Bachelor's Thesis Computing Science

Integrating PostGuard into Filesender

Bryan Rinders
s1060340

August 18, 2024

First supervisor/assessor:
Dr. Bram Westerbaan

Second assessor:
Prof. Dr. Bart Jacobs

Radboud University

Abstract

Filesender is a web application with which large files can be uploaded and shared with a select group of people. Optionally, files can be encrypted, however the usability of this feature leaves a lot to be desired. Most notably, uploaders are expected to manually distribute the encryption password to the recipients and are therefore responsible for doing so in a secure manner.

In this thesis we explore whether PostGuard is a suitable candidate to alleviate this issue. We do this by developing a prototype that integrates PostGuard into Filesender. PostGuard is an encryption service that is privacy- and user-friendly. It uses a Trusted Third Party (TTP) to manage the cryptographic keys, and Yivi—an identity management platform, which includes an identity wallet—for authentication to the TTP. Ultimately, we conclude that it is technically feasible to integrate PostGuard into Filesender; there are however some considerations to keep in mind when working out this prototype and running it in production. We provide recommendations for future work and for working out a full implementation of the prototype.

Keywords: Filesender, PostGuard, file encryption

Contents

1 Introduction **2**

2 Preliminaries **4**
- 2.1 Filesender . . . 4
  - 2.1.1 Authentication . . . 4
  - 2.1.2 Privacy . . . 5
  - 2.1.3 Walkthrough . . . 6
- 2.2 Yivi . . . 6
  - 2.2.1 Idemix . . . 8
  - 2.2.2 Security properties . . . 8
  - 2.2.3 Discussion . . . 9
- 2.3 Identity-based encryption . . . 10
- 2.4 PostGuard . . . 11
  - 2.4.1 IBKEM . . . 12
  - 2.4.2 PostGuard and JavaScript . . . 13

3 Development of the prototype **15**
- 3.1 PostGuard encryption . . . 15
- 3.2 The new user experience . . . 16
  - 3.2.1 User Interface . . . 17
  - 3.2.2 Walk-through of the new user experience . . . 17
- 3.3 Webpack . . . 19
  - 3.3.1 History . . . 22
  - 3.3.2 Bundlers . . . 22
  - 3.3.3 Content Security Policy . . . 23

4 Discussion **26**
- 4.1 Evaluation of the prototype . . . 27
- 4.2 Limitations . . . 30
- 4.3 Future work . . . 30

5 Conclusion **32**

Chapter 1

Introduction

FileSender (FS) is an open-source web application for sending large files to a select group of people [1]. It is similar in functionality to WeTransfer [2]. When uploading files, an uploader can specify recipients by giving a list of email addresses or manually distribute a download link. The uploader can optionally have FS encrypt the files (client-side), using a password. Recipients can, after receiving a download link or an email from FS, download the files. If the files are encrypted FS will, before downloading, prompt for the password—which the recipient received from the uploader—and then download and decrypt the files (client-side).

Currently, FS leaves the sharing of the password as a manual exercise. To *securely* send this password to all recipients an external application is needed that can authenticate contacts and send end-to-end encrypted messages. An example of an application that has these features is Signal. However, it would be much nicer to have a feature built into FS that simplified this process.

This is where PostGuard (PG) [3] might come into play. PG is a project that aims to simplify the encryption process for end-users by removing the burden of manual key management. It does this through the use of a Trusted Third Party (TTP), who is in charge of key generation and distribution. To encrypt a message, the sender uses the TTP's Master Public Key and one or more attributes of the recipient. An attribute is a statement or property about someone, such as "I am over 18 years old". Decryption of the message requires the recipient to prove ownership of the attributes to the TTP. When successful, the TTP generates the *user secret key* that the recipient can use to decrypt the message. PG uses Yivi [4]—a privacy-friendly identity platform—for authentication i.e. proving ownership of one or more attributes.

Our research question is as follows:

"Is it feasible to integrate PostGuard into Filesender, and, if so, what are the steps necessary to implement this?"

The following sub-questions came up during our research:

1. What to encrypt using PostGuard?
The goal is to encrypt files. However, as we will discuss in more detail in section 3.1, PG can encrypt different kinds of messages such as emails or files. We can therefore use PG to either directly or indirectly encrypt files. To clarify with an example, we could add the FS encryption password to an email, encrypt the email using PG and send that to the recipient(s). Hence there are multiple ways to achieve this goal using PG. We will explore several of them and discuss the pros and cons.

2. How will Filesender change?
What has to change in the front- and back-end to integrate PG into FS?

In the preliminaries (chapter 2) we will cover the necessary background information on FS, PG and the components that make up PG. Then we will discuss all the important decisions that were made during the development of the prototype (chapter 3) and we will answer the sub-questions mentioned earlier. Sub-question 1 will be answered in section 3.1, and sub-question 2 in section 3.2. We will end with a discussion of our research and directions for future work (chapter 4), and finally a conclusion (chapter 5).

Chapter 2

Preliminaries

In this chapter we discuss FS (section 2.1), the components that make up PG—Yivi (section 2.2), and identity-based encryption (section 2.3)—and we end with PG itself (section 2.4).

2.1 Filesender

As explained in the introduction, FS is a web application for sending large files to one or more recipients. Recipients can be specified by listing their email addresses or by manually distributing a download link to them. The files are available for a limited time period, as specified by the uploader, and can optionally be encrypted (client-side) by FS. The subsequent decryption of the files will also happen on the client-side, so in more technical terms, FS offers end-to-end encryption (E2EE) for file transfers.

FS is open-source, allowing anyone to set up and run their own instance, which many organizations across the globe do [5]. FS is particularly popular in the research and education community, with prominent instances such as RedCLARA [6] in Latin America and GÉANT [7] in Europe. To improve accessibility and usability, FS supports many languages, including English, Spanish, French, Korean, and Chinese. The application is highly customizable, enabling the instance's system administrators to configure many things, from major aspects, like choosing the database software, to finer details, such as setting the maximum file size for uploads.

2.1.1 Authentication

FS generally requires uploaders to be logged in, offering built-in support for authentication through Shibboleth and SimpleSAMLphp libraries. This enables authentication via various protocols such as SAML2, LDAP, RADIUS,

Active Directory, and even social platforms like Facebook.

However, FS also accommodates guest users through access vouchers, which can be made available for [8]:

- single-use vouchers;
- unlimited-use vouchers;
- vouchers that only allow sending files to the voucher creator.

The type of voucher is determined by the creator, who must be authenticated with FS.

Downloading files does not require authentication; access is granted simply by having a download link, regardless of whether the link was distributed via FS email or manually by the uploader. If the files are encrypted, the encryption password is also required.

2.1.2 Privacy

FS is designed with user privacy in mind, protecting both uploaders and recipients [8]. It separates logging of per transfer audit trails—which has a configurable lifetime—and logging for statistical purposes. An audit report can automatically be sent to the uploader on transfer expiration, after which the audit logging can be deleted. This means the sensitive data is kept exactly as long as it is needed. There is also support for logging additional user account parameters in the stat-log, such as an organization identifier allowing statistics per customer. Moreover, FS can be configured to enforce the mandatory use of HTTPS, ensuring that all connections with the FS server are encrypted.

FS also allows file sharing with anonymous recipients, where the uploader manually distributes a download link. To prevent FS from logging the recipient's IP address, they can use the Tor browser. However, maintaining anonymity online involves steps outside of FS's control and is the responsibility of the recipient, so these measures won't be discussed further.

FS does *not* allow uploaders to be anonymous; they must always authenticate with FS. A guest uploader is also not entirely anonymous, as they are invited via email, meaning their guest account is linked to a specific email address.

Finally, FS offers the ability to share files confidentially through E2EE, ensuring that only the uploader and the intended recipient(s) can access the content.

As we will soon see, in section 2.4, PG is also a privacy oriented project. Thus, in this respect, PG and FS are a good match.

