Towards Model-Based Development of Decentralised Peer-to-Peer Data Vaults

Alfa Yohannis
University of York
York, United Kingdom
alfa.yohannis@york.ac.uk

Alfonso de la Vega
University of York
York, United Kingdom
alfonso.delavega@york.ac.uk

Delaram Kahrobaei
University of York
York, United Kingdom
delaram.kahrobaei@york.ac.uk

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

ABSTRACT

Using centralised data storage systems has been the standard practice followed by online service providers when managing the personal data of their users. This method requires users to trust these providers and, to some extent, users are not in full control over their data. The development of applications around decentralised data vaults, i.e., encrypted storage systems located in user-managed devices, can give this control back to the users as sole owners of the data. However, the development of such applications is not effort-free, and it requires developers to have specialised knowledge, such as how to deploy secure and peer-to-peer communication systems. We present Vaultage, a model-based framework that can simplify the development of data vault applications. We demonstrate its core features through a social network application case study and include some initial evaluation results, showing Vaultage's code generation capabilities and some profiling analysis of the generated network components.

CCS CONCEPTS

• **Software and its engineering** → **Development frameworks and environments**; *Automatic programming*; • **Security and privacy** → *Software security engineering*.

KEYWORDS

Data Privacy, Encryption, Decentralised Data, Model-Driven Engineering, Generative Programming

ACM Reference Format:
Alfa Yohannis, Alfonso de la Vega, Delaram Kahrobaei, and Dimitris Kolovos. 2020. Towards Model-Based Development of Decentralised Peer-to-Peer Data Vaults. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3417990.3420043

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3420043

1 INTRODUCTION

The majority of online software-based service providers manage the data of their users in a centralised manner. This practise requires users to transfer control of their personal data to their service providers by uploading it to remote servers. This control transfer is almost always an unavoidable step when users want to use any of the provided services (e.g. email, social networks, search engines), and it may come with consequences: it limits users' self-management of their data, and this data could be used irresponsibly (i.e. third parties using data without its owners' consents).

Most service providers use Terms of Service (ToS) to define how users' data will be employed and protected, and it is up to the final users to accept the terms. Unfortunately, most of these ToSs are expressed in a language that is complex and tedious to understand [9], and are presented in ways that might direct users to ignore the terms and immediately jump to use the provided services [17].

In order to bring back control of personal data to users, the European Union has issued the General Data Protection Regulation (GDPR) – a set of rules regarding the processing, movement, and protection of personal data [18]. The regulation sets up some rights to users that providers should ensure when delivering their services. These include, among others, the right to be informed of the processing that users data might receive; the right to object to certain data processes (e.g. personalised marketing); or the right to be forgotten (erasure of any user data). While the GDPR is a great improvement on how user data must be managed, users still need to trust their service providers would comply with the GDPR rules along with any other applied regulation.

This situation could be improved by including the requirement of users having full control of their data in the design phase of an application development. A potential design to support this requirement involves storing users data into their personal devices instead of handing data over to service providers. These user-managed storage systems can be denoted as personal or decentralised data vaults [10]. Following such a design, data is always under user control, and any external entity requesting access to a data vault can be granted so directly by the vault owner, i.e., the final user.

In this paper, we present Vaultage, a model-based framework that can simplify the development of applications based on decentralised data vaults. Vaultage allows modeling both the data to be stored in a data vault, and the valid requests that a vault might receive. From a model containing this information, Vaultage generates a set of Java classes for the internal usage of the available data, and a secure communication infrastructure that can be used to interchange the specified requests and responses between the data vaults of a specific application.

We have tested Vaultage code generation capabilities by creating different data vault-based applications. Also, we have started a set of performance evaluation tests oriented to measure the performance of the different components of the generated architecture.

The rest of this paper is structured as follows. Section 2 presents the decentralised data vaults used in Vaultage. Section 3 introduces the running example that is used to explain the architecture and main features of Vaultage in Section 4. Section 5 discusses our evaluation efforts to assess Vaultage. Finally, Section 6 comments on related work, and Section 7 concludes the paper and outlines future work.

2 DECENTRALISED DATA VAULTS

A decentralised data vault[1] aims to store personal data in user-managed devices, thus giving these users more control over their data [10]. This is in contrast with common online services (e.g. cloud storages, social networks, search engines, etc.) manner of working, where users are required to trust the management of their personal data to a third party under often shady terms of service [9].

