. This is further specified in section 3.4.1.

3.3.3 Vault Fragments

The fragments of the secure vault are mainly part of the distributed data flows between nodes and participate in two separate data flows: requesting fragments from the network, and sending updated fragments to the network. These are of course only initiated after a secure communication channel has been established as the result of a successful remote authentication flow.

A node must request fragments from the network in order to construct its temporary secure vault once the user has been authenticated (both locally and remotely). A node must also send fragments to the network once a change has been made in its vault (e.g. through addition or deletion of an entry) in order to reflect the the change in the network's distributed state.

This means a node must support the following vault fragment data flows:

- Request network fragments.
- Respond with local fragment.
- Send updated fragments to network.
- Accept a new updated fragment.

When responding to a network request with the node's local fragment, and accepting a new updated fragment, the node takes on a server role, waiting for the request from the other node after the initial secure channel has been established. In the former flow the node must respond to the request with its node fragment, while the latter flow does not require a response–it simply has to accept the new fragment and save it.

When requesting network fragments and sending updated fragments, the node takes on the client role and initiates the data flow once a secure channel has been established.

The generation of the vault fragments, i.e. the process of fragmenting the secure vault data, must emphasize high levels of entry as discussed in section 3.1. In order to achieve this the fragmentation should follow the process described from the pseudo code in listing 3.1.

```
let C be the number of desired fragments
serialize vault into byte array B_ARR
initialize C empty target byte arrays
initialize C empty mask integer arrays
for each byte B in B_ARR
    let I be the index of B in B_ARR
    let N be a randomly selected integer where 0 ≤ N < C
    add B to the N'th target array
    add I to the N'th mask array
end
```

Listing 3.1: Vault fragmentation psuedo code.

Using a cryptographically strong random number generator to generate N the entropy of a fragment scales exponentially with the number of nodes in the network, and thus the number of fragments. Each byte will have a $\frac{1}{C}$ chance of being assigned to any given fragment. This means the probability of subsequent bytes of a string being assigned to the same fragment can be expressed as eq. (3.3) for a string of l continuous bytes when generating C number of fragments.

$$p(l) : \frac{1}{C^l} \tag{3.3}$$

As an example, assuming standard UTF-8 string encoding where a password with 4 characters requires 4 bytes of data, fragmenting this data for two nodes would result in a $\frac{1}{2^4} \approx 0.063$ probability of the whole password being stored of the same fragment. For the same password fragmented for three nodes it would be a $\frac{1}{3^4} \approx 0.012$ probability. For a normal password that requires at least 8 characters, fragmented between just two nodes, the probability is down to just $\frac{1}{2^8} \approx 0.004$ while it is an insignificant $\frac{1}{3^8} \approx 0.0002$ for a network with three nodes.

This is a significant improvement of entropy of the vault fragment data and demonstrates the effective security of this model simply when going from two to three devices. It goes without saying that for a network with just *one* node it is equivalent to using a normal password manager since the whole vault will be contained in the one fragment. But as soon as other nodes are added to the device the attacker has to brute-force a growing proportion of the vault data even after they have compromised a device, discouraging the attack in the first place.

The mask arrays are needed in the fragments to be able to map the bytes back into a complete array again in the correct order when constructing a vault from

its fragments. This does have the unfortunate effect of allowing an attacker who compromise a fragment know where the data belongs in the complete vault's data array. But the entropy of the fragmentation process still means the attacker has to brute-force the missing data.

3.4 Security

With the data flows and the architectural components defined for the DPM design, the focus shifts on the general security schemes that will be required in order to implement the strict security model set out for the system. The focus is first laid on how the system can implement the secure communication channels. Then a brief discussion is presented on the design for securing the vault fragment storage; and lastly the design choices for the hashing and key-derivation schemes are discussed.

3.4.1 Secure Communication Channels

In order for a channel to be secure it must provide authenticity, confidentiality and integrity guarantees. The DPM design will ensure these guarantees by using encryption schemes to secure the channel—more specifically the data transmitted on the channel.

Asymmetric encryption is a common way of securing communication channels in a distributed system since it does not impose a the parties using the channel to use the same secret key; instead a a pair of public/private keys are used, one which is readily available and usually used for encrypting secret data (the public key), and one that is secret knowledge with the recipient and used to decrypt the data (the private key).

However, there are some drawbacks with using asymmetric encryption: (1) it is generally much slower and more resource demanding than its counterpart; and (2) it is a one-way communication. The same set of asymmetric keys cannot be used to guarantee secure communication both ways on the channel (i.e. both sender-to-receiver and receiver-to-sender). This makes the asymmetric schemes unfit for establishing a secure unicast channel, as is required in this design.

The symmetric encryption schemes, unlike the asymmetric counterparts, are much more suitable for establishing the security guarantees on a unicast channel, and doing so more efficiently. The downside to symmetric schemes in a

distributed system is the need to establish a shared secret key to be used for encrypting and decrypting the channel data. Clearly this secret primitive cannot simply be generated on one node and then transmitted on an insecure channel to the other node – this would break the confidentiality of the key and any agent that intercepts the key in transit would be able to eavesdrop on the channel after it has been secured with encryption. Some schemes address this issue with asymmetric encryption to exchange secret key information, but such schemes generally do not guaranteed authenticity of the user/node sending the data. Additionally, such a scheme would introduce the need for generating key-pairs at the start of the unicast connection and exchange public keys in order to securely transmit the required secret data for both nodes to derive a new shared secret key used to secure the channel.

Much more elegant means of handling this issue has already been developed and is common practice in the industry. In particular, the use of PAKE schemes to exchange secret key information without the need to transmit any secret data between nodes, but still being able to independently derive a shared secret key.

Such schemes ensures authenticity since the data is exchanged based on mutual knowledge of a password—in this system the user's master password—which enables nodes to authenticate each other based on their ability to establish a secure connection. Once the key has been established, confidentiality is guaranteed through the encryption of the channel data using the agreed secret key. The integrity guarantee is added by using an Authenticated Encryption (AE) scheme when encrypting the channel data, which guarantees integrity (data authenticity).

3.4.1.1 SAE - A PAKE Scheme

The PAKE scheme chosen for the DPM design is called Simultaneous Authentication of Equals (SAE) and is presented in [Har08]. It describes a PAKE scheme ideal for P2P networks as it does not impose any strict lockstep protocols and can even happen simultaneously between different nodes. The scheme uses a finite cyclic group in which the discrete logarithm problem is infeasible, in order to allow two or more nodes to independently derive a shared secret key based on their mutual knowledge of a secret password and some non-secret parameters generated by—and exchanged between—the participating nodes.

By using this scheme as an initial handshake protocol when an channel is established between two nodes an insecure channel can be made secure using the generated shared key to encrypt any subsequent channel data. Additionally–as a PAKE scheme – it also serves as an authentication method, ensuring both

nodes have the same secret knowledge of the user's master password in order to successfully execute the handshake.

The SAE scheme offers the following security properties, which are further detailed with informal proofs in [Har08]:

- *Not trivially secure* - the scheme works and generates a shared secret key.
- *Passive attacks are infeasible* - an attacker passively listening on a channel between two normal nodes cannot derive or obtain the legitimate secret key.
- *Active attacks are infeasible* - an attacker that is actively engaging with a normal node cannot forge messages that would lead to the attacker deriving or obtaining a legitimate secret key.
- *Dictionary attacks are not possible* - an attacker cannot use a compiled dictionary of secrets and protocol parameters to derive a legitimate secret key.
- *Forward security* - obtaining the secret password does not provide an attacker with the means of obtaining the secret key of any previous sessions.
- *Resistance to Denning-Sacco attacks* - obtaining the secret key of a session does not provide an attacker with the means of obtaining the secret key of any other sessions.

These security properties are vital to show that the scheme satisfies the threat model and its requirements to a secure channel by protecting against *eavesdropping*, *replay messages*, and *man-in-the-middle* attacks. As shown in the following sections, using the SAE scheme to generate a shared secret key that is then used to secure the channel data using an AE scheme, the nodes successfully establish a secure channel for communication.

Eavesdropping

The SAE scheme's protection against passive attacks ensure the security protocol is safe from eavesdropping, i.e. an eavesdropper will not be able to break the secure channel by eavesdropping and gathering data during the SAE handshake protocol. Once the channel is secure, the encryption ensures confidentiality of the channel data and prevents eavesdropping on any subsequent data transmitted on the channel.

Replay Messages

There are two kinds of replay attacks: one where packets are replayed within the session of a connection, and one where packets from different sessions are replayed in the current session.

The TCP transportation protocol that was chosen for the unicast messaging in the architecture design ensures reliable delivery of messages[2], which ensures no replay messages within a TCP session, and by extension within the session of a channel.

The SAE scheme protects against replay packets across sessions due to its forward security and Denning-Sacco attack resistance. These security properties ensure the session-local secret primitives involved in the handshake prevents replaying of messages encrypted using an old session secret key, since this key is different from the one generated in the current session and using it would result in invalid decryption of the replay message, which would then simply be discarded.

Man-In-The-Middle

Active MITM attacks can occur in two ways: (1) an attacker can intercept the message transmitted on the channel and alter it before delivering it to the receiver; or (2) an attacker can forge entirely new messages that do not originate from an honest sender and deliver them to the receiver.

By using an AE encryption scheme with data integrity guarantee it ensures the system will detect any altering in the encrypted data transmitted on the secure channel and mark it as invalid. Under such a scheme, successfully altering existing messages would require the attacker to have knowledge of the secret key used to secure the channel—knowledge the attacker does not possess.

Fabricating entirely new messages is also not possible. Under the SAE scheme the security-guarantee against active attacks ensure the attacker cannot gain knowledge of the secret key used to secure the channel through interaction with an honest node. Therefore the attacker cannot possess the primitives needed to compose a valid encrypted data that a receiver would accept (successfully decrypt with error).

Thus it is shown that using the SAE protocol as an initial handshake to secure

[2] `https://tools.ietf.org/html/rfc793` Dec. 2019

a unicast channel satisfy all the properties of the threat model and the system requirements.

3.4.2 Secure Storage

As per the system requirements, the local vault fragment stored on a device must be protected using encryption. In order to ensure the data is not changed by any other process running on the device the encryption operations must use an AE scheme for data integrity.

The obvious choice for the secure primitive used as a key to these encryption operations is that of the user's master password derivative used for user/node authentication, since this is already stored in the DPM process when the user starts the application. However, the american agency for standards, National Institute of Standards and Technology (NIST), has published a comprehensive set of papers detailing their recommendation for digital key management. In [NIS16], they recommend never re-using the same secret keys, whether in the same application context or in different contexts.

Furthermore, the threat model in section 3.1 grants an attacker the ability to break any storage encryption on a node's device. Assuming this would compromise the key used to encrypt the data the result of such an attack would grant the attacker knowledge of the user's master password key derivative. While this would not allow the attacker to directly access a node (the derivation function cannot be inverted to derive the original master password used in local user authentication), this knowledge could be used in the remote authentication flows when establishing secure connections in the network. This would break the temporary trust boundary in the system and thus invalidate such a design choice.

Following the NIST recommendation and in compliance with the threat model, a second-order derivative of the user's master password must be used as the secret encryption key when storing secure data on the device. Thus, the local device encryption key $k_{storage}$ is defined as shown in eq. (3.4), where $KDF(b)$ is the key derivation function of base key b and mp is the user's master password.

$$k_{storage} = KDF(KDF(mp)) \tag{3.4}$$

Since the derivation function is not invertible, breaking the local storage encryption and deriving $k_{storage}$ would only reveal the second-order derivation

secret and not expose the first-order derivation secret that is used in the remote authentication flows.

3.4.3 KDF & Hashing

The security design makes a distinction between hashing general data and hashing secret keys/passwords. When a scheme needs to compute a secure one-way data transformation on a secret key it must use a key derivation function, which has stronger security and more configurable parameters than general hashing algorithms. When the data is not a secret key it is sufficient to use a general hashing algorithm.

This design choice follows the recommendations from NIST in [NIS16], and from European Union Agency for Cybersecurity (ENISA) in [ENI14].

3.5 System Modelling

The final step of defining the the general DPM design is to model its functionality and provide a preliminary model for its implementation.

These functionalities are discovered using a use case model consisting of several use cases describing a user's interaction with the system – the interactions required for the user to achieve some desired goal. This use case model is then analysed to produce two subsequent models: a tentative model for the minimum set of components required to satisfy the use cases, and a series of interaction diagrams that realize the use cases by describing the interaction flow between the internal components in the system and how they achieve the user's goal.

The modeling is done in accordance to the UML2 standard and is in part inspired by the analysis and design phases in Unified Process (UP), as covered in [AN13].

The design choices for the architecture, data flow, and security previously documented in this chapter all serve to inform this modelling process. These design specifications defining the limits and requirements of the elements that go into the use cases, their components, and interactions.

3.5.1 Use Case Model

The purpose of the use case model is to specify the user's interactions with the system in order to achieve a series of goals. Normally these use cases would be found during a discovery phase involving system engineers, domain experts, and product stakeholders, but for the purpose of this project, the author mebodies all of these roles, discovering and defining the the user's goals and desired interactions with the system, as set forth by the thesis objectives and the system requirements.

The model is visualized in fig. 3.4, and the concrete use cases are specified in appendix B. The model covers the basic features defined in the system requirements, allowing a user to start and configure the application, sign-in, create and delete vault entries, view and search the vault entries, and finally sign-out. Editing entries has been omitted since it is semantically equivalent to deleting an entry and adding a new one with the corrected data – there is no internal or external state that is affected by this beyond the isolated vault entries.

The model also covers the more system-technical use cases needed for the application to interact with the rest of its node network, whether it is for getting the network's vault fragments or notifying the network of new fragments.

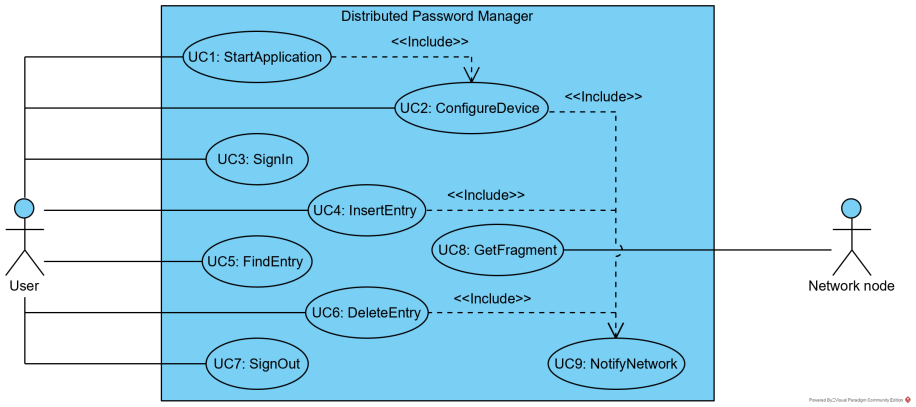

Figure 3.4: Use case model overview.