2.1.3 Walkthrough

An example usage of FS with file encryption—assuming that (i) recipients are specified by email address, (ii) FS handles password generation, and (iii) Signal is used to distribute the password—is as follows:

- The uploader (Alice), in her browser, on the FS webpage:
  1. selects the files to be uploaded;
  2. enables file encryption;
  3. lets FS generate a random password;
  4. specifies the recipients by email address;
  5. clicks the "Send" button;
  6. copies the encryption password and uses Signal to send the it to each of the recipients.

  After step 5 the files are encrypted (client-side) and send to FS's server.

- A recipient (Bob):
  1. retrieves the encryption password from Signal sent by Alice;
  2. opens the email from FS and follows the download link;
  3. clicks the "Download" button on the FS download page;
  4. enters the password from step 1;
  5. clicks "OK" to confirm the password.

  FS will then download and decrypt (client-side) the files, in Bob's browser, and place them in his download directory. To check the correctness of the password entered in step 4, FS will download a part (`5 MiB` blob) of the file—or the entire file if the file is smaller than `5 MiB`—and tries to decrypt it. If the blob is decrypted successfully, FS will continue downloading and decrypting `5 MiB` blobs until the entire decrypted file is on the recipients device. If the decryption fails FS will stop the download and decryption process.

2.2 Yivi

Yivi [4] is one of the components used by PG. It was formerly known as IRMA (an acronym for I Reveal My Attributes)—it is still referred to as IRMA in some places—and is an identity wallet based on IBM's idemix, an attribute-based credential scheme [9], [10].

The basic idea behind a digital identity wallet is that users can store and selectively disclose their attributes, with consent. As mentioned in the introduction, an attribute is a statement or property about someone such as “I am over 18 years old” or “My name is Alice Alison”. For example, Alice, who goes to the Radboud University has an attribute student. If Alice wants to order something with student discount from a web-shop, she needs to prove she is a student. This situation is exactly what Yivi was built for. In a traditional authentication system Alice has to login using her university email address or another method proving she goes to a specific university. These methods will leak Alice’s place or field of study to the web-shop. There is also no way for Alice to find out what other information is requested by the web-shop from the university. Using Yivi however, Alice is able to prove she is a student without leaking her place of study or any other information. With Yivi Alice can also see exactly which attributes the web-shop is requesting and can decide whether or not to disclose these attributes. Furthermore, the disclosing of attributes is done without the need for a third party. Alice’s attributes are stored locally, in her identity wallet, and can be disclosed from there. This is explained further in section 2.2.2 (item: No privacy hotspots) and depicted in fig. 2.1.

But before Alice can start using Yivi she must first obtain attributes, which starts by downloading the Yivi mobile app—available for iOS and Android—and registering in the app. Then, to obtain attributes Alice must authenticate to an issuer, which is a party that issues attributes. An example of an issuer is the municipality of Nijmegen, it issues attributes such as name, address and age. Continuing our web-shop example, for Alice to obtain the student attribute, she has to:

- open the Yivi mobile app;
- click “retrieve my personal data”;
- choose “Education and research” by the Privacy by Design Foundation;
- click “add”, to confirm the attributes that will be retrieved;
- authenticate with her research institution;
- click “Load attributes in IRMA app”;
- click “open IRMA app”;
- click “add data”;
- enter her Yivi pin-code.

Now Alice has the student attribute in her Yivi wallet.

Figure 2.1 also nicely illustrates Yivi’s decentralized architecture. Because attributes are stored only locally on Alice’s device, the issuer does not learn

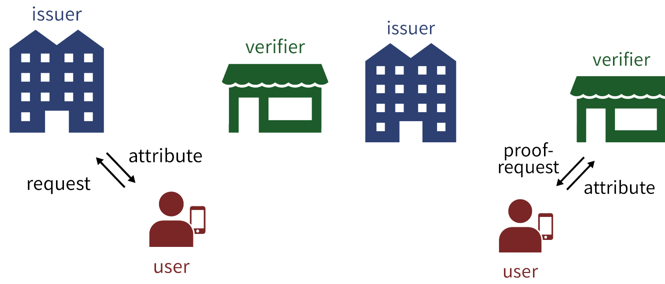


Figure 2.1: Issuance and disclosure of attributes [11].

Alice's internet habits, as opposed to a centralized single sign-on authentication scheme. For the sake of completeness, we need to mention that Yivi contains two components that are not decentralized. Those two components are the scheme manager [12] and the key server [13], both are out of scope for this thesis, but they do not compromise Yivi's privacy-friendly nature.

2.2.1 Idemix

As already mentioned, Yivi is based on IBM's idemix. Idemix is an anonymous credential system that uses non-trivial cryptography. It allows users to authenticate to a party in a privacy-preserving manner. It all starts with attributes. An attribute, as previously described, is a statement or property about someone that is issued by an issuer. Once the user has chosen which attributes to download from an issuer, the attributes are grouped into a credential and blindly signed by the issuer. The details of these *blind* signatures are out of scope for this thesis, but they have two important consequences (i) issuer unlinkability, the issuer cannot trace the usage of the credential and (ii) credential integrity, verifiers can check the integrity of a credential [9]. To disclose attributes to a verifier, a user can selectively show attributes from the credential along with a zero-knowledge disclosure proof—which proves to the verifier that the user:

- knows the non-disclosed attributes; and
- owns a valid signature over the credential;

without revealing the non-disclosed attributes or the signature [13].

2.2.2 Security properties

Below follows a list of security properties provided by Yivi, relevant to our thesis [13]:

Credential unforgeability.
Only issuers can issue valid credentials.

Eavesdropper replay attacks.
: Replay attacks of disclosed attributes are prevented by using a nonce.

Verifier replay attacks.
: Verifiers cannot (ab)use what they learn from disclosures i.e. verifiers cannot disclose the received attributes to other verifiers, because parts of the credential's signature are always hidden using proofs of knowledge—even when all attributes are disclosed.

No impersonation attacks.
: Attributes and credentials reside on the user's device and nowhere else and can therefore only be used by the owner of the device. Technically, issuers could impersonate users, but they are a Trusted Third Party. More mitigations to this attack are described in section 2.2.3.

No privacy hotspots.
: When disclosing attributes only the user and verifier are involved, the issuer does not play a role in this process. This is illustrated in fig. 2.1.

2.2.3 Discussion

Using Yivi, the user is fully in control of their attributes and whether or not they are shared with a verifier. This is great for privacy, however it can also be viewed negatively, since control also means responsibility. The user must maintain their attributes and ensure they are accurate and up-to-date—this is already partly automated, because attributes have an expiration date. It is also important for users to keep an eye on verifiers and what attributes they request, to ensure only necessary attributes are disclosed.

A potential problem with Yivi is the issuer. The issuer is a Trusted Third Party (TTP) and, as the name suggests, this party must be trusted for Yivi ecosystem to function as intended. A potential issue could be the signing of invalid attributes/credentials. To ensure this is the case, a potential Yivi issuer must sign a contract with the Privacy by Design Foundation and SIDN. This contract requires, among other things, proper protection of the IRMA private key [14]. Furthermore, trustworthiness of issuers—and the attributes they issue—can be determined by verifiers themselves. Since the same attribute type can be issued by multiple issuers, verifiers can pick and choose from which issuers they accept attributes. For example, full names issued by a government institution are probably more trustworthy then the same attribute issued by a social media company.

Finally, locally storing attributes can become problematic when a user loses their phone. In this situation, the user must manually re-download all their attributes to their new phone (Yivi wallet), which can become tedious after multiple downloads. A backup feature is in the works [9], that allow users

to backup their Yivi wallet and restore them on a new device.