The data vaults promoted in this paper are composed of the following parts:

(1) **Data schema**. Our vaults have a predefined schema of the data they may store, which depends on the application domain. For instance, a medical app might store data about patients and their treatments. All data vaults of the same type share the same schema.
(2) **Data request operations**. Data vaults might receive external requests to access certain data contained in them, and then it is the responsibility of the vault owners/managers to appropriately respond to these requests, or to reject them. The explicit set of possible request operations that a concrete data vault type can expect has to be established. This set of request operations is similar to the REST API provided by a web service [12].

Therefore, any application wishing to include data vaults into its architecture must start by defining the two properties described above. While this inclusion can improve the privacy, security and users' control over any personal data used by the application [6, 7, 10], it does not come without challenges:

(1) Disruption on domain/business and analytical processes since data might not always be available (i.e. users can turn off their devices anytime or revoke permissions).
(2) Performance might be reduced since data are stored in personal devices which are generally less powerful than dedicated servers.
(3) The veracity of certain data items might need to be validated by requests external to any user-managed data vault, e.g., the reputation of a user in a second-hand online market, or simply the number of likes of a post in a social network.
(4) In terms of application development, the definition of a data vault must be followed by the implementation of an infrastructure to persist the defined data schema, and to enable communications to receive and respond to the set of possible requests. The fact that data is stored in a decentralised way can make network configuration more complex (e.g. routing or firewall aspects), and asynchronous/parallel processing of requests and responses is required (synchronisation, locking, racing, and timing problems).

This paper focuses on the last challenge, related to properly implementing an infrastructure to integrate data vaults into an application. This challenge is orthogonal to any application using data vaults, which creates an opportunity for code reuse and automatic generation. Starting from a data vault definition, the objective of the Vaultage framework is to automatically generate a set of software artefacts responsible for representing the data that can be stored in the vault (useful for the internal management of this data), and a communication network suitable for securely sending requests to and receiving responses from data vaults.

We use a running example throughout the paper to show how Vaultage supports developers during the integration of data vaults into an application. This example is presented in the next section.

3 RUNNING EXAMPLE: FAIRNET

A social network is a platform where users can create relationships between them based on their shared interests or opinions. In a very basic form, a social network allows users to create an online profile; link with other users by establishing friend or follower connections; and share information such as text posts, images, or videos. In traditional, centrally-managed systems, all data of such a social network would be stored in the service provider systems. Along this paper, we describe how to define Fairnet, a data vault-based social network application where data is owned by the social network users.

In Fairnet, each user would be the owner of a data vault, and would communicate with other users by sending requests to their respective vaults. We describe next the requirements of Fairnet: the data schema and the available requests that can be sent to a Fairnet vault.

3.1 Data Schema

A Fairnet data vault stores the following information of its user:

- As profile information, only the name of the user is stored. (RA1)
- A list of created posts. A post is composed of an id, a title, a text content, and a timestamp. Also, a post can be marked as public, which is a boolean value checked when receiving requests. (RA2)
- A list of accepted friends. We store the name of each friend. (RA3)

3.2 Data Vault Requests

There are three different requests that can be sent to a Fairnet vault:

- *addFriend*: send a friend request to another another Fairnet vault (i.e. to another user). This request includes information

[1] For simplicity, in the remaining of the paper we just refer to data vaults.

to identify the requester, and can be answered with a positive or negative response. (RB1)

- *getPosts*: ask for the list of post titles of a user. The title of a post is considered public in Fairnet, so this request is usually properly responded by the receiver. (RB2)
- *getPost*: ask for the data of a concrete post owned by the user. This user would send the requested data if the post is marked as public, or if the requester is a friend of the owner. In any other situation, the request would be rejected. (RB3)

In the following section, we describe how the data schema, possible requests of a vault such as the one informally presented above are modelled in Vaultage, and how, from the model of a data vault, different software artefacts can be generated.

4 VAULTAGE

Vaultage is a framework for simplifying the development of data vault applications, such as Fairnet. It achieves that by (1) providing core functionalities that are responsible for exchanging encrypted messages between peers, (2) generating application-specific strongly-typed wrappers of the core functionalities, and (3) generating skeleton code for application-specific functionalities. This way, developers can focus on developing the main functionalities of an application without having to worry about message/data exchange and encryption.

In Sections 3.1 and 3.2, we presented the two main requirements for Fairnet to be a vault-based social network. We started by describing the data schema of the vault, and then we defined the requests that it can accept. In the following sections, we discuss the different aspects – data vault representation, network architecture, encryption, and code generation – of Vaultage, and how it addresses these two requirements of the Fairnet application.