One important aspect of use cases is their ability to capture the system's requirements. If a use case does not satisfy any requirements it is either unnecessary, not detailed enough, or there is a hidden requirement that has not yet been discovered. Similarly, if there are requirements that are not captured in any use case then either the use cases are not detailed enough, or there are hidden use cases that have not yet been discovered.

A requirements traceability matrix can be used to visualize this mapping between use cases and system requirements, as shown in table 3.3. Each requirement is mapped to at least one use case, and there are no use cases that do not map to any requirements, indicating that the model sufficiently covers the system's requirements.

	Use Case								
	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9
Features									
FR1				✓					
FR2					✓				
FR3				✓		✓			
Security									
SR1		✓	✓					✓	✓
SR2		✓	✓					✓	✓
SR3			✓					✓	✓
SR4			✓						✓
SR5								✓	✓
SR6	✓		✓				✓	✓	
SR7			✓						
SR8	✓		✓					✓	

Table 3.3: Requirements traceability matrix.

3.5.2 Analysis Classes

The set of preliminary components/classes required to satisfy the functionality described in the use case model is discovered by analysing the use cases and the resources they were made from, e.g. the system architecture specification, system requirements, and thesis objectives.

There are many different methodologies for discovering the preliminary components, some of which are described in [AN13]: noun/verb analysis, Class, Responsibility, Collaborators (CRC) analysis, and Rational Unified Process (RUP) stereotypes. The CRC and RUP methodologies are secondary procedures that usually build on top of preliminary noun/verb analysis and serve to refine the model. For the purpose of this project the noun/verb analysis is sufficient for analysing the small use case model, its requirements, and the other design specifications (architecture, data flow, and security models).

The discovered classes are shown in fig. 3.5. The operations and attributes of

the classes are omitted for the sake of simplicity and readability.

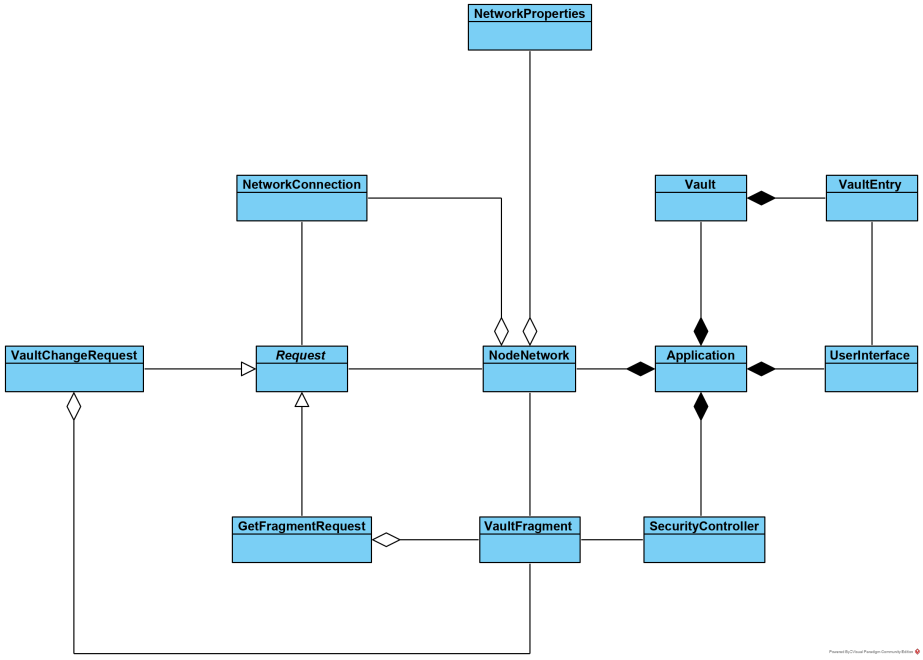

Figure 3.5: Class diagram of analysis classes.

3.5.3 Use Case Realization

The final part of the design phase is the use case realization. This model binds together all of the other design models by demonstrating how the preliminary components should behave in order to realize the use cases and thus achieve the user's goal. Therefore this is behavioral system model that defines the core behavioral design for the DPM system.

The artifact of this analysis process is an activity diagram model, with the diagrams specified in appendix C. Table 3.4 shows the mapping between the use cases and the activity diagram that realize the use case. It should be noted that not all the diagrams are shown in this mapping, as some of the activity diagrams do not map directly to a use case but is instead a sub-diagram encapsulating system behavior that is relevant to several use case realizations and is therefore abstracted into its own diagram.

The choice of activity diagrams as the behavioral model over other models such

Use Case	Activity Diagram
UC1 StartApplication	Figure C.1 Start Application
UC2 ConfigureDevice	Figure C.2 Configure Device
UC3 SignIn	Figure C.7 Sign In
UC4 InsertEntry	Figure C.8 Insert Entry
UC5 FindEntry	Figure C.9 Find Entry
UC6 DeleteEntry	Figure C.10 Delete Entry
UC7 SignOut	Figure C.11 Sign Out
UC8 GetFragment	Figure C.12 Answer Network Discovery
UC9 NotifyNetwork	Figure C.5 Send Network Fragments

Table 3.4: Use case and activity diagram mapping.

as communication diagrams, sequence diagrams, or finite state machines as proposed in [AN13], were made due to their simple syntax and readability, relative to the other alternatives. Activity diagrams do not require a strict mapping between the analysis class model and the activities in each diagram, freeing up the system architect to choose a more free wording when expressing the system activities, which in term are often more readable to product owners and domain experts. Despite this, the model must still reflect the other design specifications in order to tie the models together in one coherent design model.

With the architecture, preliminary components, security principles, and system behavior defined in the DPM design, the next step is to refine this grand model and implement it in a proof-of-concept prototype.

Chapter 4

Implementation

In order to demonstrate the correctness of this new design model—that is, the effectiveness of the new security properties—the DPM design is revised and implemented as a proof-of-concept prototype. This chapter documents this implementation.

The documentation covers the language and development environment chosen for this prototype, a revised system model for binding the design model and implementation together, and a detailed discussion on the cryptographic schemes chosen based on the security model.

A digital copy of the prototype implementation code is accompanying this thesis report, but the code base is also made available online[1].

4.1 Language & Environment

The DPM design is language agnostic and makes no presumptions on the language required for its implementation, so the choice of coding language and

[1] `https://github.com/denDAY04/thesis-code`

platform was at the author's digression. Table 4.1 shows the top 5 programming languages by usage as of Dec. 2019[2].

#	Language	Usage
1	Java	17.3%
2	C	16.1%
3	Python	10.3%
4	C++	6.2%
5	C#	4.8%

Table 4.1: TIOBE top 5 programming languages as of Dec. 2019

Java executes in its own platform agnostic runtime environment so cross-platform projects do not have to consider writing or compiling code to specific platforms. This means any code written in this language is easily portable between platforms, which is a very important feature for DPM, where the application is meant to be used on different devices that may very well use different operating systems and thus require different platform support.

Java also has a strong concurrency and networking API, features which are readily required in a distributed system such as DPM is. The language has a well-developed community with many open-source libraries. It is also supported by several mature third-party project-management and build chain frameworks such as Gradel and Apache Maven.

Both Python and C# shares many of Java's properties, but Python's syntax language rules are very different from Java and the other languages in the top 5. It is also a scripting language as opposed to a strongly typed compiled language. C# is almost identical to Java in all aspects, but it does not have as big of an open-source community and is not as well-established for cross-platform use due to its initial limitations to Microsoft Windows platforms (though recent open-source initiatives have attempted to change this).

The C and C++ languages are typically used for more low-level or performance-critical applications such as system drivers, operating systems, and embedded applications due to their complex but preferment system API and resource management. Code written in these languages must be compiled to platform-native applications that do not execute in any kind of runtime environment, thus the responsibility of cross-platform support is entirely on the developer to build for the proper platform targets.

[2] `https://www.tiobe.com/tiobe-index/` Dec. 2019

Based on these language characteristics, and existing familiarity with the language, Java was chosen for this prototype implementation. The Standard Edition (SE) version 11 of the language is used due to its recent but stable and well-supported release, and it has arithmetic operations on large numbers that older version of the language lacks – a feature that is required for many cryptographic operations.

Because of the inherent cross-platform support from the language and its runtime, the prototype was not developed with any particular platform or operating system in mind. Nevertheless, the prototype was developed and tested on Windows 10 systems.

4.2 External Libraries

The implementation makes heavy use of third-party open-source libraries in the field of *cryptography*, *logging*, *User Interface (UI)*, and *testing* where the native support in the Java API is not sufficient for the needs of this project. These libraries and their licenses are briefly documented in the following sections.

Cryptography

The Java API already contains a strong security and crypto base, but for some of the cryptographic schemes (section 4.4) the Java API does not provide the required primitives and protocols. For this reason, the Bouncy Castle crypto library is included in this implementation.

This library provides a wide range of cryptographic schemes for Java and is provided under Bouncy Castle's own open-source license[3], which is inspired by the MIT license.

Logging

In order to help debugging any errors in the system it had to support logging to external log files. This is implemented using the SLF4J library. It is a simple yet powerful implementation for the open-source log4j library that allows

[3] `https://www.bouncycastle.org/licence.html` Dec. 2019

logging to be configured using simple application properties rather than complex configuration code.

SLF4J is provided under the open-source MIT license, and log4j is provided under the open-source Apache license 2.0.

User Interface

The text UI is handled using Text IO, a library that provides a text UI in the system's console, or optionally opens a Java UI element providing a text terminal if the default console is not accessible. It includes property-configuration like the logging framework for setting different appearance properties in the console and has a very an agile API for interacting with user input, prompts, and messages.

This library is provided under the open-source Apache license 2.0.

Testing

Unit tests are written during the development of the prototype in order to provide continuous testing of the implementation's components and ensure they conform to their feature responsibilities. These tests are created and executed using JUnit 5.

This library is provided under the open-source Eclipse Public License 2.0[4].

4.3 System Modelling

The system model documents the internal components and their inter-component behavior, which was developed in the prototype in order to implement the DPM design model. It builds on the the component and behavioral models from the design, and was inspired by the design and implementation phases from UP, with UML diagrams as documentation.

[4] `https://www.eclipse.org/legal/epl-v20.html` Dec. 2019

4.3.1 Deployment Model

The deployment model encapsulates the physical artifacts generated from the implementation when it is built and deployed on a device, ready to be executed as an application. These artifacts are the physical files that constitute the DPM system, and are shown in the deployment diagram in fig. 4.1.

The core artifact—the system executable—is the *dpm.jar* file. Alongside the executable JAR file are the *lib* and *config* folders, which contain the library dependencies and the system configuration files, respectively. The only file the user ever has to interact with is the executable JAR file.

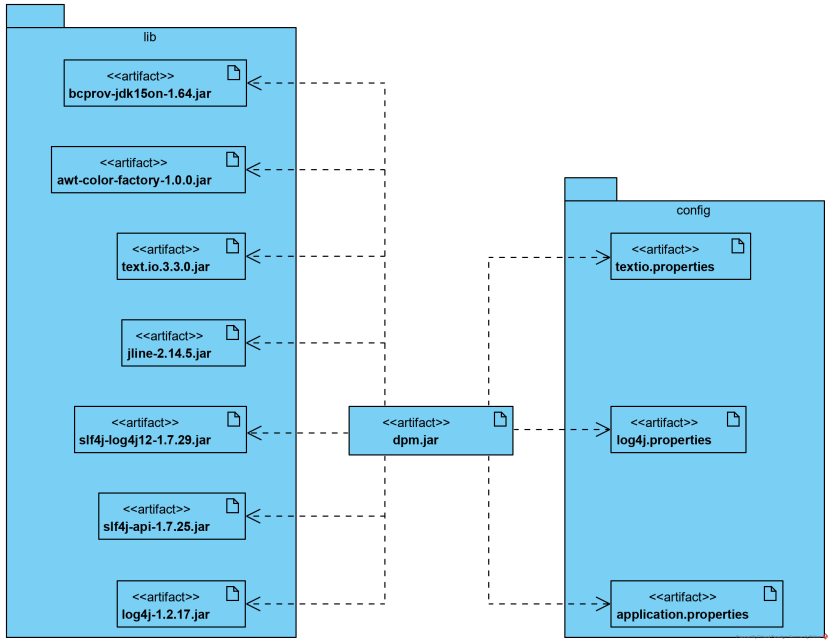

Figure 4.1: DPM deployment diagram.

4.3.2 Package Model

Although a DPM instance consists of only one executable module it contains many internal components/classes that are segmented into packages denoting areas of general functional responsibilities. These packages are shown in fig. 4.2.

The areas of responsibility for each of the packages are listed below. The base

package *dpm* contains the executable entry-point (class with main function) that also handles the initial device configuration when the application is started the first time. Any subsequent operations are delegated to the components with the relevant responsibility as denoted by their package location.

- *properties* - data classes encapsulating properties that are stored on the device.
- *vault* - the secure vault and vault fragments.
- *security* - security/crypto manager and primitive data classes.
- *util* - general utility functionality required by different packages.
- *ui* - text UI manager.
  - *actions* - available user input actions for each state in the UI.
- *network* - network I/O manager.
  - *connections* - self-contained concurrent network connections.
  - *packets* - network data packets.

4.3.3 Class Model

The class/component model that is part of the DPM design model is only a preliminary one meant to suggest the tentative minimum set of components needed to fulfill the system's use cases.

The implementation contains a refined version of this model in order to document the prototype's concrete components and their inter-relationships. These components are defined using UML class diagrams are contained in appendix D.

The model does not contain *one* complete diagram that displays all relationships between classes in the system since this is often very confusing to navigate and does not offer any concise information. Instead this model favours the use of separate diagrams that show the core relationships relevant to each package in the package model.