2.3 Identity-based encryption

The other component used by PG is identity-based encryption (IBE). It was first posed as a challenge in 1984 by Shamir [15] and the first construction was given in 2001 by Boneh and Franklin [16]. Identity-based encryption allows two parties to securely communicate with each other and verify each other's signatures without exchanging public keys. It uses Private Key Generator centers (PKG), whose job is to generate and distribute the necessary keys to the users in the network. It does this using a *Master Public Key* (MPK), which it shares with all the users, and a *Master Secret Key* (MSK), which is kept secret. IBE is based on public key cryptography, but instead of random public keys it uses some unique information about the identity of the user, denoted by `ID`. An `ID` can be any string of characters, but something publicly known to the users on the network, such as an email address, is most practical.

Boneh and Franklin defined four algorithms that form the basis for their IBE scheme. A graphical depiction of the algorithms is shown in fig. 2.2.

Setup
: The PKG chooses the system parameters and computes the MPK and MSK. (Not depicted in fig. 2.2.)

Extract
: The PKG computes for a given `ID`, using its MSK, the private key $d_{\mathtt{ID}}$, so `ID` and $d_{\mathtt{ID}}$ are a public, private key pair. In fig. 2.2, Bob obtains $d_{\mathtt{Bob}}$ in step 5.

Encrypt
: A message M is encrypted using the system parameters, the MPK and a public key `ID`. In fig. 2.2, Alice encrypts a message for Bob with $\mathtt{ID}_{\mathtt{Bob}}$, `bob@example.com`.

Decrypt
: The ciphertext is decrypted using the system parameters and the recipient's private key, $d_{\mathtt{ID}}$. In fig. 2.2, Bob thus decrypts Alice's message with $d_{\mathtt{Bob}}$.

IBE reduces the complexity of en- and decryption for the end users by moving the key management to the PKG. This is however also a drawback in this scheme. The PKG is a Trusted Third Party, who can compute the private key of any user and can therefore decrypt and sign any message made available to them without authorization. As a matter of fact, anyone with access to the MSK can decrypt and sign messages without authorization. Hence it is

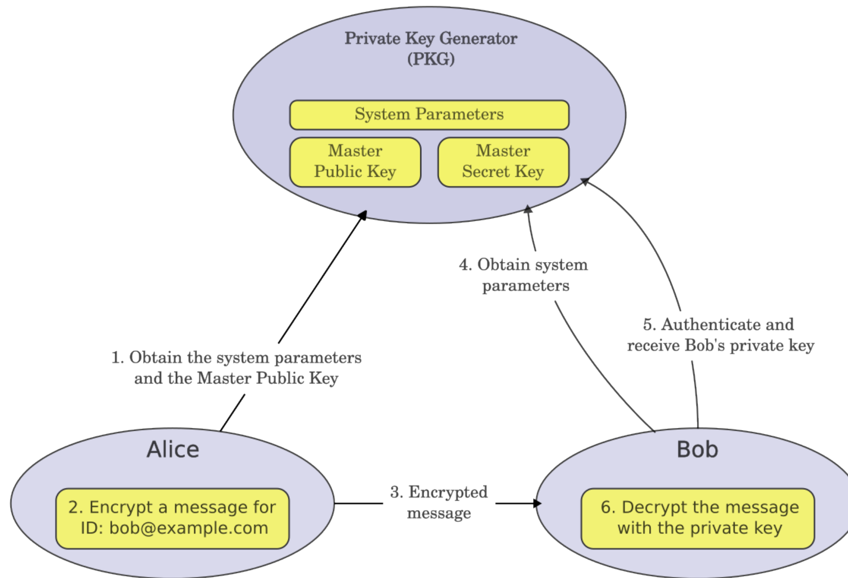


Figure 2.2: Identity-based encryption steps, based on [17].

important for the PKG to properly protect the MSK.

2.4 PostGuard

A major challenge in the wide scale adoption of cryptography and encryption is the key management problem. PG is a project which aims to solve this problem and make encryption accessible for non-technical users. It does this by moving the key management to a Trusted Third Party. More specifically, PG combines IBE—the specific IBE flavor used by PG is discussed in more detail in section 2.4.1—with a digital identity wallet (Yivi) [18]. This combination makes PG a privacy- and user-friendly method of encryption.

PG was originally designed as a solution for email encryption [18], but can also be used to encrypt files or strings of text [19]. To encrypt a file (or a string), the sender has to specify one or more recipients, by their Yivi attribute(s) (e.g. email address). Thereafter, a recipient can decrypt the file—provided that ownership of the attributes, as specified by the sender, can be proven to the PKG. Proving ownership of attributes is done with Yivi. By combining IBE and Yivi, PG reduces—from the user's perspective—decryption to authentication [18].

Yivi is utilized by PG for authentication to the PKG (the Trusted Third Party) using Yivi attributes. These attributes also serve as the recipient's `ID` (i.e. public key) in the IBE scheme. PG refers to these `ID`s as *policies*, and any combination of Yivi attributes can be used as a *policy* for encryption. Only

users possessing all specified attributes can decrypt the message. Notably, *policies* do not need to be uniquely identifying; for example, a message can be encrypted for all students.

In fig. 2.3 a typical PG session is depicted. The dark red actions require user interaction. All other actions, depicted in a lighter red, are automatic.

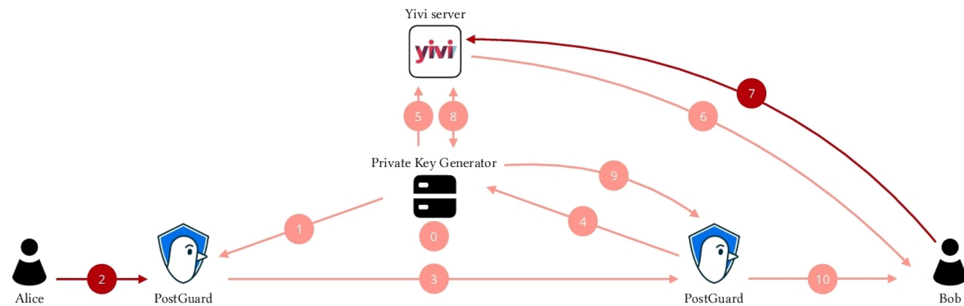


Figure 2.3: The general PG setup and session flow [18].

Each step of fig. 2.3 is described below [20]. Note that digitally signing the message, by Alice, is *not* included in this session flow.

0. Before the first PG session can be started, the PKG must run the IBE `setup` algorithm, which generates a Master Public Key (MPK) and Master Secret Key (MSK).
1. Alice's client retrieves the MPK from the PKG.
2. Alice specifies Bob's identity and writes a message.
3. Alice's client uses the MPK and Bob's identity to encrypt the message and sends the ciphertext to Bob via a possibly untrusted channel.
4. Bob's client asks for a key to decrypt the ciphertext.
5. The PKG starts an authentication session at the Yivi server.
6. Bob is asked to reveal his identity, to the Yivi server, using a QR code.
7. Bob reveals his identity using the Yivi app.
8. The Yivi server sends the authentication results to the PKG.
9. The PKG issues a key for Bob's identity.
10. Bob's client decrypts the ciphertext using his just obtained key.

2.4.1 IBKEM

The IBE flavor used by PG is what they call Identity-based Key Encapsulation Mechanism (IBKEM) [21]. More specifically, it uses CGWKV [22], [23],

which is an IBKEM implementation based on work by Chen, Gay, and Wee [24] with generic transformations for protection against Chosen-Ciphertext Attacks (CCA) [25].

In an IBKEM the `encrypt` and `decrypt` algorithms, from IBE, are replaced with `encaps` and `decaps`, respectively, that allow the sender and recipient to setup a shared secret, `k`. This `k` can then be used to encrypt a message using symmetric cryptography. `encaps(mpk, id)` → `k,c` generates—from the PKG's master public key and an IBE identity—a secret key, `k`, and encapsulates `k` to a ciphertext, `c`. `decaps(usk,c)` → `k` decapsulates—from the *user secret key*, `usk`, and the ciphertext, `c`—to the secret key, `k`. In this scenario, `id` and `usk` are a public, private key pair. `usk` is equivalent to IBE's $d_{\mathtt{ID}}$, as defined in section 2.3, and is generated by the PKG. The shared secret key, `k`, can then be used in a more efficient symmetric encryption algorithm. PG uses `AES-128-GCM` [21]. In section 2.4.2 we briefly mention how PG ensures `k` is different for every encapsulation.