4.1 Data Vault Representation

The first task for developers when using Vaultage involves creating a model to define a data vault application. This model contains both the data schema and request operations accepted by the data vault (see Section 2). Instead of devising a new modelling language, Vaultage currently uses Ecore models to define data vaults. An Ecore model contains all the relevant aspects to describe the data schema of a vault (by defining different classes), as well as the set of requests (via operations). As an example, Listing 1 shows the model of Fairnet in the Emfatic notation[2], which has been specified based on the information of Sections 3.1 and 3.2.

In the model, the *Fairnetvault*, *Friend* and *Post* classes are defined (lines 4-13, 15-18 and 20-24). When defining a data vault, one of the classes of the model has to be defined as the *vault* class. To do so, one class has to be marked with the `@vault` annotation, indicating that the implementation of that class will be generated as a vault class (e.g., the `FairnetVault` class in Listing 1). This vault class is the one that identifies the data vault inside the code of the application (e.g. Fairnet in the example). The attributes defined in the vault class determine the contents stored in the data vault, i.e., the data schema. Other classes in the model are used as data containers and to provide domain types that might be used along the application. In the example, a Fairnet's data vault, according to the *FairnetVault* class, stores a name, a list of *Friends*, and a list of *Posts* (lines 4, 5 and 6, respectively). This definition matches the data schema requirements of Section 3.1 (RA1, RA2 and RA3).

Listing 1: Fairnet's model.

```
1  @GenModel(basePackage="org.vaultage.demo.
       fairnet")
2  package fairnet;
3
4  @vault
5  class FairnetVault {
6    attr String name;
7    val Friend[*] friends;
8    val Post[*] posts;
9
10   op Boolean addFriend(String friendName);
11   op Post getPost(String postId);
12   op String[*] getPosts();
13 }
14
15 class Friend {
16   attr String name;
17   attr String publicKey;
18 }
19
20 class Post {
21   attr String title;
22   attr String content;
23   attr String timestamp;
24   attr boolean isPublic;
25 }
```

In addition, the vault class also has to contain the requests that can be accessed by other vaults. These requests are defined as operations of the vault class. For example, to represent the available requests in Fairnet, i.e., *addFriend*, *getPosts*, and *getPost* (requirements RB1, RB2, and RB3 of Section 3.2), three operations with the same name are defined inside the *FairnetVault* class (lines 10, 11 and 12, respectively). The parameters of these operations represent data included in the requests, and the return value indicates the type of the response that should be provided for each request. For instance, an `addFriend` operation requires a `friendName` string to be provided as a way to identify the requester. The result of this operation is indicated with a boolean value, which will be `true` if the friend request is accepted and `false` if it is rejected.

4.2 Network Architecture

In its simplest form, a data vault application is composed of a set of users, and each one is the owner of a data vault. The interactions of a user within the application generate data requests and responses to be interchanged with data vaults of other users. So, one of the main requirements for the Vaultage framework is providing a secure communication mechanism for these data vaults, which we present in this section.

As data vaults are decentralised, we opted for using a relay communication system provided by a message broker. Data vaults

[2] https://www.eclipse.org/emfatic/

would subscribe to the broker using a public identifier. Then, it is possible to send requests to a concrete data vault by using their identifier when sending the request. There are several message broker applications available, such as Apache ActiveMQ[3] (the one currently in use by Vaultage), Kafka[4], or Mosquitto [8].

Figure 1 includes a diagram of the described network, using the Fairnet example. Each user is subscribed to an ActiveMQ queue, which is used by the message broker to deposit messages coming from other users[5]. The figure also shows an example of the messages that would be interchanged in Fairnet when a friend request is sent from one user to another. In the example, Alice sends a friend invite to Bob (step 1). This action in the Fairnet app is translated into an *addFriend* request message, including the appropriate parameters (step 2). In this case, the parameters are the destination of the message (i.e. the "Bob" queue / data vault), and the name of the user that sends the friend request ("Alice"). This request is relayed by the message broker into Bob's queue, which is received and translated into a Fairnet friend request in Bob's app (step 3). Then, in step 4, Bob accepts the friend invite from Alice. This acceptance is encoded as an *addFriend* response message, with "Alice" as the recipient of the response, and the `true` value to indicate that Bob has accepted the friend invite. As before, the message arrives at Alice's queue through the broker, and it is translated into a notification of Bob's accepting the initial request.