There are many pieces of information that could be added to the diagrams in order to carry more details; information such as component attributes and operations, relationship multiplicity, and relationship role names, even Object Constraint Language (OCL) specifications. All these additional information are

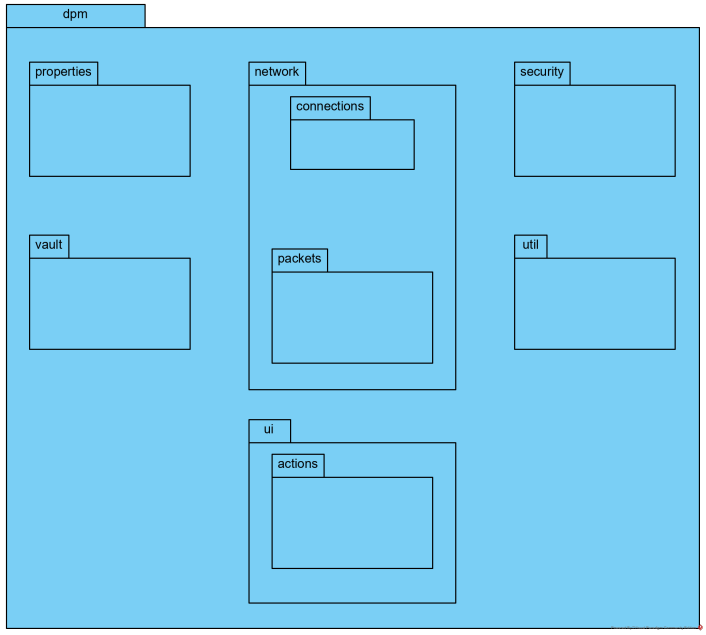

Figure 4.2: DPM package diagram.

important for strict models that must closely define every aspect of a model, whether for the purpose of proving a model's correctness or to serve as input to a process that generate/develop a system based on this model (whether by a manual programmer or an automated process).

For the purpose of simply documenting a proof-of-concept prototype, all these information would only serve to add unnecessary complexity for diagrams whose purpose is otherwise simple. This project does not propose a final product intended for production release, and thus the necessity for documenting these minute details are deemed less important. The model satisfy to documents the core building blocks of the prototype through the components and directed relationships.

4.3.4 Behavioral Model

The design model defined the system's behavior through a series of activity diagrams that realize the system's use case, and these models served to dictate the behavior the prototype must exude.

For the implementation model, the behavioral model documents some of the core behaviors in the prototype that was only developed during implementation and not part of the initial design model. These behavioral models cover the UI in order to visualize its behavioral navigation, and the concrete implementation of the inter-process communication paradigms.

4.3.4.1 User Interface

The system's simple UI implementation is described using a behavioral state machine, as shown in fig. 4.3. In principle, the system design and even this implementation model does not require the UI to text-based; it could just as well be a graphical UI and both the design and implementation model would still be valid. However, for the sake of simplicity this prototype forego this choice, since a graphical UI is more comprehensive and takes more time to develop.

It should be noted that the state machine diagram type is chosen as a simple and comprehensive model to encapsulate the behavior of the UI, but the implementation is not in fact coded as a strict state machine.

4.3.4.2 Inter-process Communication

The behavioral design model highlights the frequent use of activities that require communication with the DPM instance node network, whether it is for fetching the network's vault fragments, or updating the network with new fragments each time a change is made in the vault. And these actions are always preceded by a a network discovery query due to the nature of the dynamic peer-network.

This inter-process communication is a significant proportion of the logic in the DPM implementation, and for this reason it was decided to make use of an asynchronous concurrent communication model. This allows for the code to free up the resources in the system by using asynchronous handling of the socket communication, and multi-threaded concurrency for each communication channel. It relies on three core components:

- *DiscoveryListener* - handles discovery of the dynamic node network, and listens for these discovery requests from the network; all using multicasts.
- *ClientConnection* - connects to a unicast channel and sends a request.
- *ServerConnection* - prepares a unicast channel and responds to a request.

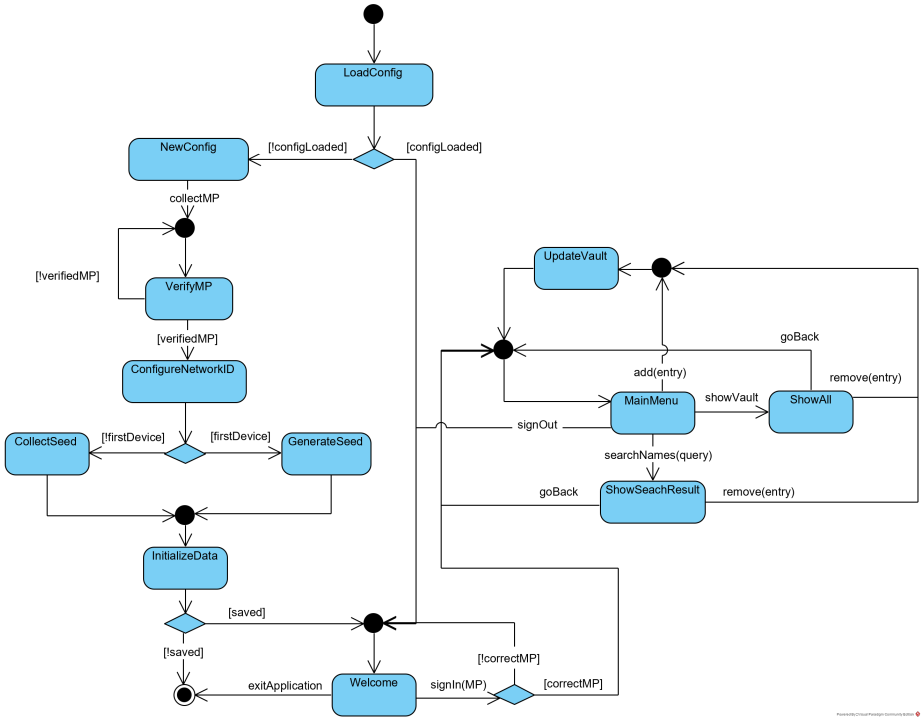

Figure 4.3: Behavioral state machine for the UI.

All these components are self-contained and run in their own threads within the concurrent model. This allows the unicast channels to be handled in parallel rather than sequentially when a node has to communicate with the node network, and offers some optimization to the processing. It also allows the system to handle the network management through multicasting to take place concurrently in the background of the application, rather than having to freeze the main UI application thread when the system needs to respond to requests from the node network.

It should be noted that the naming convention of using client and server in the names of these components are purely representative of the component's role–whether it must open the channel and receive a request (server), or if the component is expecting to connect to a channel and send a request (client). These names are inspired by the properties of client-server architecture but it is not reflective of any such architecture in the node network.

The behavior of the *DiscoveryListener* class is modeled in fig. 4.4 using a behavioral state machine. Its main task are either to listen for discovery requests,

or initiate discovery requests to the network.

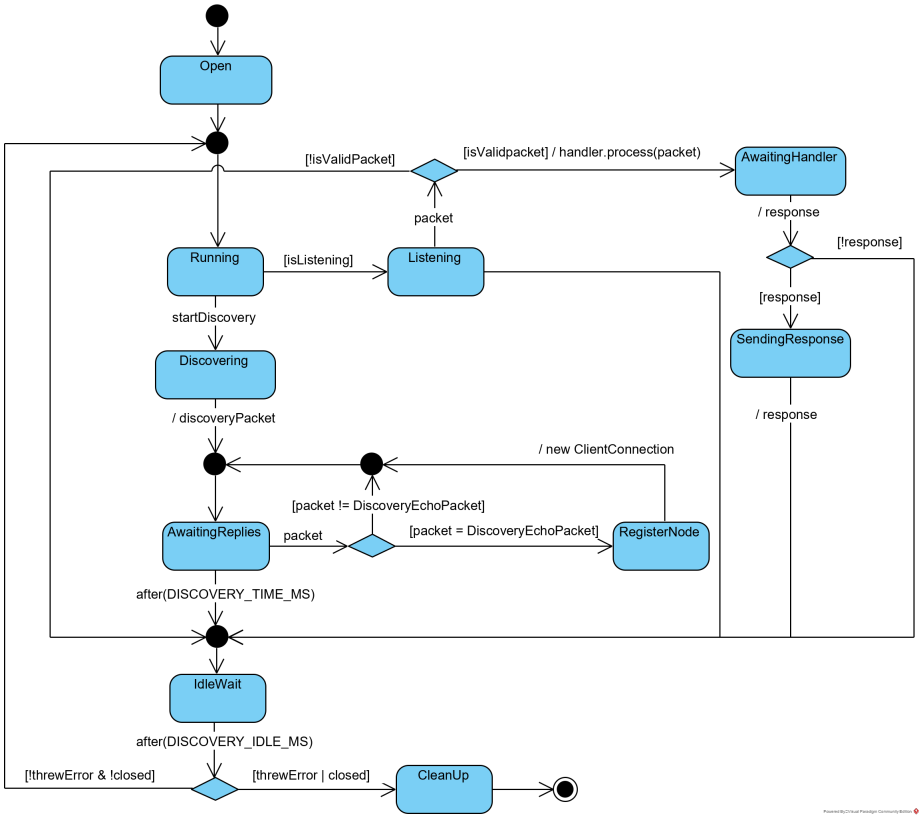

Figure 4.4: Behavioral state machine for the *DiscoveryListener* class.

Network Discovery

As shown in fig. 4.4, when the listener receives a packet from the network a handler checks whether the packet is valid (i.e. meant for this DPM instance's network). If this is the case the handler will spawn a new instance of the *ServerConnection* component, which prepares a new unicast channel. The listener will reply with a response to the node that sent the initial packet, informing this node that the new channel has been created and is waiting for connection.

On the other node, upon receiving this reply, knows that it has discovered another node in the network and will then spawn an instance of the *ClientConnection* component, which then connects to the channel. Thus the network

discovery has been carried out between two nodes, and any subsequent requests can be made on the unicast channel

Unicast Communication

Once a connection has been established between two nodes a very simple protocol is used to handle any possible communication, as is shown in the sequence diagram in fig. 4.5.

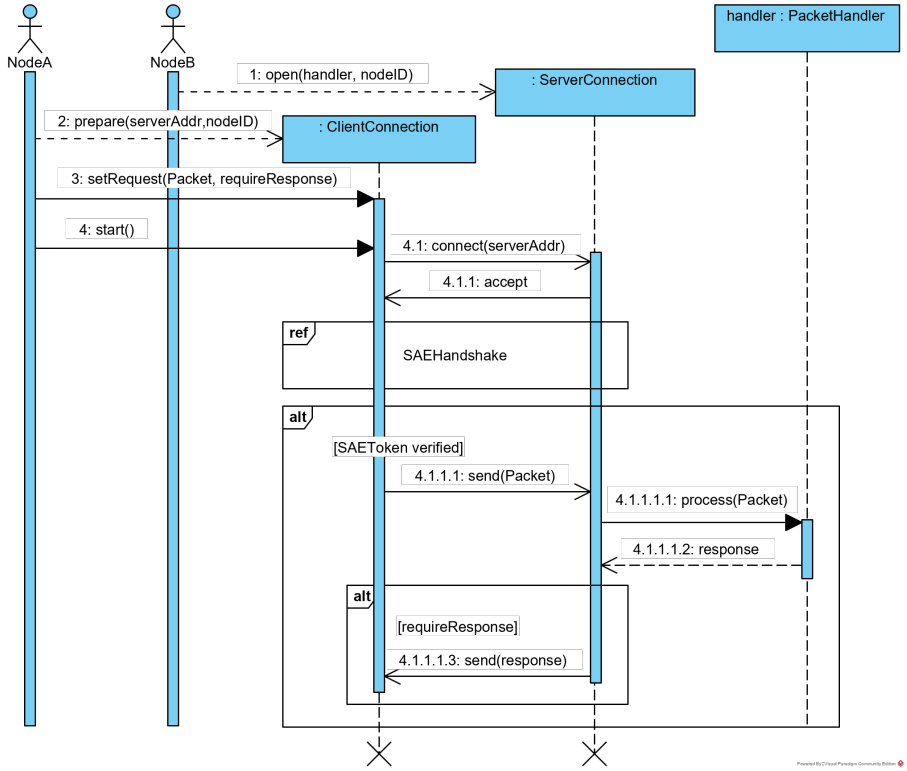

Figure 4.5: Unicast communication sequence diagram between *ClientConnection* and *ServerConnection*.

Initially, once the client node A has connected to the server node B, the protocol starts with an SAE handshake, which use the the SAE protocol to establish mutual trust between the nodes and authenticate each other. If this handshake succeeds the client will send its request and the server node may optionally respond, but is not always required to, e.g. when the request is simply

a new/updated vault fragment.

The flow for the SAE handshake is shown in sequence diagram fig. 4.6.

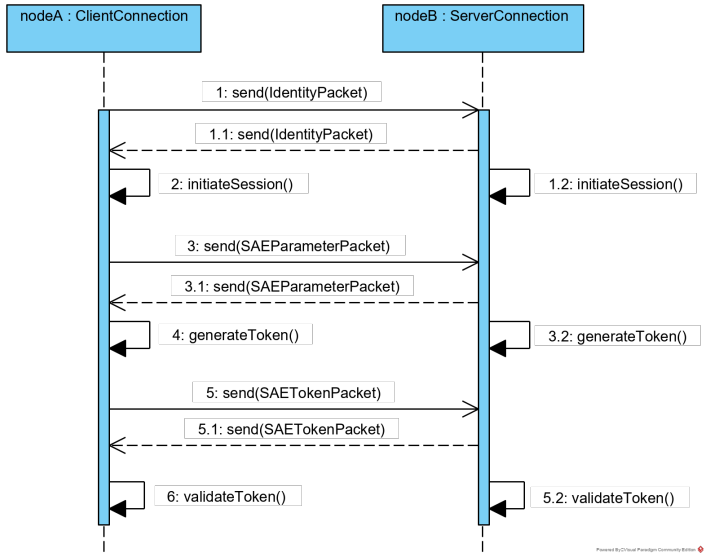

Figure 4.6: SAE handshake sequence diagram.

4.4 Cryptographic Schemes

The security model in the design defines a range of specification and general choices for the cryptographic schemes and primitives that are required, but it did not specifically define which concrete schemes, algorithms, and primitives needed to be implemented.

Table 4.2 lists the schemes chosen for this prototype implementation, informed by the security model in the DPM design. These choices are discussed along with possible alternatives in the following sections.

4.4.1 SAE Finite Cyclic Group

The specification in [Har08] defines two different schemes possible for implementing the SAE protocol with its finite cyclic group: either the group can

Type	Scheme
SAE Finite Cyclic Group	Elliptic Curve - Curve25519
Random Number Generator	DRBG
Hash Function	SHA3-256
Key Derivation Function	PBKDF2 with SHA3-256 HMAC
Encryption	AES/GCM

Table 4.2: Cryptographic schemes chosen for the DPM implementation.

be a prime modulus group, or a finite prime field over an Elliptic Curve (EC). The EC design has several advantages over normal prime group, and is also the recommended choice in [Har08].

EC computations are inherently difficult to reverse when operating on specifically designed secure curves, and as a result you can achieve better security with small-sized keys, compared to normal prime groups—of which the reverse operations are less difficult, though still very difficult. This is highlighted in the analysis commentary in [Eff09] where it is shown how the strengths of EC boosts the strength of encryption keys such that an EC key of 256 bit has equivalent security as that of a typical 3072 bit RSA encryption key. A massive efficiency boost for no loss in security.