PG can also encrypt a message for multiple recipients at once using Multi-Recipient Identity-Based Key Encapsulation (mIBKEM). This encapsulates a single shared secret for multiple identities, using the `multi_encaps` and `multi_decaps` algorithms [23].

2.4.2 PostGuard and JavaScript

To integrate PG into web applications, PG offers `pg-wasm`, a library with WebAssembly bindings that enable JavaScript or TypeScript to call into the PG Rust library [26].

When using `pg-wasm` for encryption a *policy* must be provided that specifies the recipients. A policy is a JSON object of key value pairs where the key is a recipient identifier and the value an object containing a timestamp and an array of attributes. For example an encryption policy for Bob with email attribute bob@example.com looks like:

```
{
  Bob: {
    ts: 1721383918,
    con: [{
            t: "irma-demo.sidn-pbdf.email.email",
            v: "bob@example.com"
         }],
  },
}
```

In the example, `con` is short for conjunction and when a recipient wants to decrypt a message it must prove ownership of all specified attributes in

the conjunction. In this particular conjunction there is only one attribute (`bob@example.com`). Each attribute, in the conjunction, can be further destructed to an object with a `type`, `t`, and a `value`, `v`. The type is a Yivi attribute type, in the form `scheme.issuer.credentialtype.attribute` and must be defined in the Yivi scheme manager [12], and the value is what the recipient is supposed to prove ownership of. The `ts`, in the example, is a Unix timestamp; its purpose is to ensure that the IBKEM shared secret, `k`, is different for every encryption.

Messages encrypted with PG are also digitally signed (with PG), before encrypting—it uses a so-called hybrid Sign-then-Encrypt (StE) protocol [21]. To do this a similar conjunction of attributes, as in the previous example, must be specified. To sign a message the signer must prove ownership of these attributes. Signing can be done with public attributes, visible to everyone, and hidden attributes, visible only to recipients. Signing is a relatively new feature in PG and is not yet very well documented. Therefore, we will not discuss it further.

Chapter 3

Development of the prototype

This chapter will discuss the development of the prototype, the issues we encountered and the subsequent decisions made to resolve them.

3.1 PostGuard encryption

As mentioned in the introduction, PG can encrypt both files and strings of text. Initially, we attempted to encrypt the files themselves directly using PG. This option turned out to lead to an unforeseen technical problem. It resulted in a clash between the input data type expected by FS, a file object, and the output data type of PG, a writable stream. As far as we are aware, there is no efficient conversion between these two data types and as a result we would have to either:

1. rewrite a large part of the FS code base; or
2. temporarily write the PG encrypted file(s) to the file system and upload those using the existing FS code.

Both of these solutions have serious drawbacks. Option 1 is undesirable because it will take a lot of time and we prefer to minimize the impact of our prototype on the FS code base, i.e. leave as much of the existing FS code as possible untouched, to make it easier to implement a production ready version of this prototype. Option 2 will impact the upload speed, especially for large files, since every file must first be written to the local file system before it can be uploaded to the FS server. This approach also requires at least double the size of the to-be-uploaded files in disk space, because both the original and the encrypted file will be on the disk, at least for the time between encrypting and uploading.

We decided before development had started, to leave the original FS password encryption in place and let the end-users decide whether to use it or PG for file encryption. As a result, when combined with the above proposed solution for PG file encryption, a file can be encrypted using either a password or PG. This forces the uploader to choose which decryption method is used by the recipient. However, it would be preferable to have password decryption as a backup when files are encrypted using PG, allowing recipients to decrypt the files even if the PG or Yivi servers are offline. Note that the recipient, when choosing password decryption, must have received the encryption password from the uploader.

To resolve these two issues two other options were conceptualized. The first option was to add the encryption password to the email that is sent to the recipients and encrypt this email using PG. This would solve all the previously mentioned problems. A small downside with this option is that the recipient has to still manually copy and paste the password from the email to the FS webpage. However, the bigger issue is that emails are generated on the FS server, but the password cannot leave the client's browser unencrypted. Therefore, to include the password in the email, the emails will have to be generated and encrypted client-side. This would involve rewriting a lot of the email back-end (PHP) to front-end (JavaScript). Another possibility could be to send the (on FS's server) generated email to the client, allow the client to add the password and encrypt the email using PG, send it back to FS's server, and finally let FS send the encrypted email to the recipient. Clearly, these solutions are anything but elegant.

The second option involves encrypting the password used by FS to encrypt the files with PG and adding the PG-encrypted password as metadata to the file transfer. This metadata can then be stored on the FS server alongside the encrypted files. To include the PG-encrypted password in the metadata without modifying FS's back-end, it can be incorporated into an existing meta data option. To facilitate the detection and extraction of the password from the option, it should be surrounded by unique magic bytes. This process is discussed in more detail in section 4.1.

The second option solves all previously mentioned problems and also circumvents the issues encountered when encrypting emails using PG. Moreover, this will allow us to reuse the existing FS code for file encryption. Therefore, this is the option that is used in the final prototype.

3.2 The new user experience

In this section we will discuss what we have changed in FS and how the prototype can be used. Before we do this we must mention that only email addresses can be used as PG-encryption and -signing attributes. This limi-

tation is further discussed in section 4.2.

3.2.1 User Interface

Using PG as described in section 3.1 (encrypting the FS encryption password with PG), it is possible to hide the usage of the FS password entirely. However, we have opted to leave original password encryption in place to let the end-user decide which en- or decryption method to use. This choice will result in a less clean UI for the upload and download page, but in our opinion, this is outweighed by having (i) user choice and (ii) a backup en- and decryption method. To assist users in navigating this more complex UI, clear instructions on using the new PG feature must be provided.

This means that a new PG encryption checkbox has been added to the upload page. It will be displayed when the "File Encryption" checkbox is checked and hidden otherwise. When the PG encryption checkbox is checked, it is mandatory to specify recipients by email address. Furthermore, with PG encryption, signing messages is mandatory; this is done using the uploader's email address as signing attribute. If the uploader has multiple email addresses linked to their FS account, then an email address can be picked from a selection menu. This selection menu already exists in the original FS UI, it is used as the sender email address when recipients are notified by email about the file transfer.

On the download page a new "PG Download" button has been added, which will decrypt a file using PG. If the file(s) were encrypted using PG for multiple recipients, to decrypt using PG, the recipient must also specify their attribute (email address) used for PG decryption. The recipient can do this via a newly added selection menu, which will contain all recipients' email addresses. If the file(s) were encrypted (using PG) for only one recipient, the (PG) decryption will automatically use the sole available email address.

3.2.2 Walk-through of the new user experience

This section describes the new user experience (UX), uploading and downloading files, with FS using PG. We do this by following Alice and Bob through an example usage of our prototype. The following sections assume the uploader is not a guest FS user and is already logged in to FS. Additionally, it is assumed that, both uploader and recipient have Yivi installed on their phone, with an email identity/attribute linked to their Yivi account. Alice, `alice@example.com`, and Bob, `bob@example.com`, are the uploader and recipient respectively.

Uploading files

Alice wants to send a confidential file to Bob using FS and encrypt it with PG. To do this she (the steps are also depicted in fig. 3.1):

1. goes to the FS upload page in her browser;
2. chooses the file to be uploaded;
3. ticks the "File Encryption" checkbox
4. enters a valid encryption password or lets FS generate one;
5. ticks the "PostGuard Encryption" checkbox;
6. specifies Bob's email address, `bob@example.com`;
7. picks the attribute (i.e. email address) used for signing the upload;
8. clicks "Send", which starts a Yivi session and displays a Yivi QR code in the browser for signing;
9. using her smartphone:
    1. scans the QR code;
    2. discloses her email address to PG.