4.3 Encryption

All request and response messages sent in the architecture presented in the previous section are secured with asymmetric encryption [15]. The use of encryption is ingrained into the network configuration: data vaults subscribe to the message broker with their public key as their public identifier. For instance, in the example of Figure 1, "Alice" and "Bob" are the public identifiers used for communication. In a real context, Alice and Bob's public keys would be used.

When a message is sent to another data vault, a *double encryption* of the message is performed, in the following order:

(1) The message is encrypted using the private key of the sender. This allows knowing that a message comes for a certain data vault, which reduces the possibility of impersonating attacks.
(2) A new encryption is performed, using this time the receiver's public key. This is the standard encryption step that seeks that the contents of the message are only accessible by the receiver.

When a message is received, the inverse decryption is performed, i.e., by using the private key of the receiver first, and then the public key of the sender.

Currently, Vaultage uses Java's built-in RSA (key size 2048 bits) [11] as the algorithm for key-pair generation, and one of the RSA implementations[6] provided by Bouncy Castle[7] for ciphering. By applying the double encryption presented in this section, we satisfy the encrypted messaging challenge in Section 2 (challenge 4).

4.4 Code Generation

The Vaultage generator takes as input a data vault model in Ecore as described in Section 4.1. This generator has been implemented using the Epsilon Generation Language (EGL) [16], and the generation templates provide code in the Java language.

The generator provides classes for different concerns. In the following, we explain these concerns over the class diagram of Figure 2, which contains the generated classes for the Fairnet Example:

- **Vault**. The class that plays the role of the vault class contains the attributes defined in the data schema. In the figure, the vault class is `FairnetVault` (top right), which has a `name`, `post` and `friends` properties. The vault class also has one method for each available data vault request. These methods are called with the appropriate parameters when a message containing the associated request is received (e.g. an *addFriend* request would trigger a call to the *addFriend* method of the `FairnetVault` class). So, to determine how a request is handled, developers would only need to include some code in the associated method. An extra token parameter we have not discussed before appears in the signature of some methods. This token is used to link responses to their associated requests, as several requests might be received at the same time. A `FairnetVaultBase` parent class is also generated. This class contains properties and methods required for Vaultage to work (e.g. the vault's public and private keys), which must not be modified directly by the developers.
- **Remote Vault**. A class with the same name as the vault and the `Remote` prefix (e.g. `RemoteFairnetVault` in the top left of the figure) is created to provide developers with a high-level interface to send the requests and responses to other data vaults. This class requires the `remotePublicKey` that will receive the message, and the local vault to prepare the messages. For each request, a pair of methods is created: one to send requests, and another one to respond them. As an example, the `getPosts` method would be called to send a request to a remote vault, while the `respondToGetPosts` one would be executed by that remote vault to respond to the request.
- **Internal Entities**. These are the classes used in the definition of the data schema of a data vault model. In the example, the `Friend` and `Post` classes would be internal entity ones.
- **Response Handlers**. These interfaces (bottom left) must be implemented to provide the code that would be called when receiving a response to a previously sent request. There is one interface for each possible data vault request. For instance, the `AddFriendResponseHandler` `run` method would be called in step 6 of Figure 1, that is, when Alice receives Bob's response regarding her previous friend request.
- **Message Handlers**. These are auto-generated classes that manage the broker messages in the background. They are responsible for calling the appropriate Vault class method when a request is received, and of invoking the associated

[3] https://activemq.apache.org/
[4] https://kafka.apache.org/
[5] As a technical side note, we are also considering the use of ActiveMQ topics, which would be beneficial when including support for multiple user devices. https://activemq.apache.org/how-does-a-queue-compare-to-a-topic
[6] Precisely, the *RSA/ECB/OAEPWith-SHA256AndMGF1Padding* algorithm
[7] https://bouncycastle.org/