Furthermore, common prime groups has also shown to have pre-computation vulnerabilities, minimizing their effective security. This vulnerability, known as *logjam*, is presented in [Adr+15] and was found in common key-exchange protocol implementations in libraries that use common prime fields and properties. It speculates that the resources necessary to utilize this vulnerability is within the scope of national agencies, and would allow the attacker to break the encryption of a secure channel. As a result, the analysis advocates the use of EC schemes since these are not vulnerable to the pre-computation attacks, or using much greater prime fields with different properties.

For the choice of the specific curve design, [Ber] was the primary source of analysis as it documents many common curve designs, their characteristics, and known vulnerabilities. For this reason, *Curve25519* was chosen for its secure properties and its industry prevalence, since this curve has been widely adopted in the industry as the go-to alternative curve that is outside the NIST standard[5]. This is shown—by among others—by the Internet Engineering Task Force (IETF) as they adopted this curve and Curve448 in the TLS 1.3 protocol

[5]The industry is moving away from the NIST curves as there is a prevailing suspicion surrounding the curves' parameters and whether they might allow a backdoor.

standard proposed in RFC8446[6].

4.4.2 Random Number Generator

The Deterministic Random Bit Generator (DRBG) is a scheme for generating pseudorandom bits using deterministic algorithms with an entropy source, and is recommended by NIST in [NIS15a]. The generation of bits makes it possible to use this mechanism for generating any amount of random data since a string of bits can be interpreted as a series of data bytes, integers, or floating points.

This Random Bit Generator (RBG) is a default implementation in Java and is included by its standard SUN provider[7].

Another popular RBG scheme is defined as part of the PKCS#11 suite of cryptographic mechanism, but the support for this scheme is not as readily available among java security providers as it require access to native PKCS#11 libraries and does not offer an implementations of the cryptographic functions in the provider module. Due to this configuration miss-match the choice of DRBG was made, following the NIST recommendation.

4.4.3 Hash Function

The choice of a SHA3 algorithm is based on the standard defined by NIST in [NIS15b]. It is the most recent and secure iteration of the SHA family of cryptographic algorithms and schemes.

The primary use of the hash function for this system's security model is to transform data derived from EC computations used as part of the SAE protocol. The chosen *Curve25519* design has a bit-resolution of 255 for its field-size, meaning the use of a 256-bit hash function will ensures that any output from this function has at the bit-resolution to cover the full field-size of the curve, and therefore the size to cover the full field of values found on the curve. Using a hash function larger than this is not necessary for the security model and would only impact performance, which is a significant drawback when used during inter-process communication.

[6] `https://tools.ietf.org/html/rfc8446` Dec. 2019
[7] `https://docs.oracle.com/javase/9/security/oracleproviders.htm` Dec. 2019

4.4.4 Key Derivation Function

A very popular Key Dereivation Function (KDF) scheme is PBKDF2, which is recommended by IETF in their password-based cryptography specification[8] and by ENISA [ENI14]. This scheme support using a hash-based RBG to strengthen the entropy of the derivation function, and for this implementation the use of the SHA3-256 algorithm was chosen, for the same reason as in section 4.4.3. The key output key size is specified as a recommended 128 bits.

Better security could be obtained by using greater key output sizes, but this added security is not necessary since additional complexity is added to the scheme through the use of EC over common prime cyclic groups, as well as the additional entropy added by the SAE protocol. Additionally, a greater-length key output would result in a slower key derivation process, which is not insignificant for this application, where the KDF scheme is used iteratively as part of the SAE protocol, in the process of computing valid points on the curve. Thus a performance hit on the KDF scheme would affect the performance of the system exponentially.

Iteration count is one of the parameters offered in KDF schemes, defining the number of iterations the function to run, each one generating a new derivative of the previous function output. As per the implementation recommendations made in [ENI14], by IETF, and in [NIS16], the implementation uses an iteration count of 1000 to increase the entropy strength of the derived key. This is an appropriate tradeoff between key strength and time-complexity required to generate a key, which is a sensitive performance matter, as mentioned in the previous section.

Salting is an integral part of the KDF schemes in order to avoid dictionary attacks, yet the implementation in this system forgo this feature in some cases. It is *not* used during the SAE protocol, *nor* is it used for the temporary storage of the user's master password in-memory; it *is* however used for the encryption of the local vault fragment stored on the device.

The SAE protocol inherently implements its own salting feature through session-local random variables that are part of of the key computations; this makes the KDF salting unnecessary. Additionally, the introduction of salting at this stage would introduce a new variable that would have to be transmitted between the

[8] `https://tools.ietf.org/html/rfc8018` Dec. 2019

two nodes participating in the protocol, as they must both use the same salt in order to arrive at the same intermediary key.

Similarly, using KDF salting for the temporary master password introduces additional data that must be exchanged between the nodes prior to initiating the SAE protocol handshake, to ensure they feed an identical key derivative of the master password into the protocol. The dictionary-attacks are also not a threat for these temporary derivatives since they are only kept in-memory and are not exposed to any readable outputs without going through the SAE mutations first. This means an attacker cannot gain access to read the derivative and compare it against a compiled dictionary.

The only place the salting is used is for generating the key derivative used during encryption and decryption of the local vault fragment stored on the device. Since this representation of data is not dependant upon by any external node or actor, the system is free to use salting to add the additional security. The salt is stored in the same file as the encrypted vault fragment data, as the first 128 bits of data - the same length as the KDF output key itself.

Popular alternatives to the PBKDF2 scheme are BCrypt and its successor SCrypt. [ENI14] recommends these implementations over PBKDF2 since they add better resilience against dictionary attacks, but they also have a performance hit due to this added protection. For the purpose of this system and its prototype the PBKDF2 standard offers sufficient levels of security and the added protection against dictionary attacks is not deemed necessary.

4.4.5 Encryption

All encryption in the system, both on the secure channels and of the local vault fragment data, is handled by symmetric block cipher encryption schemes, of which Advanced Encryption Standard (AES) is one of the most popular schemes. This is recommended by ENISA ([ENI14]), NIST ([NIS16]), and by OWASP[9].

The encryption is carried out using the 128 bit key output from the KDF scheme, since this has the least noticeable hit on performance compared to 192 and 256-bit keys which are 20 and 40% slower, respectively, according to [ENI14]. 128 bit keys are still proven to be secure with no feasible way of breaking the encryption, when it is run in a proper operation mode.

[9] `https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html` Dec. 2019

The operation mode for the scheme, as per the recommendations cited, is Galois/Counter Mode (GCM) standardized in [NIS07]. This mode offers the authenticated encryption required by the system design, ensuring data confidentiality and integrity.

Following the NIST recommendation [NIS07] the encryption uses a 96-bit Initialization Vector (IV) that is generated using the prototype's RBG to ensure unique randomness, which is required by GCM's security model to prove its security. The choice of 96 bits is a balance for optimal security and balanced performance. A smaller IV compromise the degree of security for the encryption, while a larger IV has been shown to introduce speculative vulnerabilities due to the truncation methods in the operation mode ([ENI14]).

A stream cipher of any given scheme could have been an alternative to the chosen block cipher implementation, but these are generally more suited for very large amounts of data where the use of block ciphers are simply too inefficient or the initial size of the data to be encrypted is unknown. Due to the rather finite amount of data used in this prototype (a vault fragment would never exceed more than a few megabytes at most) block ciphers offer better security with minimal performance impact.

4.5 Data Storage

The system generates two data storage files during execution: *network.prop* which stores the application's network properties, and *vault.frag* which stores the local vault fragment. These files are generated in a new *data* directory, and of the two files only the fragment file is encrypted, since the network data is not secret and does not need confidentiality.

The network properties files includes the node's unique ID, the network ID, and the seed used during the generation of the network ID. The fragment file contains the encrypted data, and is prefixed with the 96-bit encryption IV and the 128-bit KDF salt.

4.6 Configuration

The project makes use of two different kinds configuration: static and dynamic.

The static configurations are highlighted in section 4.3.1, as the **.properties* files deployed in the *config* directory. These configuration files define the static properties of the application, such properties as the UI look, logging format, and storage paths, to mention a few.

The dynamic properties are those of the network properties file, as mentioned in section 4.5. These properties are not pre-defined and are part of the deployment model; instead they are generated by the application as part of its initial configuration phase.

All these files, wether dynamic or static, are used by the application and are not intended for the user to directly interact with them.

Chapter 5

Evaluation

With the system prototype finished, only the validation of the design and the implementation remains. The evaluation is carried out on the basis of tests – both for the functionality of the system and its compliance with the requirements model.

It is followed by an evaluation of the implemented security model in respect to the design's threat model; this is emphasized due to the nature of this thesis and the objective's focus on proposing a new password manager design with increased security. And finally, reflections are made on the core design decisions and their impact on the prototype implementation; specifically in respect to how these decisions could be changed or otherwise made better in future works.

5.1 Testing

The testing of the prototype implementation has been split into two categories. One category is the mandatory acceptance tests that ensure the developed prototype satisfies the system's requirements model and the objectives of this thesis. The other category is functional tests carried out during the implementation phase.

5.1.1 Functional Tests

During implementation a continuous model of functional tests were developed to compliment the implementation of the system's core components, and these tests took the form of unit tests. These form of tests validate the correctness of each core functionality of the system's internal components.

The tests were included in order to help the implementation effort, being able to test the functionality of isolated components prior to their integration with other parts of the system. This helped to ensure the completeness of the full functionality of the system and is a powerful tool that saves time when trying to triangulate functional errors that may arise during system-wide functionality tests such as integration or end-to-end tests.

Because the tests compliments the implementation phase, the tests themselves are part of the prototype's codebase, and are located in the *src/test* directory of the system's codebase. The summary of the tests are listed in table 5.1, by the component test class and its individual unit tests.

Component / unit-under-test	Status
BufferHelperTest	
Get data from random-access buffer	Passed
Get data from int buffer	Passed
Get data from byte buffer	Passed
NetworkInterfaceHelperTest	
Get active network interface controller address	Passed
Get active network interface controller	Passed
StorageHelperTest	
Create and get path	Passed
PacketTest	
Packet serialize/deserialize	Passed
PropertiesContainerTest	
Load properties from file	Passed
SecureVaultTest	
Build vault from fragments	Passed
Fragment vault	Passed
Remove entry from vault	Passed
Get all vault entries	Passed
Search vault	Passed
Build empty vault	Passed
Add entry to vault	Passed
Get builder	Passed
Add invalid fragment to builder fails	Passed

Add valid fragment to builder	Passed
Add fragments complete the builder	Passed
Build vault fails when incomplete	Passed
SecurityControllerTest	
Save fragment	Passed
Load fragment	Passed
Get singleton instance	Passed
Verify master password	Passed
Encryption/Decryption	Passed
Get random generator	Passed
Encryption/Decryption fails for different base key	Passed
Initiate SAE session	Passed
ECC is thread safe	Passed
NetworkPropertiesTest	
Generate network properties	Passed
Load from file	Passed
Load from invalid file returns null	Passed
VaultFragmentTest	
Add fragment to builder fails for bad index	Passed
Add fragment to builder	Passed
Get builder	Passed
Get builder fails with invalid size	Passed
Build fragment	Passed

Table 5.1: Functional unit test summary

5.1.2 Acceptance Tests

While the functional tests are useful for the implementation phase, they are not sufficient for validating the final and complete state of the solution and its ability to satisfy all the system requirements when it is operating.

For such validation acceptance tests are a more fitting tool. The acceptance test process validates the software's completeness by evaluating a complete system's ability to satisfy its requirements when in proper use by a user, and not in artificially segmented test environments that only operate a component or groups of components at a time.

The use case model defined in section 3.5.1 captures all the requirements and formed the basis of the system's behavioral implementation. This makes the use case model a candidate for the execution and evaluation of the acceptance

tests, since these use cases define concrete actions and expected results from the user's point of view when using the system.

By having a test agent assume the primary role in each use case, the agent can execute the role's actions in each use case and validate the results and correctness of the implementation and compare it to the use case specification. If the test passes the specification the system satisfy the requirements captured by this use case. And so the process repeats until all use cases have been tested.

This process is documented in appendix E, and the results are summarized in table 5.2. As can be seen, the solution satisfies the system requirements by verifiably implementing the use cases.

Use case	Status	Documented
UC1	Passed	Appendix E.1
UC2	Passed	Appendix E.2
UC3	Passed	Appendix E.3
UC4	Passed	Appendix E.4
UC5	Passed	Appendix E.5
UC6	Passed	Appendix E.6
UC7	Passed	Appendix E.7
UC8	Passed	Appendix E.8
UC9	Passed	Appendix E.9

Table 5.2: Acceptance test results.

Some screenshots of the running prototype are shown in figs. 5.1 to 5.4. These do not document all available screens and features from the application, but serves to demonstrate the look-and-feel of the prototype and the user experience.

5.2 Security Model

Although the implementation of the security model is inherently evaluated through the use cases and the acceptance testing, as it is part of the functional use of the system and its features, it is also worth evaluating it directly against the threat model (section 3.1).

In the model an attacker was assumed to have the following properties:

- Can gain access to a node's vault storage.