Note that in step 8, PG digitally signs the FS encryption password. This does not contribute anything to the authenticity or integrity of the uploaded file(s), but signing is a mandatory part of the PG encryption process and cannot be bypassed. This information is therefore discarded on the download page.

After step 9 is successfully completed, the encrypted files will be on the FS server and can be downloaded by Bob. The encryption of the files by FS as well as the encryption of the FS encryption password by PG is done client-side.

It is important to note that recipients must be specified explicitly, as opposed to sharing a download link, in our case by email address; they are used by PG as (part of) the public key for encryption. This is an important privacy consideration. When using PG, the FS server will know the identity of the recipients, because it will send an email with a download link to all the recipients.

Technically, it is possible to use only email addresses for PG encryption and allow the uploader to manually distribute the download link. At first glance, this might seem to hide the recipients' identities from the FS server. However, the recipients' email addresses are also visible in the PG header of the encrypted password. Therefore, it is not possible to keep the recipients'

identities hidden from FS when using PG. This conclusion is based on the implementation details discussed in section 4.1 (see the "Download Page" discussion).

For Alice to upload more than one file at once she simply has to select more files during step 2, all the other step remain the same.

Downloading files

After Alice has uploaded the file, Bob receives an email from FS with a download link. To download and decrypt the file Bob (the steps are also depicted in fig. 3.2):

1. follows the link in the email, this opens the FS download page in Bob's browser;
2. if the file was encrypted for multiple people, he chooses his email address from the selection menu, used as the attribute for decrypting the download, otherwise no action is required;
3. clicks the "PostGuard Download" button, on the FS download page, which starts a Yivi session and displays a Yivi QR code in the browser;
4. using his smartphone:
    1. scans the Yivi QR code;
    2. discloses his email address to PG.

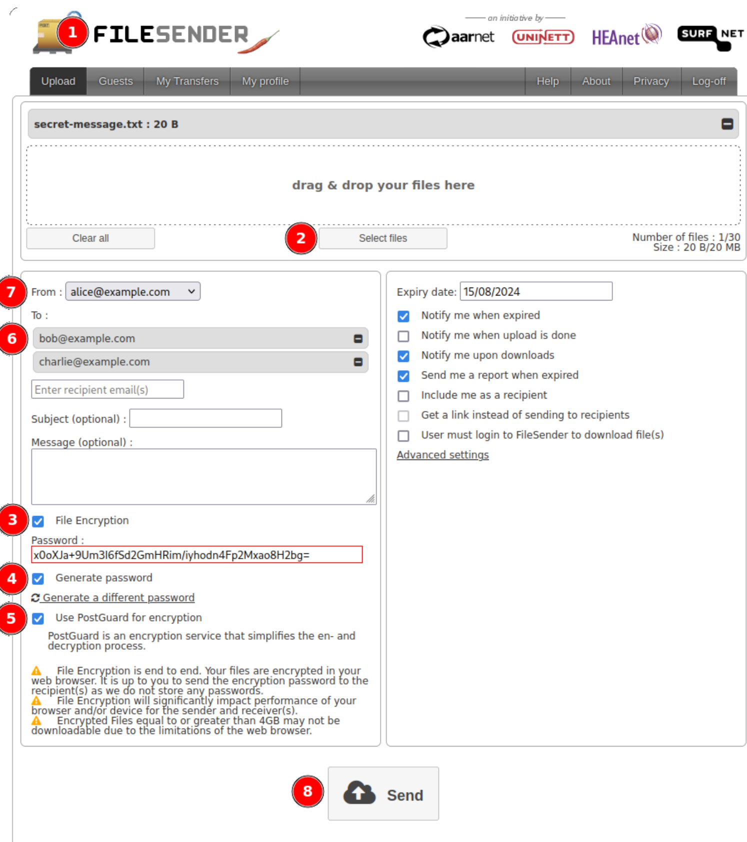
After step 4.2 is successfully completed, the PG encrypted password is decrypted, with which the downloaded file can be decrypted, all in Bob's browser. When there are multiple files available for download, Bob has to do step 3 for each file. However, because FS caches the decryption password, subsequent downloads will not require Bob to authenticate himself again to PG's Private Key Generator (PKG).

As mentioned in the section on uploading files, the PG signature is discarded as it does not provide any useful information.

3.3 Webpack

One technical hurdle we encountered when developing the prototype was what to encrypt using PG (section 3.1). Another even more technical one, but not less important, has to do with JavaScript dependency management, which brings us to Webpack [27].

To understand why we have opted to use Webpack, or in more general terms a JavaScript bundler, some knowledge about JavaScript and its history is required.



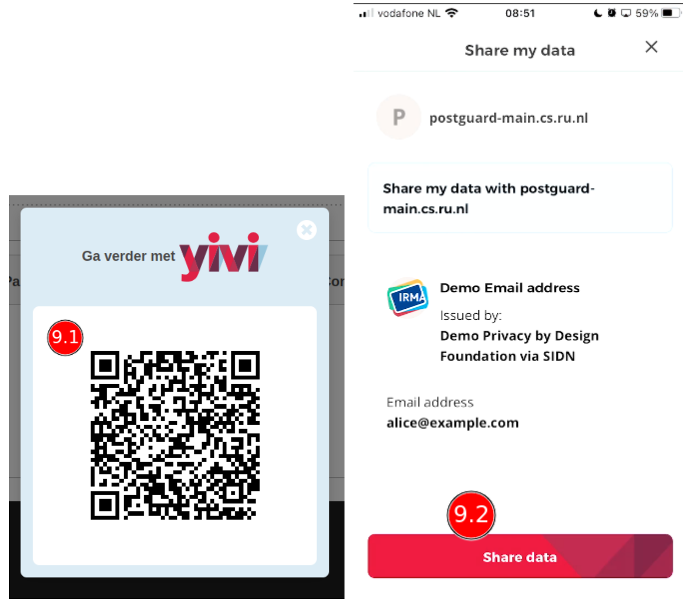


Figure 3.1: Steps for uploading a file with PG.

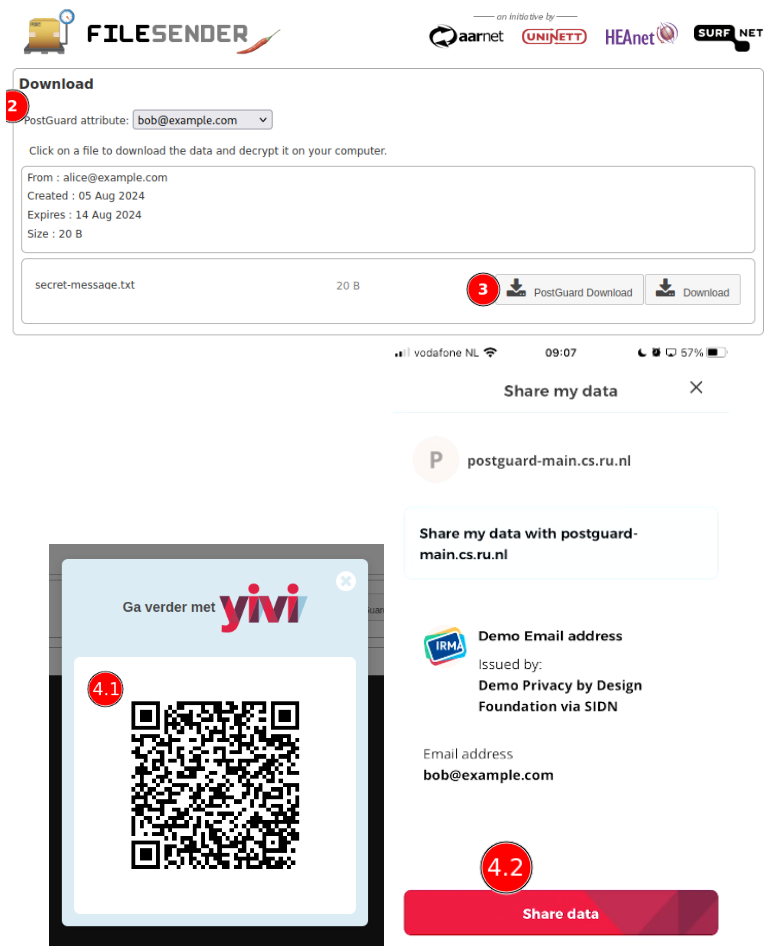


Figure 3.2: Steps for downloading a file with PG.