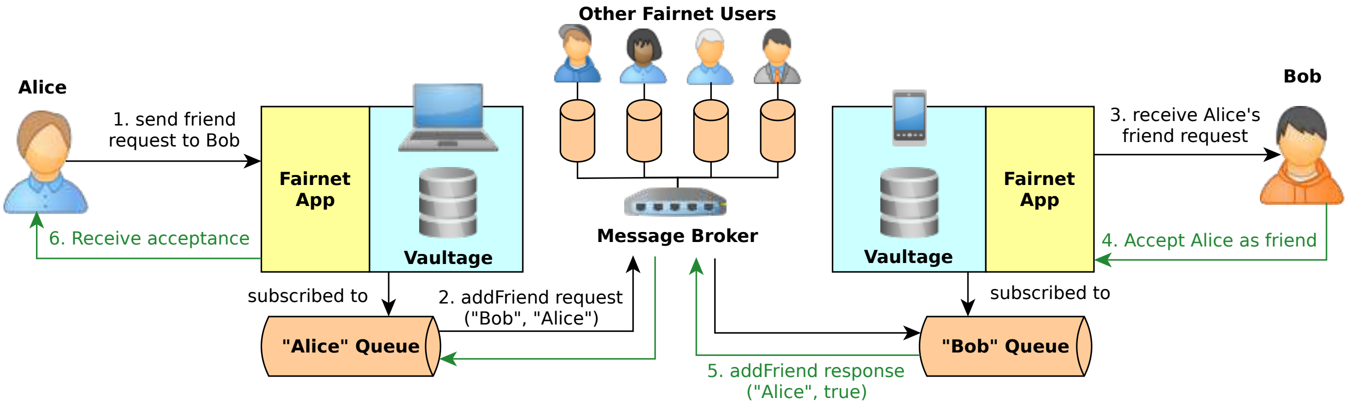
Other Fairnet Users
Alice
1. send friend request to Bob
6. Receive acceptance
Fairnet App
Vaultage
Message Broker
Vaultage
Fairnet App
Bob
3. receive Alice's friend request
4. Accept Alice as friend
subscribed to
2. addFriend request ("Bob", "Alice")
"Alice" Queue
subscribed to
"Bob" Queue
5. addFriend response ("Alice", true)

Figure 1: Network architecture provided by Vaultage

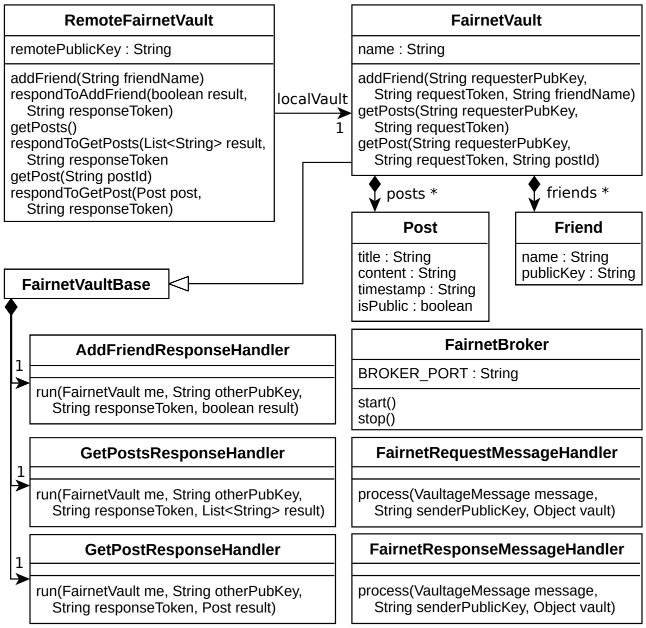
RemoteFairnetVault
remotePublicKey : String
addFriend(String friendName)
respondToAddFriend(boolean result, String responseToken)
getPosts()
respondToGetPosts(List<String> result, String responseToken
getPost(String postId)
respondToGetPost(Post post, String responseToken)
localVault
1
FairnetVault
name : String
addFriend(String requesterPubKey, String requestToken, String friendName)
getPosts(String requesterPubKey, String requestToken)
getPost(String requesterPubKey, String requestToken, String postId)
posts *
friends *
Post
title : String
content : String
timestamp : String
isPublic : boolean
Friend
name : String
publicKey : String
FairnetVaultBase
1
AddFriendResponseHandler
run(FairnetVault me, String otherPubKey, String responseToken, boolean result)
1
GetPostsResponseHandler
run(FairnetVault me, String otherPubKey, String responseToken, List<String> result)
1
GetPostResponseHandler
run(FairnetVault me, String otherPubKey, String responseToken, Post result)
FairnetBroker
BROKER_PORT : String
start()
stop()
FairnetRequestMessageHandler
process(VaultageMessage message, String senderPublicKey, Object vault)
FairnetResponseMessageHandler
process(VaultageMessage message, String senderPublicKey, Object vault)

Figure 2: Class diagram of the classes that are automatically generated by Vaultage for the Fairnet example.

response handler when a response message of a previous request arrives.