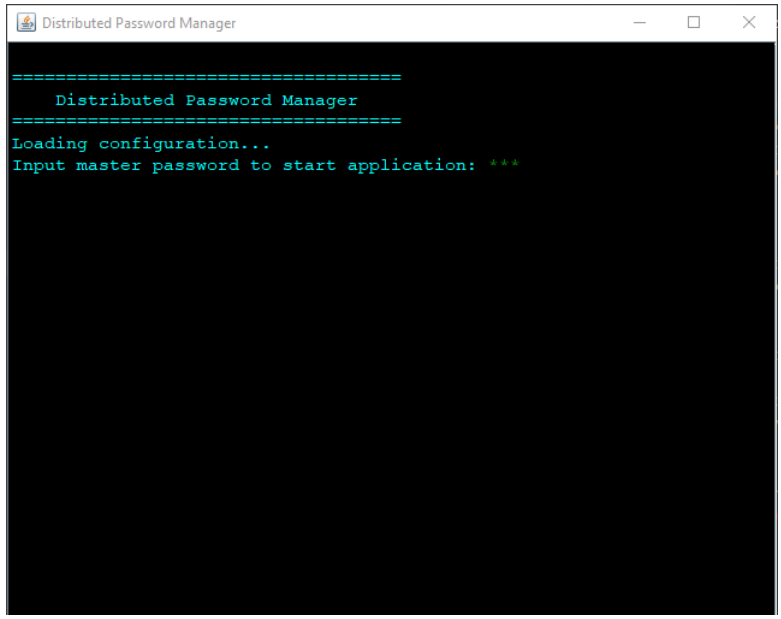

Figure 5.1: Prototype screenshot - startup

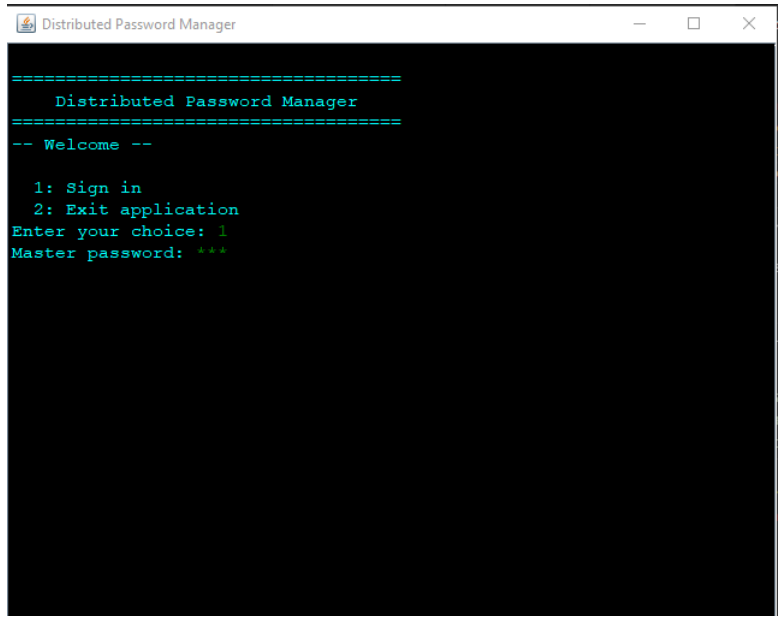

Figure 5.2: Prototype screenshot - log-in

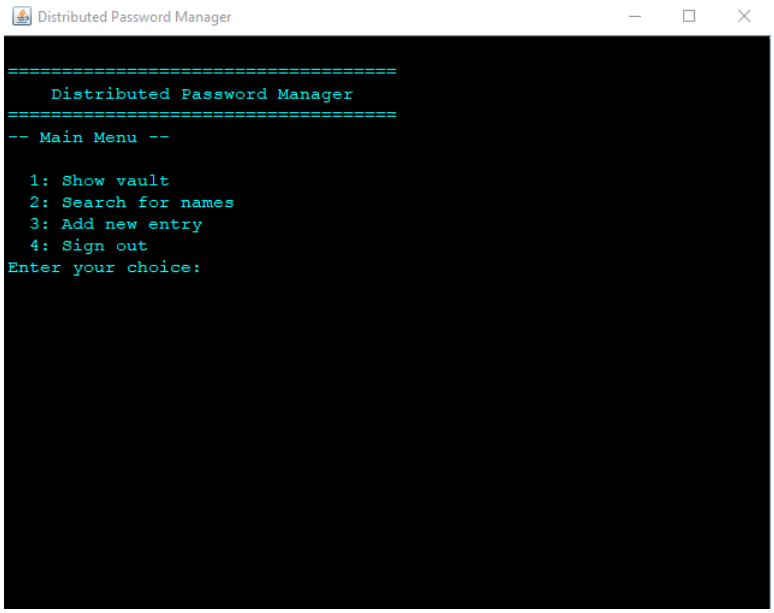

Figure 5.3: Prototype screenshot - menu

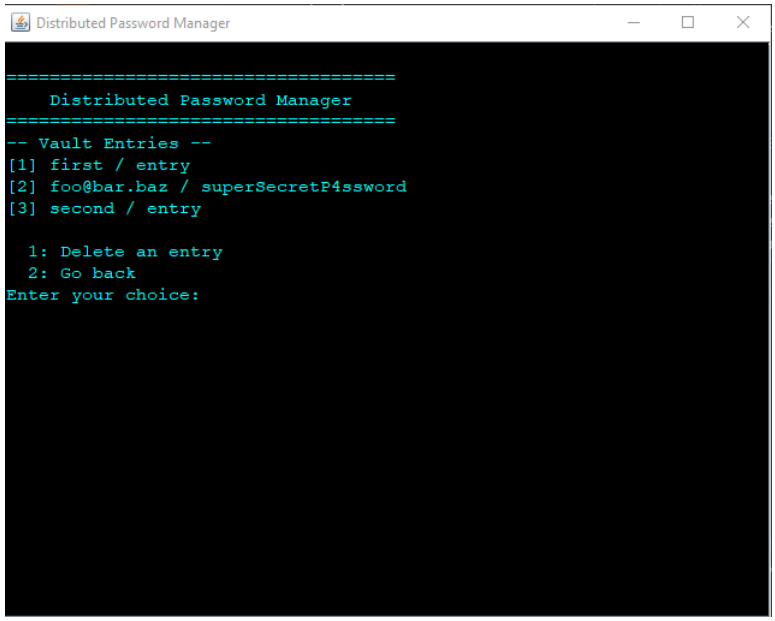

Figure 5.4: Prototype screenshot - vault

- Can compromise the secured vault storage.
- Can emulate/corrupt a node's behavior.
- Can intercept data communicated in the node network.

The countermeasures implemented by the security model and the DPM design in response to these properties are shown in table 5.3. The attacker's access to the node's storage is countered by ensuring the sensitive vault fragment data is encrypted (sections 3.4.2 and 4.4.5) and protected against tampering. This is only trivially secure due to the attacker's second property that allows for it to *eventual* be broken at some time in the future.

Being able to *eventually* break the secure data storage, the attacker gains access to the vault fragment data. To counter this and ensure the compromised data does not reveal usable secret data the security model fragments the sensitive data and decentralize it by distributing it into the node network. The entropy of this fragmentation is the core defense of this countermeasure, as discussed in section 3.3.3.

Attacker Property	Security Model Countermeasure
Gain Storage Access	Authenticated data encryption
Compromise Secure Access	Decentralized fragmented data
Corrupt Node behavior	PAKE secured channels
Intercept Communication	Message encryption

Table 5.3: The countermeasures to the threat model's attacker properties implemented in the DPM design and prototype.

In order to keep a corrupted node from maliciously interfering with the rest of the network or gain access to the full secure vault (by legitimately obtaining the other nodes' fragments) the system implementation makes use of the SAE protocol, a PAKE scheme designed to ensure mutual authentication in a P2P network without ever transmitting sensitive data (section 3.4.1). This implementation ensures the corrupted node cannot gain access to the rest of the network as it will not possess the secret knowledge (the user's master password) required to gain the authenticated trust of the other nodes in the network.

The SAE protocol also ensures additional security properties against eavesdropping, replay, passive, and active attacks, making it useful for establishing a secure channel of communication. The product of this protocol is a shared secret key, a derivative of which the system uses to establish a secure connection through data encryption. Using authenticated encryption the channel ensures

both confidentiality and integrity (section 3.4.1), with the underlying channel architecture ensuring reliability (section 3.2.2).

Thus this thesis has shown how the DPM design and the prototype implementation successfully has addressed the security properties raised in the system's threat model, and proposed a design for a password manager with strengthened security compared to the industry solutions and academic models, as analysed in this thesis (chapter 2).

5.3 Future Improvements

As a simple prototype that serves as a proof-of-concept there are many aspects that could be improved in the implementation and its design, given further time. Some of of these aspects have already been highlighted ín the design chapter of this thesis.

5.3.1 Network Model

Chief among these possible improvements is the network structure. The choice of a simple networking model that rely on multicasting and unicasting, as documented in section 3.2.2, works for the simple prototyping for this thesis, but in a production system this model is simply not scalable.

The biggest problem of this model is the unreliable support for multicasting in most public network. The developed prototype use IPv4 for its network communication, and this IP model has very unreliable support for multicasting. Many networks only allow these messages to be send within the LAN group, and some networks are even configured to block it entirely.

IPv6 does offer more standardized support for multicasting, but it still faces the administrative issues of networks being configured to partially or fully block this method of communication, or severely limit it for corporate or network administrative use.

In order to expand the prototype to work in WAN and not just LAN, the system would need a more robust networking model; one that would require an overlay network to maintain the P2P network structure and handle message routing and resource queries.

5.3.2 Vault Fragmentation Entropy

Another interesting point of further improvements is the degree of entropy introduced in the vault fragmentation process.

The threat model (section 3.1) dictates that an attacker will *eventually* break any encrypted data that is stored on a node's device. This means the attacker will at some point gain access to read the plain fragment data. The degree of entropy used when fragmenting the vault then becomes the sole security preventing the attacker from obtaining sensitive data in its entirety. The higher the entropy the harder it should be for the attacker to guess the missing data from the fragment.

Because the fragment contains a mask denoting the absolute position of each of the fragment's data bytes, the entropy cannot be further increased by simply manipulating the position of the data bytes as such a change would have to be reflected in the byte mask. Scrambling the position of the data bytes would only be a solution if the design could abstract away the mask array and devise another method to ensure the correct order of fragment bytes when combining fragments into the complete vault.

A different solution to adding more entropy, rather than scrambling the byte positions, may also be to add a type of faux/decoy entries in the secure vault, as first introduced in [Boj+10] and [Agh+16]. These entries would obscure the real data stored in the vault, adding dummy data in the fragments and increasing its entropy. Such an addition would have to be carefully designed to make the faux entries look real to an observer, and they should not hinder the user's experience with navigating the vault (i.e. they should never be presented to the user when using the DPM application).

5.3.3 Local Authentication Flow

The local authentication flow is also subject to further improvements, particularly since this issue is one that affects the user's experience when using the application: the user has to input their master password twice; once when starting the application and again when signing in, as shown in the behavioral design model.

The user has to input their password to start the application, since the system needs knowledge of the password in order to communicate with other nodes in the network. This means the application can either get this information from

the user, or store-and-retrieve it on the system. This later solution is an issue, however, as it violates the threat model.

Any encrypted version of the password stored on the device can *eventually* be broken according to the threat model, and this would provide an attacker with access to the application and its secure vault. Stroring a derivative of the password is not suitable either, as this operation is not reversible, so the application could not infer the password from this derivative. One might surmise the derivative could function in place of the password when authenticating with other network nodes, but this breaks the temporary trust boundary since an attacker in possession of the derivative would be able to complete the remote authentication flow.

The only conceivable way to solve this issue is to relax the threat model to allow for the application to store a secure representation of the user' master password on the device, whether through strong encryption or using the host system's API for storing secure data (e.g. the operating system's API for storing secret keys).

5.3.4 Availability (Always-on)

The last issue with the design and implementation that warrants further study and work is the issue of the system imposing an always-on policy. The requirements model says that every node in the network must authenticate the user, and this implies an always-on availability restriction since a node cannot authenticate another node if it is not online/available. And this was also the primary reason for not considering time un-coupled communication paradigms in section 3.2.2.

In the design proposed in this thesis, the authentication flow is an integrated part of the password manager, which requires the application to always be running and available (though the user need not be signed in). This is hardly an optimal solution, especially if the system is changed to support operating in WAN,allowing the application to be running in a cloud server rather on the user's own local devices.

Addressing this issue could be achieved by moving the remote authentication flow to a separate component, running in its own process. Such a component could be handled as a daemon process, allowing it to start on boot-up of the underlying operating system and requiring no user interaction, and would be optimal for running on cloud servers. However, this would also require the changes proposed in section 5.3.3 in order to not require the user to input their

master password every time the process is started.

5.4 Project Reflection

As per DTU requirements the reflection upon the project process and the project plan is included in the project-plan.pdf document separate to this thesis.

The project progressed through its projected phases as expected, distributing almost equal time between domain analysis, system design, and prototype implementation, with the rest of the time spend on technology research and thesis writing activities.

The resulting DPM design and prototype has shown to satisfy the objectives of this thesis and proposed a new password manager solution that offers increased security for its user, through the use of data decentralization in a P2P network. It has also showed that there are many aspects of the design that could (and likely should) be re-evaluated and further studied for the solution to become mature for product use in the industry. Until such work, this thesis joins the category of academic models without industry use, such as studied in the domain analysis in section 2.2.

Chapter 6

Conclusion

This thesis sat out to show how data decentralization in a P2P based distributed system could form the basis of a new password manager with an increased level of security compared to the standard solutions that are on the market today and academic papers. It did so by starting with defining a strict security and requirements model.

An analysis of the password manager domain showed how both the commercial market and the academic field lack solutions that satisfy these requirements. This served to motivate the work carried out in this thesis, as there was a clear need for a solution that could address the strict security model and improve the overall security of password manager schemes through peer-based data decentralization.

This thesis proposed a design for a new password manager that use such peer-based data decentralization and could satisfy these requirements, increasing the security of the system beyond that of any existing solution when distributed between user devices. This design was implemented in a prototype application written in Java 11, demonstrating the useability of such a solution as proof-of-concept.

The prototype was tested with both two and three connected user devices, demonstrating the use of data decentralization. These tests showed the solution

satisfy the strict requirements.

The thesis ended with an evaluation of the project and the result, and reflected on a set of future improvements that should motivate further work with the design in the future, both for better user experience and for a more scalable design in commercial use.

Appendix A

Literature Review

This appendix presents the full process of the literature review that was carried out in order to find literature for the domain analysis of academic research and models, section 2.2.

The final result of the reviewing process – and the full set of literature produced for the domain analysis – is found in appendix A.3.

A.1 Methodology

The primary tool used for the literature review is the snowballing model proposed by [Woh14]. This model requires a starting set of literature and grows the set by studying and evaluating an academic piece of literature's references and citations. This is done iteratively using the process described in fig. A.1.

In order to acquire a starting set of literature, and as a further tool for accessing works, two search engines were used. DTU's own academic search engine[1] was the primary choice, with Google's Scholar search engine[2] used as a secondary tool in situations where the primary search enginge did not return any results.

[1] `https://findit.dtu.dk`
[2] `https://scholar.google.com`

The initial starting set of the literature was thus found using the search engines and with the search queries *password manager*, *distributed password manager*, and *peer password manager*. These were used in order to discover literature that was within the scope and objectivs of this thesis.

In order to create a manageable scope for the snowballing model and with the aim of only processing relevant literature pieces, a scope for the process was created with the following properties

- Publication year *must not* be earlier than 2010 (i.e. further back than this year). Security analysis on models and schemes that are 10+ years old can rapidly become irrelevant as new and better models are proposed, encryption schemes become outdated and insecure. This parameter is introduced in an effort to stay relevant to the present-day security landscape.
- The literature *must* present a new model or scheme. There are plenty of literature pieces that document vital analysis of existing solutions, but for the piece to be relevant to this literature review and analysis the piece must present a new model, at the very least.
- The model *must* be some abstraction of a password manager or similar context. The goal of this review and the subsequent analysis is not to process broad data security topics.