3.3.1 History

Technically there is not just one JavaScript; each browser implements/uses its own JavaScript engine, which is the software that executes the JavaScript code. Since the minor differences in JavaScript support for different engines/browsers is not of importance to us, we will for the remainder of this section, refer to JavaScript as the language accepted by all JavaScript engines.

JavaScript, when it was first released in 1995 [28] had no true support for modules. At that time, developers created "modular" code by including each required JavaScript file individually on every HTML page, using the `<script>` tag with a `src` attribute pointing to the "module". This method of manually managing dependencies comes with some serious drawbacks [29]:

- Namespace pollution. All functions and variables are added to the global namespace.
- Manual dependency management. The order in which the JavaScript files are loaded on an HTML page must be determined manually.

Then in 2009, the first Node.js, a JavaScript runtime environment, was released [30]. Since then it has made JavaScript a popular language for server-side programming. To support modules, Node.js originally used the CommonJS module system; since version 12 (released 2019), it also supports the ES6 module system [31], discussed later. CommonJS solved the modularization issues in JavaScript, at least on the server-side. Unfortunately, CommonJS is not supported by browsers, client-side.

Skipping forward to 2015, ECMAScript 2015, also referred to as ES6, was released [32]. ES6 is a specification for a general purpose scripting language, which is implemented by JavaScript. ES6 added, among other things, support for JavaScript modules by defining the `import` and `export` functions [32, Sec. 15.2.2–15.2.3]. This solved the earlier mentioned modularization issues, but this time on the client-side.

Since CommonJS already solved the modularization issues long before ES6, a lot of JavaScript modules have been and are still written in CommonJS. Some of these modules can also be useful in front-end code. However, since these CommonJS modules cannot run directly in the browser, a solution was needed.

3.3.2 Bundlers

The solution is a JavaScript bundler. For this section we will use Webpack as example, but bundlers in general do mostly the same things.

One of the things that Webpack does is building a dependencies graph—for all local files and third party dependencies—from one or more entry files. Using this graph, it combines all the dependencies into one or more bundles, which are static assets such HTML or JavaScript files [33]. Webpack also transpiles modules (e.g. CommonJS and ES6 modules [34]) into browser executable JavaScript, using so-called loaders [35]. Hence bundlers are very useful when developing a web application with lots of third party dependencies.

FS itself has only a small number of dependencies and its developers have decided to not use a bundler, but instead manage its dependencies manually—this method has been discussed in section 3.3.1. The reason for this is probably because development of FS started before ES6 was released. The number of dependencies of FS will increase significantly when PG has been integrated. Moreover, many of PG's dependencies are written in CommonJS. Due to this and the fact that `pg-wasm` "has been configured to run in a browser via a bundler" [26], we find it necessary to add a bundler to FS.

There are many JavaScript bundlers, some of the more popular ones are, Webpack, Rollup and ESBuild. We have decided to use Webpack because it is:

- open-source;
- very popular, see fig. 3.3;
- used by the `pg-example` repository and therefore comes with a working Webpack configuration file [19].

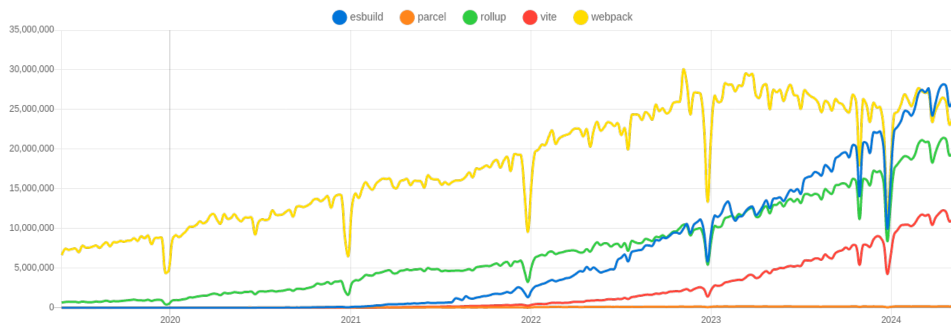


Figure 3.3: JavaScript bundler downloads over the past 5 years [36].

3.3.3 Content Security Policy

A `Content-Security-Policy` (CSP) is an HTTP response header which restricts the resources (JavaScript, CSS, etc) that can be loaded on a webpage and where they can be loaded from. Its purpose is to prevent, or at least limit

the attack surface of, cross-site scripting (XSS). FS, by default, has a pretty strict CSP and allows (for the most part) only resources originating from FS's own domain—which is possible, because FS manages its dependencies manually.

A CSP consists of policy directives, each of which describes the policy for a certain resource type or policy area [37]. Examples of directives are *font*, *script* and *style*. If a directive is not specified in the CSP, the *default* directive is used as a fallback.

The JavaScript code produced by Webpack during the bundling process does *not* comply with FS's default CSP. It contains calls to `eval`, which can cause security vulnerabilities—this is an old and well known problem within the Webpack community [38], [39], [40]. When the browser tries to load a bundle, with calls to `eval` and FS's strict CSP, it will detect this and complain about unsafe calls to `eval`—and it will not load the bundle.

While FS does offer a configuration option to completely disable CSP, this should be avoided for obvious reasons. Instead, we adjusted the Webpack configuration file to prevent it from generating code with `eval` calls [38]. However this resulted in unsafe `eval`s of WebAssembly code, due to the use of `pg-wasm`. Additionally, the Yivi front-end packages modify webpages using inline styles, causing a violation of FS's `style-src` directive. Our only solution to address these issues is to relax FS's CSP, and thus allow *unsafe* evaluation of WebAssembly and inline CSS style changes.

After relaxing the *script* and *style* directives another CSP error surfaced related to the *connect* directive, which restricts the URLs which can be loaded using script interfaces [41]. In our case, PG makes a `fetch` call to PG's PKG. This is one of the APIs restricted by the connect directive. The best solution to this problem is to allow fetching, but only to the PKG's url.

The following is FS's default CSP:

- `default-src 'self';`
- `object-src 'self';`
- `img-src 'self' data:;`
- `media-src 'none';`
- `frame-src 'self';`
- `font-src 'self';`
- `connect-src 'self';`
- `script-src 'self';`

- `style-src 'self';`

To make PG work we have modified the final three directives:

- `connect-src 'self' https://postguard-main.cs.ru.nl/`
- `script-src 'self' 'wasm-unsafe-eval`
- `style-src 'self' 'unsafe-inline'`

To minimize the attack surface of XSS, we only relaxed the CSP on the web pages that use PG, i.e. the upload and download page.

Chapter 4

Discussion

In this thesis we have explored whether or not it is possible to integrate PG in to FS. To do this we posed two sub-questions:

1. What to encrypt using PostGuard?
2. How will Filesender change?

Sub-question 1 has been answered in section 3.1, where we have discussed three different options for encrypting files using PG. We have concluded that encrypting the FS encryption password using PG is the best of those three options. Then in section 3.2 we have answered sub-question 2 and have shown what has to change in the UI to accommodate PG as a new feature and how this will impact the UX. Finally in section 3.3 we went over our reasons for using a JavaScript bundler and how this would impact FS—this required minor changes to the back-end. Ultimately, we conclude that it is technically feasible to integrate PG into FS and have shown how this can be done by implementing a prototype.