- **Broker Server Launcher**. This class is responsible for launching the Apache ActiveMQ broker server. It allows certain modifications, such as configuring the port in which the broker server is started.

5 EVALUATION

We describe here our ongoing efforts for the evaluation of the Vaultage framework. We start by presenting the different data vault applications generated during the development of Vaultage. Then, we include some initial results of the performance-related experiments we are currently carrying out.

Table 1: Vaultage model length (in lines of code) against the generated code.

Application	Model Lines	Generated Code Lines	Ratio
Fairnet	23	371	1:16
Pollen	20	332	1:17
Synthesiser	7	202	1:29

5.1 Data Vault Applications Generation

We have created three minimal data vault applications – Fairnet, Pollen, and Synthesiser – to evaluate the code generation capabilities of Vaultage. All these applications can be found online in Vaultage's open source repository[8]. Each application has a specific objective that makes them different to the others. We also measured the degree of automation that Vaultage provides by calculating the ratio between the number of lines in the input model (LM) and the number of lines of generated code (LG) for each application. Table 1 shows the obtained ratios, as well as the absolute line counts. Comments and empty lines are excluded from these counts. We expect these ratios to further improve as Vaultage matures and extra functionality is covered by the automatic generators.

5.1.1 Fairnet. This application, which was the initial example used to develop Vaultage, was already introduced in Section 3. Although simple, Fairnet is versatile enough to offer different concerns for the code generations to care about, such as friendship relationships between vaults, a mixture of public and private data (e.g. post title and contents, respectively), and requests that can be automatically answered (*getPost*, *getPosts*) including also others that require user input (*addFriend*). This versatility is also useful for presenting Vaultage, which made us select Fairnet as the running example for this paper.

5.1.2 Pollen. This application makes use of Multi-Party Computation techniques (MPC) [5] to perform polls securely. The application of MPC aims to prevent any participant of the poll to know the answer of any other participant. For instance, by applying MPC to

[8] https://github.com/York-and-Maastricht-Data-Science-Group/vaultage

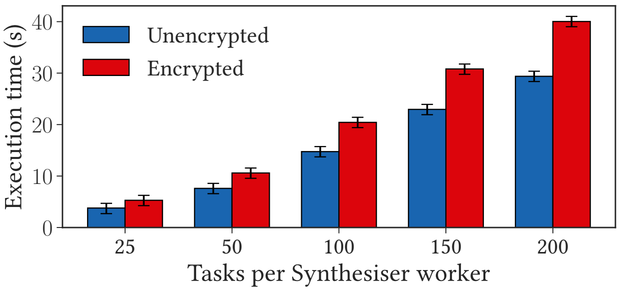

Figure 3: Time required to complete Synthesiser runs of increasing work size, using unencrypted/encrypted messages.

perform polls, it is possible to ask a set of participants some question (e.g. a 1-to-10 rating about a government decision, a workplace policy, or a teacher's performance), while ensuring that any person participating in the poll cannot know the response of any other participant, and that the originator of the poll is also only able to see the final aggregated responses of all participants. Therefore, the main benefit provided by Pollen is a non-trivial communication problem that has been very useful for polishing Vautage's auto-generated message handling architecture, so that it becomes easy to use for data vault application developers.

5.1.3 Synthesiser. This is an internal performance testing tool that allows evaluating the architecture generated by Vaultage. In this application, vaults store no data, and can only respond to an *increment* request. This request provides a number as a parameter, and it is always responded with the following number (i.e. it adds one). Each node in Synthesiser is denoted as a worker. When created, a worker receives a number of tasks to perform, each of these consisting in sending an increment request to another worker of the network. A worker finishes its work when all tasks have been completed, i.e., when it has sent and received back the provided number of tasks to complete. Given a network configuration that includes the traffic pattern for workers to distribute tasks among them, this application can be used to measure how much time is required to complete a certain number of tasks per worker. In addition, by using the same Synthesiser network configuration we can compare the performance of other system aspects, e.g., different encryption mechanisms, ways of handling request or response messages, or how data is retrieved and stored in a vault.

5.2 Vaultage Performance

We are in the process of stress-testing the auto-generated network architecture, as well as profiling the time spent in the different aspects of the communication process, such as encryption or message handling.