A.2 Snowballing Process

In the following sections the different snowballing iterations and their findings are presented. Literature pieces that are eligible for review – and which have been added to the final set for analysis – are denoted in squire brackets ([ ]), where the initial starting set is denoted with letters, e.g. [A].

Subsequent works that have been approved for the final set are denoted with the letter from which this literature piece was found followed by the number of the reference in this paper (for backwards snowballing), e.g. [A.1] for the first reference in the [A] paper. If the literature piece was found using citations (i.e. forward snowballing) the paper is marked with a single quotation, e.g [A.1'] for the first citation found for the [A] paper. These citations were found using the citation engines integrated into the search engines.

When documenting a set of references or citations that were processed in either backwards or forwards snowballing procedure, the references and citations will

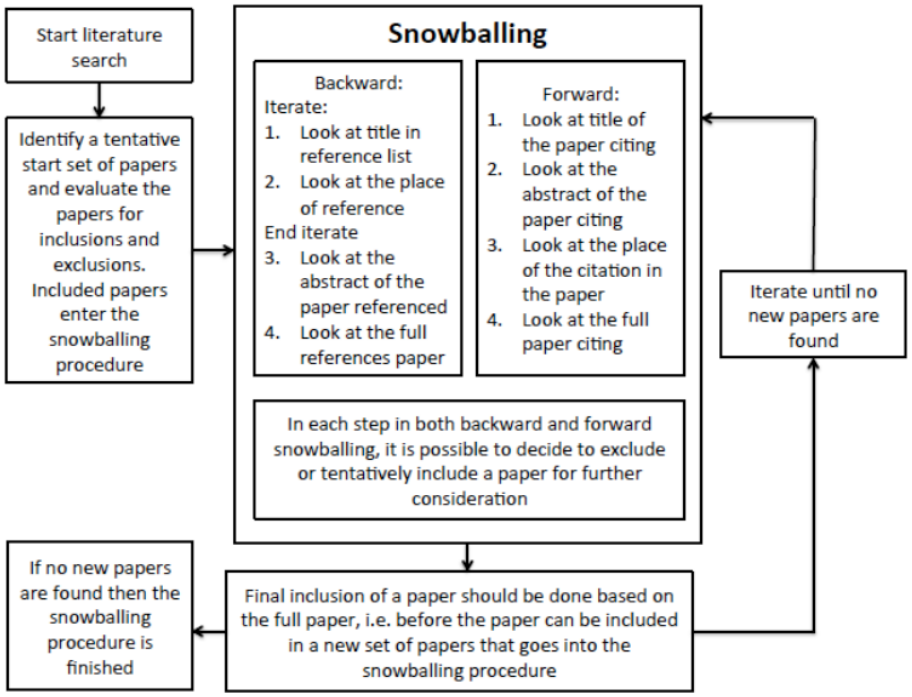

Figure A.1: Snowballing procedure. Source: [Woh14]

be listed as comma-separated values after their denoting literature piece, e.g. *[A] 1, 2, 3* for the first three references in the [A] paaper.

A.2.1 Iteration 1

The initial set of works found through the search queries consisted of three articles: [**A**] SplitPass ([Liu+18]), [**B**] DFF-PM ([Agh+16]), and [**C**] BluePass ([LWS19]).

Backwards Snowballing

The tentative set of references for the first iteration were as follows: [**A**] 2, 3, 4, 5, 8, 10, 12, 13, 14, 16, 20, 31, 33, 34; [**B**] 5, 6, 7; [**C**] 8, 10, 33, 34, 39.

It is worth noting that both [**B**] and [**C**] has plenty of interesting references

on literature covering many security topics, and in particular [**C**] covers many different works on 2FA models; yet these are out-of-scope for this review.

The final set of included references was narrowed down to: [**A**] 4; [**B**] None; [**C**] 8.

Forwards Snowballing

For this iteration the forward snowballing did not add any tentative nor final additions to the set of literature being processed. The few citations that were found were either out-of-scope for this review or they were analysis pieces that did not propose any new security models.

Result

Following the first iteration the complete set of literature was expanded with the pieces: [**A.4**] Tapas ([McC+12]); [**C.8**] NoCrack ([Cha+15]).

A.2.2 Iteration 2

The start set of the second iteration was the result of the first: [**A.4**] and [**C.8**].

Backwards Snowballing

The set of references added to the tentative pool for this iteration was as follows: [**A.4**] 3; [**C.8**] 9, 18, 23.

For the final set works that would be included in the next iteration, none of the [**C.8**] references were kept due to being prior to 2010 or referencing analysis pieces rather than security models. For [**A.4**] the one reference was kept.

Forwards Snowballing

This iteration included many more citations than the first and also resulted in a much larger tentative set: [**A.4**] 6', 7', 9', 10'; [**C.8**] None.

The [**C.8**] citations shared many with the [**A.4**] list and thus did not expand the tentative set.

The final set was reduced to [**A.4**] 6', 9'. 7' was excluded due to its scope on multi-user support and administrative features, while 10' was excluded for being too similar in work to that of the *Tapas* literature.

Result

At the end of this iteration the final literature set was expanded with [**A.4.3**] Kamouflage ([Boj+10]), [**A.4.6'**] SPHINX ([Shi+17]), [**A.4.9'**] Amnesia ([WLS16]).

A.2.3 Iteration 3

The start set of the third iteration was the result of the second: [**A.4.3**], [**A.4.6'**], and [**A.4.9'**].

Backwards Snowballing

This step in the third iteration only resulted in one tentative reference being added, specifically [**A.4.6'**] 30. [**A.4.3**] was published in 2010 and thus any references in this litearture would too old for consideration in this review, and [**A.4.9'**] simply did not add any tentative references either.

The final set was not expanded with any new literature in this step.

Forwards Snowballing

The tentative set was expanded with two new citations in this step: [**A.4.3**] 9' and [**A.4.9'**] 7'. [**A.4.6'**] had no contributions in this tep.

The final set, however, did not end up including either of the tentative pieces. While [**A.4.3**] 9' presents a fascinating model of a client-sever password manager model that does not rely on storing the passwords anywhere, the lack of such storage makes in incompatible with a majority of this thesis scope and requirements (for analysis and comparison). Thus it was deemed out of scope, much like secret vaults that rely on biometric data has not been included in the

scope of this review. [**A.4.9'**] 7' proposed an interesting model for using visual cryptography for encoding passwords and ID for use in client-server authentication, but the model – as proposed – is not directly suitable for a password manager.

Result

Due to the final set not being expand with any new literature pieces from either backwards or forwards snowballing, this iteration concludes the review process.

A.3 Findings

Following the iterations of the snowballing model the complete literature set was produced by combining the initial start set and all the final literature pieces found in each iteration. This resulted in the set shown below.

- SplitPass ([Liu+18])
- DFF-PM ([Agh+16])
- BluePass ([LWS19])
- Tapas ([McC+12])
- NoCrack ([Cha+15])
- Kamouflage ([Boj+10])
- SPHINX ([Shi+17])
- Amnesia ([WLS16])

Appendix B

Use Case Specifications

This appendix contains the detailed use case specifications modeling the requirements and features of the distributed password manager system, as covered in section 3.5.1.

It is worth noting that a use case for modifying/editing a vault entry is not included in this model since it is semantically equivalent to a deletion and insertion operation pair.

On the point of terminology, the use of *temporary storage* in the use cases refer to in-memory storage, the lifetime of which is (at most) the lifetime of the running application after it has been started; while *network* refer to the network of nodes collaborating to implement the given password manager instance.

Use case: StartApplication
ID: UC1
Description: A user wants to start the application.
Primary actors: User
Secondary actors: None
Preconditions: 1. The application is available on the device.
Main flow: 1. The user starts the application. 2. The system checks if the device has been configured. 3. If the device has not been configured 3.1 Goto UC2 4. Else 4.1 The system loads the configuration, incl. network properties. 4.2 While user's master password (MP) is not valid 4.2.1 The system gets MP from the user. 4.2.2 The system validates MP using the configuration. 4.3 The system temporarily stores a derivation of MP. 5. The system starts listening for requests from other nodes.
Postconditions: 1. The application has been started. 2. A derivation of MP is stored temporarily. 3. The system is ready to respond to requests from other nodes.
Alternative flows: None

Use case: ConfigureDevice
ID: UC2
Description: The user wants to configure the application on their device
Primary actors: User
Secondary actors: None
Preconditions: 1. The application has been started (UC1)
Main flow: 1. The user inputs their desired master password (MP). 2. The system stores a temporary derivative of MP. 3. The system prompts the user if this is the first device configured in the network. 4. If device is first in network 4.1 The system generates a network ID seed 5. Else 5.1 The system gets the network ID seed from the user 6. The system configures its network properties using MP and network ID seed. 7. The system saves the network properties on the device. 8. The system requests vault fragments from the network 9. If the system received vault fragments 9.1 The system builds its vault from the fragments and temporarily stores it. 10. Else 10.1 The system builds an empty vault and temporarily stores it. 11. Goto UC9. 12. The system deletes its temporarily stored vault.
Postconditions: 1. A derivative of MP is stored temporarily. 2. The network properties are stored on the device. 3. A fragment of the vault is stored on the device.
Alternative flows: None

Use case: SignIn
ID: UC3
Description: The user wants to sign in on a configured device.
Primary actors: User
Secondary actors: Node network
Preconditions: 1. The application has been started (UC1) and configured on the device (UC2).
Main flow: 1. The user inputs their MP. 2. While validation of MP fails 2.1 The system validates MP locally against its derivative of MP. 2.2 IF validation fails 2.2.1 The system displays an error. 2.3 The system validates MP remotely in the network. 2.4 IF validation fails 2.4.1 The system displays an error. 3. The system requests vault fragments from the network. 4. The system loads and decrypts local vault fragment. 5. The system constructs its vault from the fragments. 6. The system sets the user as validated.
Postconditions: 1. The user's MP has been validated locally. 2. The user's MP has been validated remotely by the network. 3. The secure vault is constructed temporarily. 4. The user has been validated.
Alternative flows: None

Use case: InsertEntry
ID: UC4
Description: The user wants to inserts an entry into the vault.
Primary actors: User
Secondary actors: Node network
Preconditions: 1. The user is signed in (UC3).
Main flow: 1. The user selects option to add an entry in the password manager. 2. The user inputs the entry data (identifying name and secret password). 3. The system saves the entry to the temporary vault. 4. Goto UC9.
Postconditions: 1. The new entry is in the temporary vault.
Alternative flows: None

Use case: FindEntry
ID: UC5
Description: The user wants to search for an entry in the vault.
Primary actors: User
Secondary actors: None
Preconditions: 1. The user is signed in (UC3).
Main flow: 1. The user selects option to search for an entry in the password manager. 2. The user inputs a search query on the entry's name. 3. The system searches its temporary vault for matches to the query 4. The system presents the result to the user.
Postconditions: 1. The user has been presented with the result set of the search query.
Alternative flows: None

Use case: DeleteEntry
ID: UC6
Description: The user wants to delete an entry from the password manager's vault.
Primary actors: User
Secondary actors: Node network
Preconditions: 1. The user is signed in (UC3) and has located a password entry (UC5).
Main flow: 1. The user selects option to delete the entry from the vault. 2. The system removes the entry from its temporary vault. 3. Goto UC9.
Postconditions: 1. The deleted entry is no longer in the temporary vault.
Alternative flows: None

Use case: SignOut
ID: UC7
Description: The user wants to sign out of the application.
Primary actors: User
Secondary actors: None
Preconditions: 1. The user is signed in (UC3).
Main flow: 1. The user selects option to sign out. 2. The system deletes its temporary vault. 3. The system invalidates the user
Postconditions: 1. The temporary vault has been deleted. 2. The user has been invalidated.
Alternative flows: None

Use case: GetFragment
ID: UC8
Description: A remote node in the network wants to get the local node's vault fragment.
Primary actors: Network node
Secondary actors: None
Preconditions: 1. The application has been started (UC1) and is configured on the device (UC2).
Main flow: 1. The network node sends a request get the system's local vault fragment. 2. The system opens a connection to the node. 3. The system authenticates the node. 4. The system secures the connection. 5. The system loads and decrypts its local vault fragment. 6. The system transmits its vault fragment over the secured connection. 7. The system closes the connection.
Postconditions: 1. The network node has been authenticated. 2. The system's local vault fragment has been sent to the network node.
Alternative flows: FailedNodeAuthentication 1. The alternative flow begins after step 3 of the main flow. 2. If node authentication fails 2.1 The system closes the connection. 2.2 The use case does not return to main flow.

Use case: NotifyNetwork
ID: UC9
Description: The system wants to notify the node network of changes to the vault.
Primary actors: The system
Secondary actors: None
Preconditions: 1. The system has stored a temporary vault.
Main flow: 1. The system fragments the temporary vault into a number of fragments matching the number of network nodes. 2. The system encrypts one fragment and saves it to the device. 3. If there are more than one fragment 3.1 For each node in the network 3.1.1 The system connects the the node. 3.1.2 The system authenticates the node. 3.1.3 The system secures the connection. 3.1.4 The system transmits one of the remaining fragments
Postconditions: 1. The system has stored a new local vault fragment. 2. Each node in the network has received a new vault fragment.
Alternative flows: FailedNodeAuthentication 1. The alternative flow begins after step 3.1.2 of the main flow. 2. If node authentication fails 2.1 The system signals an error occurred. 2.2 The system closes the connection. 2.3 The use case returns to main flow 3.1.

Appendix C

Activity Diagrams

This appendix contains the activity diagrams created for the use case realization from section 3.5.3. The diagrams describe the system's internal interactions required to satisfy the use cases and model its required behavior.

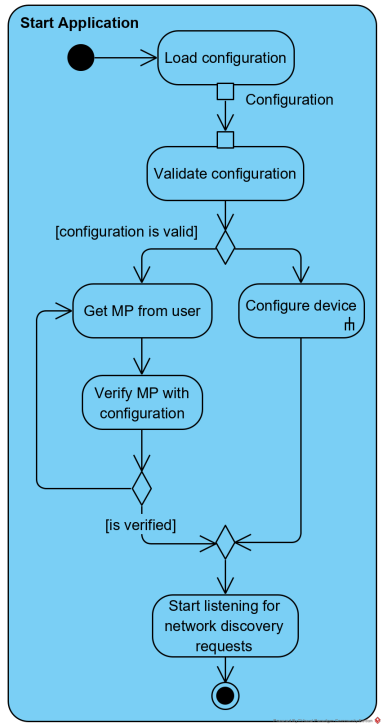

Figure C.1: Start Application activity diagram.