However, just because it is technically feasible does not necessarily mean it is a good idea. While we think that the end-user will definitely benefit from using PG in FS, there are several important factors to consider.

The number of dependencies pulled in by PG is significant and as a result breakage of a dependencies and therefore the entire FS+PG stack is a risk. It may be possible to trim the dependency list a little bit by removing unnecessary dependencies from the Webpack config or `package.json`, as they were copied over from the `pg-example` repository; most of the low-hanging fruit has already been removed.

Moreover, PG has not yet released a stable version and its source code has not yet been audited, as prominently advertised in the READMEs of PG's

repositories. Therefore, it may not be advisable to use PG in a production environment.

Despite these reservations, we think integrating PG into FS is a worthwhile effort, but should only be merged into a production environment after PG has had a security audit.

If someone decides to work out this prototype we would like to make some recommendations.

Keep password encryption as a backup
: We value user choice highly and we think the users of FS also do. Additionally, it can function as a backup en- and decryption method when, for whatever reason, the PG and/or Yivi servers are offline. Hence keeping this feature in FS will be appreciated by the users.

  If it turns out that the PG feature as proposed in our prototype is to complicated to use by casual users, it is also possible to include the password in an advanced PG options section, which would hide it from casual FS users while providing advanced users with greater control if needed.

Use a bundler
: Using a bundler is practically necessary when integrating PG into FS, because of the number of dependencies of PG and because it is recommended by `pg-wasm`. And while it is possible to use the bundler solely for generating the PG related files—we recommend also using a bundler for FS itself, because using one has more benefits beyond being able to run dependencies written in CommonJS in the browser.

We think that working out this prototype is a nice project for either a Master student or one of the developers of the PG or FS team.

4.1 Evaluation of the prototype

In this evaluation of the prototype we will discuss 3 areas: upload page, download page, and back-end. We will focus the discussion on usability and implementation details.

Upload page
: The upload page requires quite a few user actions before files can be uploaded, with PG encryption:

  - selecting files
  - checking the file encryption checkbox
  - enter a password

- **checking the PostGuard encryption checkbox**
- **specify the signing attribute**
- specifying recipients

The bold items are added by this prototype. The UI element used to specify the signing attribute already exist in the original FS and therefore the only new UI element is the PG encryption checkbox. There is little to improve, in terms of user experience, with regard to the new element, except maybe the PG encryption checkbox should not be hidden behind the file encryption checkbox, i.e. it only shows when "File encryption" is checked.

We have implemented the upload page in such a way that every step, mentioned above, must be completed before the user can start the upload process, re-using existing functions where possible.

Download page

The steps to decrypt a file are (i) choose one's own decryption attribute (email address), (ii) click the "PostGuard Download" button of the file that must be downloaded, and (iii) disclose the chosen email address with Yivi.

We have decided to add an new "PostGuard Download" next to the original download button to make it obvious to the user which method of decryption will be used. This new button is only displayed when PG can actually be used for decryption.

Furthermore, the placement of the selection menu, for the PG decryption attributes, can be improved. Currently it is rather far away from the download button perhaps placing it closer by may make usage of the PG functionality more obvious to the user.

A problem area on the download page is: how to display the decryption attributes. There are three options. The first is to show all email addresses associated with the current account, this is easy to implement, but will result in displaying attributes that cannot be used. The second option: filter the list from option 1 to only those options that can be used for decryption; this may lead to an empty list and therefore no way to decrypt the file(s). And the third option, which is the option we have decided to implement, is to show all attributes that can be used for decryption. This is however also not a great option since the actual values of PG attributes required for decryption are redacted by PG. The PG header for an encrypted message, as received from the Private Key Generator (PKG), will look like:

```
{
```

```
 'Bob': {
   ts: 1643634276,
   con: [
     { t: "irma-demo.sidn-pbdf.email.email", v: "" },
   ],
 },
}
```

This is the same JSON object as specified during the encryption, except the value `v` is redacted and hence cannot be used to display as possible decryption attribute. It is possible, during the PG encryption, to set for every recipient the key (in the example there is just one recipient: "Bob") to their email address. So the previous example will become:

```
{
 'bob@example.com': {
   ts: 1643634276,
   con: [
     { t: "irma-demo.sidn-pbdf.email.email", v: "" },
   ],
 },
}
```

Now the key can then be used as possible decryption attribute on the download page. This is how we have implemented it in the prototype. When there are multiple email addresses in the header, the user can choose their own from a selection menu. This solution obviously does not scale—multiple attributes in the `con` cannot easily be added to a key—and also leaks the recipients' email addresses to everybody with access to the encrypted message.

Back-end

Due to limitations in time, we did not modify very much in the FS back-end—except for a minor change to the CSP. Instead, as already discussed in less detail in section 3.1, to save the PG encrypted password on the FS server we added the password to an already existing option, we have chosen the "message". Before uploading, the encrypted password is prefixed to the message using the following form: `_PG_<password>~`, where `<password>` is the hex encoded, PG encrypted, password. On the download page the message is checked for the magic bytes, `_PG_`, and when found the password is carved out and removed from the message.

Even though piggybacking on the message option does not seem to cause any bugs, ideally the encrypted password should have a dedicated place in the back-end.

Coming back to CSP, even though we have restricted the policy as far as we were able to and added this relaxed CSP to only the up- and download pages, care should be taken when deciding to run this in production. Cross-site scripting is a serious threat, but because everyone's threat model is different we can *not* give a recommendation.

4.2 Limitations

For simplicity, the prototype only implements PG encryption for uploading/encrypting and downloading/decrypting normal files, not archives. Additionally, we have restricted the use of encryption and signing attributes exclusively to email addresses.

Due to time constraints, we have only used webpack to generate PG-specific files, rather than refactoring the entire FS application to use webpack.

Similarly, because of time limitations, we did not set up an SMTP server to send emails notifying recipients of new file transfers. Instead, FS always generates a download link that can be distributed manually. The outcome remains the same since the email would contain the same type of link, directing the recipient to the same download page. This approach simplifies prototype usage, particularly since, in most cases, files are encrypted for self-testing to verify that the prototype functions as advertised.

Since we were unable to get SimpleSAMLphp working properly, the prototype is configured to use what FS refers to as a "fake" account. This means FS bypasses the authentication process, using the same account for every user visiting a FS webpage. While this "fake" account is typically used by FS for automated testing, it is also suitable for development purposes. This method poses no issues, as authentication to FS is not critical for demonstrating the PG functionality within FS.

4.3 Future work

While we have a working prototype, there is still a lot of things that can be improved and extra features that can be added. The obvious being to implement the prototype upstream. This requires, at least, implementing PG for guest users, upload and download page, and for decrypting archives, for authenticated and guest users. Furthermore, a choice must be made, whether or not to use a bundler or not. In which case, either the entire FS application should be rewritten using the bundler or to manage all the, including PG's, dependencies manually. A third option could be to use the bundler only to generate the PG specific files, which is how we have developed the prototype.

A more interesting suggestion—and in our opinion one of PG's key features—let FS user encrypt files with other (including non-identifying) attribute types (e.g. age or gender). For example, PG can encrypt a message for all students using only one attribute. Adding this to FS will simplify the upload process for end-users. This is actually already possible by using mailing list email addresses, however PG attributes are more flexible.

As the author is a programmer and not a UX expert, more research should be done on how to effectively display PG attributes to the user. This is especially necessary when multiple attributes and different types of attributes can be used for en- and decryption, as suggested above. PG was initially created for email encryption and in this setting it is easy to infer the en- or decryption attributes. In a setting like ours, where users may want to use different types of attributes, UX will be more of a problem. As far as we are aware, there has not yet been any research done in this area or a project using PG in this way.