Related to profiling Vaultage communication, Figure 3 shows some preliminary results on the weight that encrypting transmitted messages has in the total time required for a Synthesiser run to complete. We ran these tests in isolation using a desktop computer with an Intel i5-6400 4-core/4-thread CPU, with 24GiB RAM running at 3200MHz, and a SATA SSD drive. In the measurements, we used three Synthesiser workers, leaving an extra core for running the associated ActiveMQ broker. We performed several Synthesiser runs of increasing work size, ranging from 25 tasks (i.e. requests sent) per worker to up to 200. To ensure our measurements were trustable, each Synthesiser run was run several times per work size, and then 95% confidence intervals for the times of each work size were calculated and depicted in the bar chart of Figure 3. Based on the values of these intervals, the measurements were stable across all work sizes. The results show that, on average, encryption/decryption of messages imposes a 37% penalty over the time required to complete a Synthesiser run without message encryption. This kind of tests could also be useful to compare different encryption approaches, e.g., measuring the cost of using greater RSA encryption key lengths.

Although we consider performing the previous tests locally is a good way to measure the encryption penalty on the transmission without being affected by the reliability of a network, we are also interested in running them in a more realistic scenario, i.e., using distributed nodes in a controlled network, or in a cloud service such as AWS or Google Cloud. Such network configurations would also be more adequate to test other issues, such as increasing the number of on-the-fly messages to stress-test the central broker, as well as the data vault nodes. We would also like to compare the effect of relaying many long messages (e.g. transmitting media content such as images or video) through the broker with establishing point-to-point communications between nodes that want to share heavy amounts of data.

6 RELATED WORK

One of the first approaches for online service providers to lend control of users data is to store it encrypted, in such a way that these providers cannot decrypt the data themselves. The number of applications following this approach is increasing, including instant messaging platforms such as Signal[9] or email service providers like Tutanota[10]. Moreover, some advanced cryptographic techniques allow doing some privacy-preserving work over encrypted data, without requiring or knowing how to decrypt it. For instance, Homomorphic Encryption [14] can be used to perform some data analysis processes over user-encrypted data without the need to know the encryption key, which might help maintain user data privacy. Solutions based in this encryption technique have been applied, among others, to recommender systems [3] and medical data [19]. Still, and despite the encryption, some users might be reluctant to lend service providers the control of their data, in which case any approach allowing users to store data in self-managed systems would be a better option.

Providing users with full control of which personal data they want to share with a third party is a service currently offered by several applications. A good example of these is Solid[11], which is based on the storage of users data in personally-managed *pods* (somewhat equivalent to Vaultage's vaults). Any third-party application made interoperable with Solid can request access to pods, and the only ones who can grant this access are the pods owners (i.e. the users). Solid does not impose any restrictions for the location

[9] https://www.signal.org/
[10] https://tutanota.com/
[11] https://inrupt.com/solid

where pods are stored, so it allows avoiding centralised backends and opens the possibility for users to store their data locally. We plan to study potential benefits of supporting some Solid components in Vaultage, such as its user authentication infrastructure. Applications following similar approaches are Digi.me[12], CozyCloud[13], or CloudLocker[14], among others.

Related to communication aspects, there are several model-driven approaches that aim to ease the definition of network configurations. These approaches focus mostly on information flow and access control [1, 4, 13], which are general concerns of any kind of network infrastructure. However, less efforts have been put into the automatic generation of secure network communication capabilities provided by Vaultage. We have only found one application aiming to generate this kind of communication-related code, in the context of Internet of Things systems. These systems are composed of (generally) low-power sensor and actuator devices that interchange data in a distributed network to provide some functionalities, such as controlling the air-conditioning system of a smart home installation. The CyprIoT framework [2] allows to define the communication of these systems by means of two domain-specific languages for the specification of the network configuration and the network policies that must be enforced, respectively. From these specifications, a model-to-text transformation step can be performed to generate the network code to deploy in the IoT devices of the system, freeing engineers from dealing with some low-level details. While Vaultage does not offer a way to specify fine-grained network constraints, it is a general-purpose framework that also supports solid encryption mechanisms for the communication between more complex nodes than the usual IoT devices. We will analyse if enabling some network policy configurations in Vaultage could be useful for some objectives, such as preventing malicious behaviours (e.g. denegation of service attacks).

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented Vaultage, a framework that is intended to support developers when creating data vault applications. Vaultage offers automatic code generation of decentralised data vault networks, including brokered messaging between vaults, and securing messages through double encryption mechanisms. Vaultage has been used to generate three different data vault applications: Fairnet (a social network), Pollen (a polling/survey application), and Synthesiser (for network performance testing).