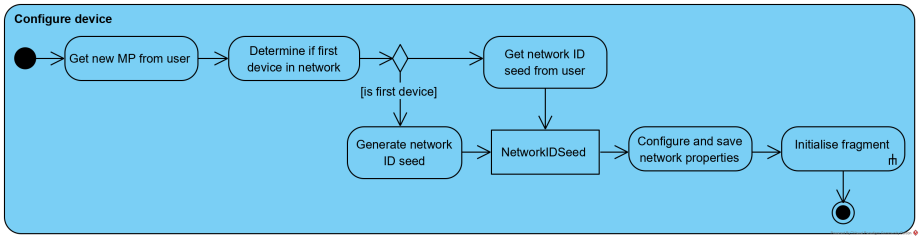

Figure C.2: Configure Device activity diagram.

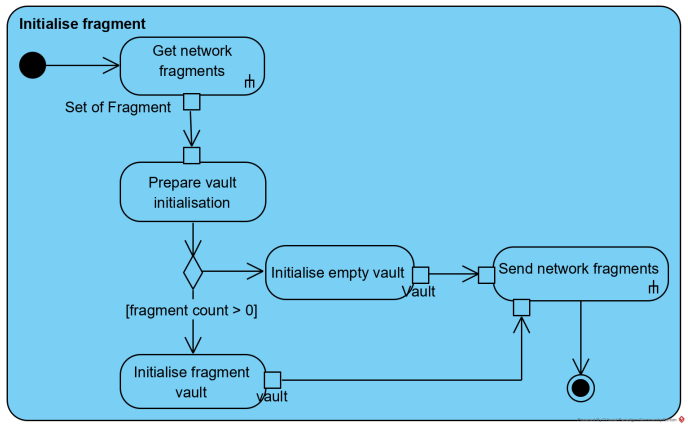

Figure C.3: Initialize Fragment activity diagram.

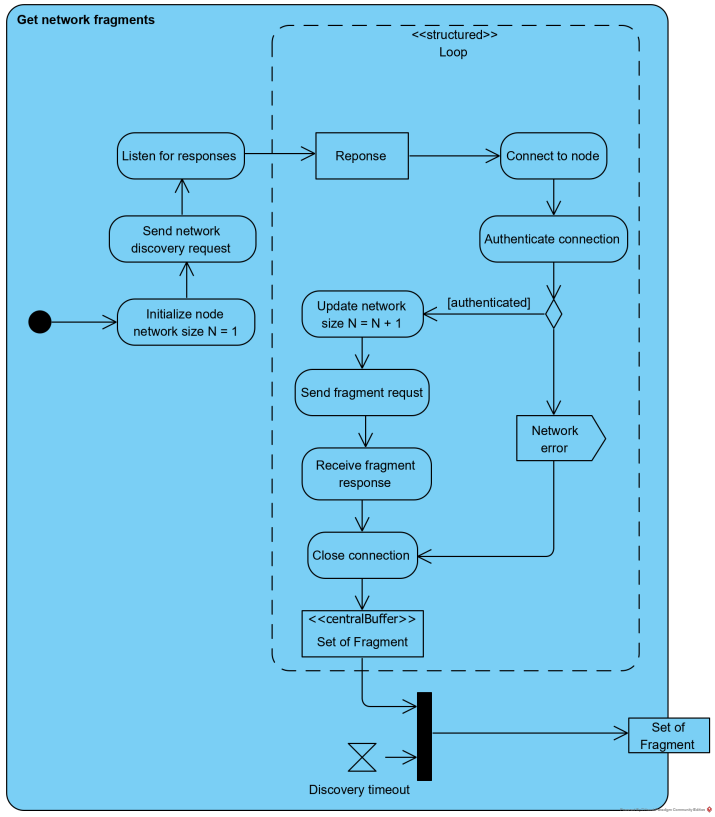

Figure C.4: Get Network Fragments activity diagram.

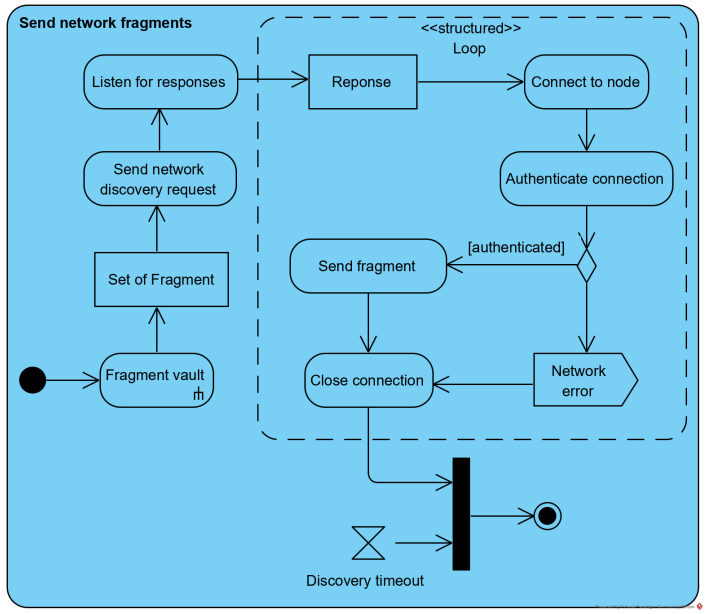

Figure C.5: Send Network Fragments activity diagram.

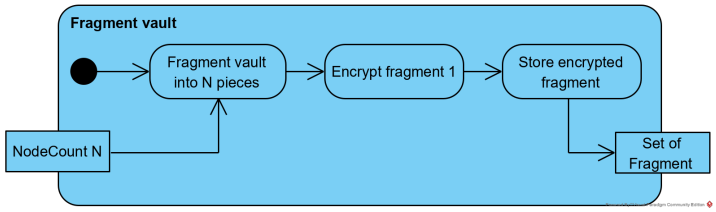

Figure C.6: Fragment Vault activity diagram.

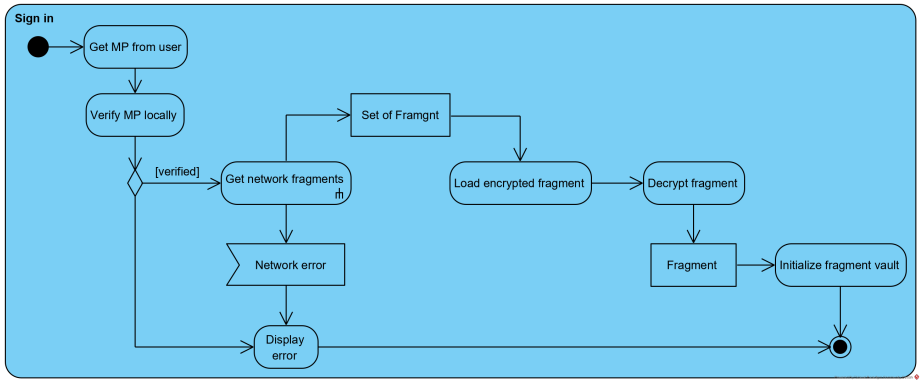

Figure C.7: Sign In activity diagram.

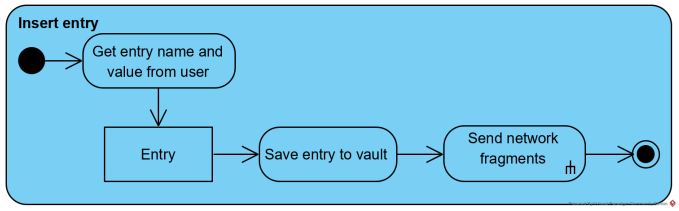

Figure C.8: Insert Entry activity diagram.

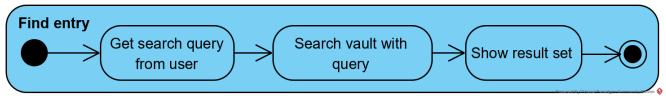

Figure C.9: Find Entry activity diagram.

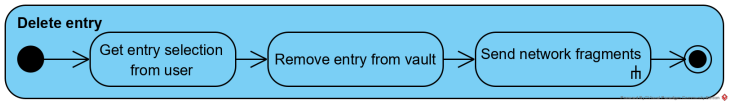

Figure C.10: Delete Entry activity diagram.

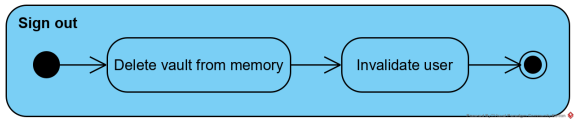

Figure C.11: Sign Out activity diagram.

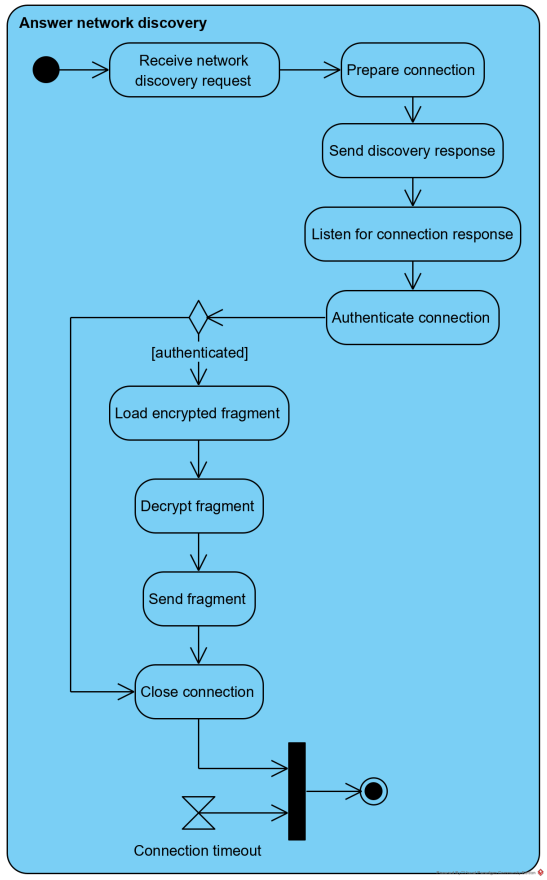

Figure C.12: Answer Network Discovery activity diagram.

Appendix D

Class Diagrams

This appendix contains the implementation class diagrams for the DPM system, as presented in section 4.3.3.

The diagrams do not present a complete, detailed model of the implementation components, but instead focus on the relationships between the components in each of the base packages in the prototype. Relationships that span the base packages are not shown.

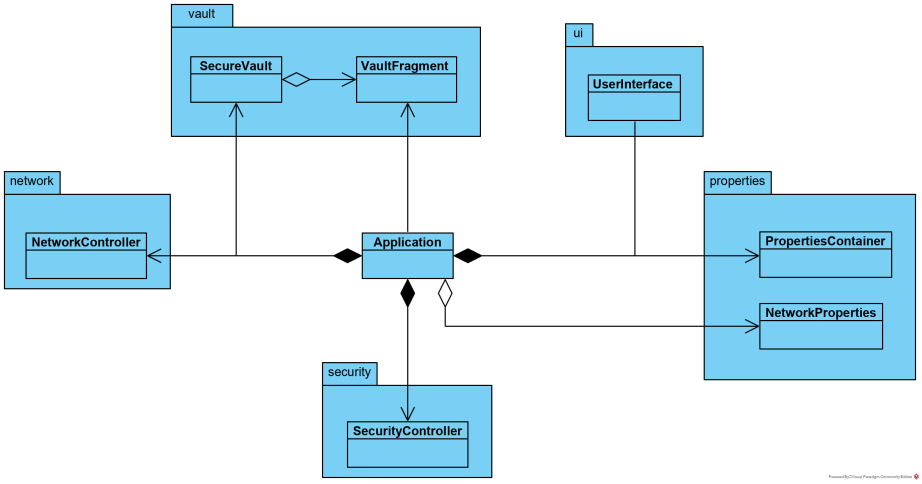

Figure D.1: Base *dpm* package class diagram.

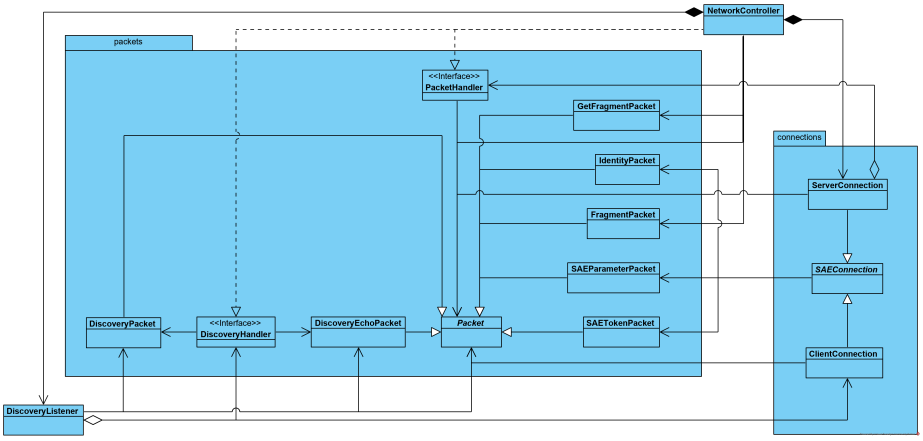

Figure D.2: *network* package class diagram.

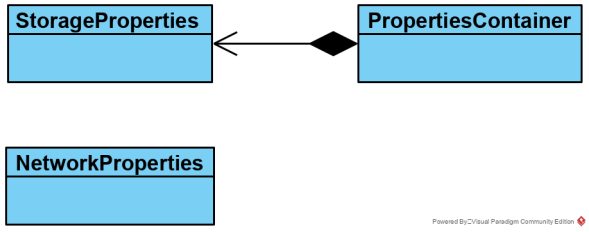

Figure D.3: *properties* package class diagram.

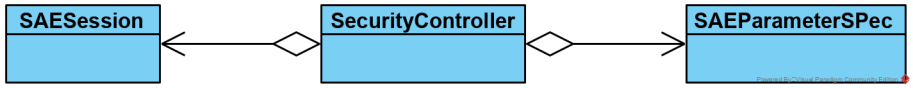

Figure D.4: *security* package class diagram.

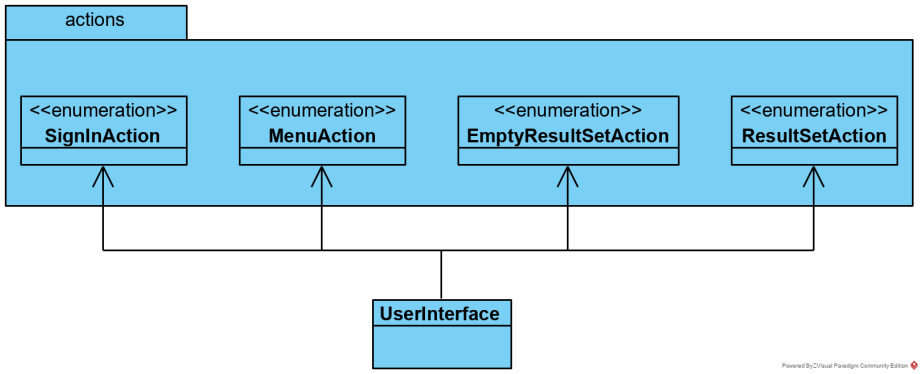

Figure D.5: *ui* package class diagram.