As discussed in section 3.2.2, the digital signature generated by PG is discarded in this prototype, because it is done over the encryption password and, therefore, does not contribute anything to the authenticity or integrity of the uploaded files. These security properties are desirable in an application like FS and can be achieved by signing the files using PG. The most straightforward approach is likely to sign the hash of each file, as signing the files directly would lead to the same issue as encountered in section 3.1.

Finally, while not related to file encryption, Yivi can be utilized as an authentication method for FS, aligning with the recent trend toward passwordless authentication.

Chapter 5

Conclusion

We have demonstrated that PG can be integrated into FS through the development of a prototype, which we have evaluated and whose limitations we have discussed. For a potential production-ready version, we have provided recommendations and identified areas for improvement.

Lastly, we have made the source code of the prototype publicly available, hosted on gitlab [42] and github [43]. The prototype was developed using PostgreSQL as the database and Apache HTTP server as a website server.

Due to challenges in aligning specific versions of various software components, we have also written several Dockerfiles and Docker Compose files in addition to the prototype. These resources are designed to simplify experimentation with and development on the prototype. For instructions on setting up and using the prototype, please refer to the README of the repository.

We hope this thesis and the accompanying prototype serve as valuable resources for those who wish to further develop this project.

Bibliography

[1] FileSender. [Online]. Available: https://www.filesender.org.

[2] WeTransfer. [Online]. Available: https://wetransfer.com.

[3] PostGuard. [Online]. Available: https://postguard.eu.

[4] Yivi. [Online]. Available: https://www.yivi.app.

[5] FileSender, *Known FileSender installations*. [Online]. Available: https://docs.filesender.org/filesender/known-installs (visited on 08/15/2024).

[6] RedCLARA. [Online]. Available: https://filesender.redclara.net/filesender/.

[7] GÉANT. [Online]. Available: https://filesender.geant.org/.

[8] FileSender, *Version 2.0 detailed feature list (draft)*, version 2.0. [Online]. Available: https://docs.filesender.org/filesender/v2.0/features/ (visited on 08/04/2024).

[9] Privacy by Design Foundation, *Yivi in detail*. [Online]. Available: https://privacybydesign.foundation/irma-explanation/ (visited on 08/04/2024).

[10] J. Camenisch and E. Van Herreweghen, "Design and implementation of the idemix anonymous credential system," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 21–30.

[11] Privacy by Design Foundation, *What is IRMA?* Version 0.15.0. [Online]. Available: https://irma.app/docs/what-is-irma/.

[12] Privacy by Design Foundation, *Irma schemes*, version 0.15.0. [Online]. Available: https://irma.app/docs/schemes/.

[13] Privacy by Design Foundation, *Technical overview*, version 0.15.0. [Online]. Available: https://irma.app/docs/overview/.

[14] Privacy by Design Foundation, *Issuer guide*, version 0.15.0. [Online]. Available: https://irma.app/docs/issuer/.

[15] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Advances in Cryptology*, Springer Berlin Heidelberg, 1985, pp. 47–53, ISBN: 978-3-540-39568-3.

[16] D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Advances in Cryptology — CRYPTO 2001*, Springer Berlin Heidelberg, 2001, pp. 213–229, ISBN: 978-3-540-44647-7.

[17] Y. Sheffer, *File:Identity Based Encryption Steps.png*, May 26, 2009. [Online]. Available: `https://en.wikipedia.org/wiki/File:Identity_Based_Encryption_Steps.png` (visited on 08/04/2024).

[18] L. Botros, M. Brandon, B. Jacobs, D. Ostkamp, H. Schraffenberger, and M. Venema, "Postguard: Towards easy and secure email communication," in *CHI Extended Abstracts*, ACM, 2023, 232:1–232:6.

[19] L. Botros, *PostGuard Example*, 2022. [Online]. Available: `https://github.com/encryption4all/pg-example/tree/01d313aab78fa89dae0be37eb653f3012d5ad255`.

[20] L. Botros, *PostGuard*. [Online]. Available: `https://github.com/encryption4all/postguard/tree/e0e1dbb4f2d70b25761b2060ff7c949ee3684ab5`.

[21] L. Botros, *PostGuard Core*. [Online]. Available: `https://github.com/encryption4all/postguard/tree/e0e1dbb4f2d70b25761b2060ff7c949ee3684ab5/pg-core`.

[22] L. Botros, *Module ibe::kem::cgw_kv*. [Online]. Available: `https://docs.rs/ibe/latest/ibe/kem/cgw_kv/index.html` (visited on 06/23/2024).

[23] L. Botros, *IBE*. [Online]. Available: `https://github.com/encryption4all/ibe/tree/8c18dec0b7769b238558872317e4c30da31fc3ac`.

[24] J. Chen, R. Gay, and H. Wee, "Improved dual system abe in prime-order groups via predicate encodings," in *Advances in Cryptology - EUROCRYPT 2015*, Springer Berlin Heidelberg, 2015, pp. 595–624, ISBN: 978-3-662-46803-6.

[25] M. Venema and L. Botros, "Efficient and generic transformations for chosen-ciphertext secure predicate encryption," *Cryptology ePrint Archive*, 2022.

[26] L. Botros, *pg-wasm*. [Online]. Available: `https://github.com/encryption4all/postguard/tree/e0e1dbb4f2d70b25761b2060ff7c949ee3684ab5/pg-wasm`.

[27] Webpack. [Online]. Available: `https://webpack.js.org`.

[28] Netscape Communications Corporation, *Netscape and Sun announce JavaScript*, Dec. 5, 1995. [Online]. Available: `https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html` (visited on 06/24/2024).

[29] L. Clark, *ES modules: A cartoon deep-dive*, Mar. 28, 2018. [Online]. Available: `https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/` (visited on 06/24/2024).

[30] R. Dahl, *Node*, version 0.0.1, May 27, 2009. [Online]. Available: `https://github.com/nodejs/node-v0.x-archive/releases/tag/v0.0.1`.

[31] NodeJS, *Differences between Node.js and the Browser*. [Online]. Available: https://nodejs.org/en/learn/getting-started/differences-between-nodejs-and-the-browser (visited on 06/24/2024).

[32] ECMA International, *ECMAScript 2015 Language Specification*, version 6.0, Jun. 2015. [Online]. Available: https://262.ecma-international.org/6.0/.

[33] Webpack, *Concepts*, version 5. [Online]. Available: https://webpack.js.org/concepts/.

[34] Webpack, *Modules*, version 5. [Online]. Available: https://webpack.js.org/concepts/modules/.

[35] Webpack, *Loaders*, version 5. [Online]. Available: https://webpack.js.org/concepts/loaders/.

[36] J. Potter, *Npm trends*, Downloads in past 5 years. [Online]. Available: https://npmtrends.com/esbuild-vs-parcel-vs-rollup-vs-vite-vs-webpack (visited on 05/29/2024).

[37] Mozilla, *Content Security Policy (CSP)*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP (visited on 08/05/2024).

[38] bakrkawkji-trustly, *Calling function or eval should be eliminated*, Sep. 5, 2017. [Online]. Available: https://github.com/webpack/webpack/issues/5627 (visited on 08/04/2024).

[39] L. Cordova, *CSP does not allow eval*, Jul. 31, 2017. [Online]. Available: https://github.com/webpack-contrib/script-loader/issues/39 (visited on 08/05/2024).

[40] Flai, *Webpack 4 build bricks CSP with unsafe-eval*, Feb. 5, 2019. [Online]. Available: https://stackoverflow.com/questions/54542737/webpack-4-build-bricks-csp-with-unsafe-eval (visited on 08/05/2024).

[41] Mozilla, *CSP: Connect-src*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/connect-src (visited on 08/05/2024).

[42] B. Rinders, *FileSender + PostGuard Prototype*. [Online]. Available: https://gitlab.com/postguard-filesender/filesender.

[43] B. Rinders, *FileSender + PostGuard Prototype*. [Online]. Available: https://github.com/bryanrinders/filesender.