Figure D.6: *util* package class diagram.

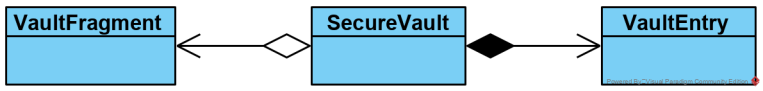

Figure D.7: *vault* package class diagram.

Appendix E

Acceptance Tests

This appendix contains the detailed documentation of this project's acceptance tests from section 5.1.2, carried out in order to validate the solution's ability to satisfy the project's use cases.

Each use case has one or more test executions which are described in the following sections. Like the use cases, they are segmented into preconditions that must be satisfied prior to the test execution, the flow of the test, and the postconditions that validate the use cases postconditions and must hold true for the test to have passed.

It should be noted that the step-numbering within the test flow does not align with the numbering within the main flow of the use case under test; instead they simply define the order of the test execution; except for in the postconditions where the numbering does align.

The use of the term *agent* in the test cases refers to the user performing the test.

E.1 Start Application

Description: Testing UC1 - starting the application with configuration already done.
Status: Passed
Preconditions:

1. The device has already been configured

Test flow:

1. The application is started by executing the dpm.jar file.
2. The agent inputs an incorrect password to test validation.
3. The agent inputs a correct master password.

Postconditions:

1. The application has been started.
2. *The condition cannot be verified by an external agent as it has no side effects.*
3. The log shows the application is listening for requests.

E.2 Configure Device

E.2.1 First Device

Description: Testing UC2 - configuring the first device in the network.
Status: Passed
Preconditions:

1. No other device has been configured.
2. The application is started.

Test flow:

1. The agent inputs the desired master password.
2. The agent inputs the master password again.
3. The agent selects that the device *is* the first in the network.
4. The agent is displayed the generated network seed and writes it down.

Postconditions:

1. *The condition cannot be verified by an external agent as it has no side effects.*
2. The agent verifies that the file data/network.prop has been created in the application's directory.
3. The agent verifies that the file data/vault.frag has been created in the application's directory.

E.2.2 Other Device

Description: Testing UC2 - configuring another device in the network.
Status: Passed
Preconditions:

1. Another device has been configured.
2. The application is started.

Test flow:

1. The agent inputs the desired master password.
2. The agent inputs the master password again.
3. The agent selects that the device *is not* the first in the network.
4. The agent inputs the network seed.

Postconditions:

1. *The condition cannot be verified by an external agent as it has no side effects.*
2. The agent verifies that the file data/network.prop has been created in the application's directory.
3. The agent verifies that the file data/vault.frag has been created in the application's directory.s.

E.3 Sign In

Description: Testing UC3 - signing in.
Status: Passed
Preconditions:

1. The local device has been configured.
2. Another device has been configured.
3. The application is started on the local.

Test flow:

1. The agent selects the option to sign in.
2. The agent inputs an incorrect password to test local validation.
3. The agent inputs the correct master password to test negative remote validation.
4. The agent starts the application on the other device.
5. The agent inputs the correct master password to test positive remote validation.

Postconditions:

1. The agent implicitly verify this by being signed in for the correct password, while being denied on the incorrect password.

2. The agent implicitly verify this by being denied when signing in without all devices being online, and is signed in when all are such.
3. *The condition cannot be verified by an external agent as it has no side effects.*
4. The agent observes that they have been signed in.

E.4 Insert Entry

Description: Testing UC4 - inserting a new vault entry.
Status: Passed
Preconditions:

1. The agent is signed in.

Test flow:

1. The agent selects the option to add a new entry.
2. The agent inputs the desired entry name.
3. The agent inputs the desired entry password.

Postconditions:

1. The agent can verify the entry by selecting the option to view the entire vault, and verify that the new entry is shown.

E.5 Find Entry

Description: Testing UC5 - searching for an entry.
Status: Passed
Preconditions:

1. The agent is signed in.

Test flow:

1. The agent selects the option to search for an entry.
2. The agent inputs the desired search query.

Postconditions:

1. The agent observes that they are displayed the result of their query.

E.6 Delete Entry

Description: Testing UC6 - deleting a vault entry.
Status: Passed
Preconditions:

1. The agent is signed in.
2. The agent has been presented with the vault or the result of a search query.

Test flow:

1. The agent selects the option to delete an entry.
2. The agent selects the entry to delete.

Postconditions:

1. The agent verify the deletion by trying to search for the entry and will observe that no such entry is found.

E.7 Sign Out

Description: Testing UC7 - signing out
Status: Passed
Preconditions:

1. The agent is signed in.

Test flow:

1. The agent selects the menu option to sign out.

Postconditions:

1. *The condition cannot be verified by an external agent as it has no side effects.*
2. The agent observes that they have been signed out and will have to sign in again to access the vault.

E.8 Get Fragment

Description: Testing UC8 - getting the fragment from another device.
Status: Passed
Preconditions:

1. The application is started on the local device, and has been configured.
2. The application is started on another device, and has been configured.

Test flow:

1. The agent sign in on the local device.
2. The agent inspects the log of the local device.

Postconditions:

1. The agent observes the log printing authentication steps for the other node connection.
2. The agent observes the log printing the reception of the other node's fragment.

E.9 Notify Network

Description: Testing UC9 - notifying the node network of vault changes.
Status: Passed
Preconditions:

1. The application is started on the local device, and has been configured.
2. The application is started on another device, and has been configured.
3. The agent is signed in on the local device.

Test flow:

1. The agent selects menu option to add a new entry to the vault.
2. The agent inputs the desired entry name.
3. The agent inputs the desired entry password.
4. The agent inspects the log on the local device.
5. The agent inspects the log on the other device.

Postconditions:

1. The agent observes the log printing that a new local fragment has been saved.
2. The agent observes the log printing that it has connected to the other node and send its request.
3. The agent observes the log on the other device printing it has saved a new fragment following the local device's request.

Bibliography

[Adr+15] David Adrian et al. “Imperfect forward secrecy: How diffie-hellman fails in practice”. eng. In: *Proceedings of the Acm Conference on Computer and Communications Security* 2015- (2015), pp. 5–17. ISSN: 15437221. DOI: 10.1145/2810103.2813707.

[Agh+16] S. Agholor et al. “A secured Mobile-Based Password Manager”. eng. In: *2016 6th International Conference on Digital Information Processing and Communications, Icdipc 2016* (2016), pp. 7470800, 103–108. DOI: 10.1109/ICDIPC.2016.7470800.

[AN13] J. Arlow and I. Neustadt. *UML 2 and the Unified Process, Second Edition*. Addison-Wesley, 2013. ISBN: 0-321-32127-8. URL: http://www.umlandtheunifiedprocess.com.

[Ber] Lange T. Bernstein D. J. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. Last checked 2019-12-18. URL: https://safecurves.cr.yp.to.

[Boj+10] Hristo Bojinov et al. “Kamouflage: Loss-resistant password management”. eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6345 (2010), pp. 286–302. ISSN: 16113349, 03029743. DOI: 10.1007/978-3-642-15497-3_18.

[Cha+15] Rahul Chatterjee et al. “Cracking-resistant password vaults using natural language encoders”. eng. In: *Proceedings - Ieee Symposium on Security and Privacy* 2015- (2015), pp. 7163043, 481–498. ISSN: 23751207, 10816011. DOI: 10.1109/SP.2015.36.

[Das+14] Anupam Das et al. “The Tangled Web of Password Reuse”. eng. In: (2014).

[Das18] Dashline. “Security White Paper”. May 2018. URL: https://assets.cdngetgo.com/1d/ee/d051d8f743b08f83ee8f3449c15d/lastpass-technical-whitepaper.pdf.

[Dra18] Nicola Dragoni. “Distributed Systems course lectures, Technical University of Denmark”. 2018.

[Eff09] Standards for Efficient Cryptography. “SEC1: Elliptic Curve Cryptography version 2.0”. May 2009. URL: https://www.secg.org/sec1-v2.pdf.

[ENI14] ENISA. “Algorithms, key size and parameters report – 2014”. eng. In: (2014). DOI: 10.2824/36822.

[Fuk+16] Masayuki Fukumitsu et al. “A proposal of a password manager satisfying security and usability by using the secret sharing and a personal server”. eng. In: *Proceedings - International Conference on Advanced Information Networking and Applications, Aina* 2016- (2016), pp. 7474152, 661–668. ISSN: 23325658, 19767684, 1550445x. DOI: 10.1109/AINA.2016.45.

[Har08] Dan Harkins. “Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks”. eng. In: *Proceedings - 2nd Int. Conf. Sensor Technol. Appl., Sensorcomm 2008, Includes: Mesh 2008 Conf. Mesh Networks; Enopt 2008 Energy Optim. Wireless Sensors Networks, Unwat 2008 Under Water Sensors Systems* (2008), pp. 4622764, 839–844. DOI: 10.1109/SENSORCOMM.2008.131.

[HV12] Cormac Herley and Paul Van Oorschot. “A research agenda acknowledging the persistence of passwords”. eng. In: *Ieee Security and Privacy* 10.1 (2012), pp. 6035662, 28–36. ISSN: 15584046, 15407993. DOI: 10.1109/MSP.2011.150.

[Jar+16] Stanislaw Jarecki et al. “Device-enhanced password protocols with optimal online-offline protection”. eng. In: *Asia Ccs 2016 - Proceedings of the 11th Acm Asia Conference on Computer and Communications Security* (2016), pp. 177–188. DOI: 10.1145/2897845.2897880.

[JR14] Ari Juels and Thomas Ristenpart. “Honey encryption: Encryption beyond the brute-force barrier”. eng. In: *Ieee Security and Privacy* 12.4 (2014), pp. 6876246, 59–62. ISSN: 15584046, 15407993. DOI: 10.1109/MSP.2014.67.

[Las] LastPass. “Technical Whitepaper”. URL: https://assets.cdngetgo.com/1d/ee/d051d8f743b08f83ee8f3449c15d/lastpass-technical-whitepaper.pdf.

[Li+14] Xiaolei Li et al. “DroidVault: A trusted data vault for android devices”. eng. In: *Proceedings of the Ieee International Conference on Engineering of Complex Computer Systems, Iceccs* (2014), pp. 6923115, 29–38. DOI: 10.1109/ICECCS.2014.13.

[Li+15] Zhiwei Li et al. “The emperor’s new password manager: Security analysis of web-based password managers”. eng. In: (2015). DOI: 10.1.1.694.8989.

[Liu+18] Yu Tao Liu et al. “SplitPass: A Mutually Distrusting Two-Party Password Manager”. eng. In: *Journal of Computer Science and Technology* 33.1 (2018), pp. 98–115. ISSN: 18604749, 10009000. DOI: 10.1007/s11390-018-1810-y.

[LWS19] Yue Li, Haining Wang, and Kun Sun. “BluePass: A Mobile Device Assisted Password Manager”. eng. In: *Icst Transactions on Security and Safety* 5.17 (2019), p. 156244. ISSN: 20329393. DOI: 10.4108/eai.10-1-2019.156244.

[McC+12] Daniel McCarney et al. “Tapas: Design, implementation, and usability evaluation of a password manager”. eng. In: *Acm International Conference Proceeding Series* (2012), pp. 89–98. DOI: 10.1145/2420950.2420964.

[NIS07] NIST. “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC”. eng. In: *SP 800-38D* (2007). DOI: 10.6028/NIST.SP.800-38D.

[NIS15a] NIST. “Recommendation for Random Number Generation Using Deterministic Random Bit Generators”. eng. In: *SP 800-90A Rev. 1* (2015). DOI: 10.6028/NIST.SP.800-90Ar1.

[NIS15b] NIST. “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”. eng. In: *FIPS PUB 202* (2015). DOI: 10.6028/NIST.FIPS.202.

[NIS16] NIST. “Recommendation for Key Management Part 1: General”. eng. In: *SP 800-57 Part 1 Revision 4* (2016). DOI: 10.6028/NIST.SP.800-57pt1r4.

[Reia] Dominik Reichl. *KeePaas Security*. Last checked 2019-08-20. URL: https://keepass.info/help/base/security.html.

[Reib] Dominik Reichl. *KeePass Features*. Last checked 2019-08-20. URL: https://keepass.info/features.html.

[Shi+17] Maliheh Shirvanian et al. “SPHINX: A Password Store that Perfectly Hides Passwords from Itself”. eng. In: *Proceedings - International Conference on Distributed Computing Systems* (2017), pp. 7980050, 1094–1104. ISSN: 10636927. DOI: 10.1109/ICDCS.2017.64.

[Sil+14] David Silver et al. “Password Managers: Attacks and Defenses”. eng. In: (2014). DOI: 10.1.1.432.8172.

[Sol] 8bit Solutions. *Security | Bitwarden Help*. Last checked 2019-08-19. URL: https://help.bitwarden.com/security/.

[Wei+09] Matt Weir et al. “Password cracking using probabilistic context-free grammars”. eng. In: *Proceedings - Ieee Symposium on Security and Privacy* (2009), pp. 5207658, 391–405. ISSN: 23751207, 10816011. DOI: 10.1109/SP.2009.8.

[WLS16] Luren Wang, Yue Li, and Kun Sun. “Amnesia: A Bilateral Generative Password Manager”. eng. In: *Proceedings - International Conference on Distributed Computing Systems* 2016- (2016), pp. 7536530, 313–322. ISSN: 10636927. DOI: 10.1109/ICDCS.2016.90.

[Woh14] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. eng. In: *Acm International Conference Proceeding Series* (2014), a38. DOI: 10.1145/2601248.2601268.

[YDC17] Dana Yang, Inshil Doh, and Kijoon Chae. “Enhanced password processing scheme based on visual cryptography and OCR”. eng. In: *International Conference on Information Networking* (2017), pp. 7899514, 254–258. ISSN: 23325658, 19767684. DOI: 10.1109/ICOIN.2017.7899514.

[ZY13] Rui Zhao and Chuan Yue. “All your browser-saved passwords could belong to us: A security analysis and a cloud-based new design”. eng. In: *Codaspy 2013 - Proceedings of the 3rd Acm Conference on Data and Application Security and Privacy* (2013), pp. 333–340. DOI: 10.1145/2435349.2435397.