For our future work, we plan to add more features such as direct messaging and synchronisation between user devices. Direct messaging will improve the efficiency of data exchange when two vaults reside in the same network, or when we want to avoid overwhelming the message broker relay capacity with heavy communications (e.g. media interchange). Synchronisation between devices will enable users to maintain copies of their vaults over multiple devices for improved availability and fault-tolerance.

ACKNOWLEDGMENTS

This work has been funded through the York-Maastricht partnership's Responsible Data Science by Design programme (https://www.york.ac.uk/maastricht/).

REFERENCES

[1] David A. Basin, Manuel Clavel, and Marina Egea. 2011. A decade of model-driven security. In *16th ACM Symposium on Access Control Models and Technologies, SACMAT 2011, Innsbruck, Austria, June 15-17, 2011, Proceedings*, Ruth Breu, Jason Crampton, and Jorge Lobo (Eds.). ACM, 1–10. https://doi.org/10.1145/1998441.1998443

[2] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. 2019. CyprIoT: framework for modelling and controlling network-based IoT applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*, Chih-Cheng Hung and George A. Papadopoulos (Eds.). ACM, 832–841. https://doi.org/10.1145/3297280.3297362

[3] Zekeriya Erkin, Thijs Veugen, Tomas Toft, and Reginald L. Lagendijk. 2012. Generating Private Recommendations Efficiently Using Homomorphic Encryption and Data Packing. *IEEE Trans. Information Forensics and Security* 7, 3 (2012), 1053–1066. https://doi.org/10.1109/TIFS.2012.2190726

[4] Christopher Gerking and David Schubert. 2019. Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures. In *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*. IEEE, 61–70. https://doi.org/10.1109/ICSA.2019.00015

[5] Oded Goldreich. 2002. Secure Multi-Party Computation (Final (incomplete) Draft, Version 1.4). Retrieved July 14, 2020 from http://www.wisdom.weizmann.ac.il/~oded/PSX/prot.pdf.

[6] Paul Henman and Mitchell Dean. 2004. The governmental powers of welfare e-administration. In *Australian Electronic Governance Conference 2004*. Melbourne, Australia: Department of Political Science, University of Melbourne.

[7] Riad Ladjel, Nicolas Anciaux, Philippe Pucheral, and Guillaume Scerri. 2019. A manifest-based framework for organizing the management of personal data at the edge of the network. In *ISD 2019 - 28th International Conference on Information Systems Development*. Toulon, France. https://hal.archives-ouvertes.fr/hal-02269203

[8] Roger A. Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *J. Open Source Softw.* 2, 13 (2017), 265. https://doi.org/10.21105/joss.00265

[9] Ewa Luger, Stuart Moran, and Tom Rodden. 2013. Consent for all: revealing the hidden complexity of terms and conditions. In *2013 ACM SIGCHI Conference on Human Factors in Computing Systems, CHI '13, Paris, France, April 27 - May 2, 2013*, Wendy E. Mackay, Stephen A. Brewster, and Susanne Bødker (Eds.). ACM, 2687–2696. https://doi.org/10.1145/2470654.2481371

[10] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. 2010. Personal Data Vaults: A Locus of Control for Personal Data Streams. In *Proceedings of the 6th International COnference (Co-NEXT '10)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/1921168.1921191

[11] Oracle. 2020. Java Cryptography Architecture Standard Algorithm Name Documentation for JDK 8. Retrieved July 12, 2020 from https ://docs.oracle.com/javase/8/docs/technotes/guides/security/ Standard-Names.html#KeyPairGenerator.

[12] Cesare Pautasso. 2014. *RESTful Web Services: Principles, Patterns, Emerging Technologies*. Springer New York, New York, NY, 31–51. https://doi.org/10.1007/978-1-4614-7518-7_2

[13] Salvador Martínez Perez, Joaquín García-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, and Jordi Cabot. 2013. Model-Driven Extraction and Analysis of Network Security Policies. In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings (Lecture Notes in Computer Science)*, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke (Eds.), Vol. 8107. Springer, 52–68. https://doi.org/10.1007/978-3-642-41533-3_4

[14] Manish M. Potey, C.A. Dhote, and Deepak H. Sharma. 2016. Homomorphic Encryption for Security of Cloud Data. *Procedia Computer Science* 79 (2016), 175 – 181. https://doi.org/10.1016/j.procs.2016.03.023 Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[15] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342

[16] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. 2008. The Epsilon Generation Language. In *Model Driven Architecture – Foundations and Applications*, Ina Schieferdecker and Alan Hartman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[12] https://digi.me/
[13] https://cozy.io/en/
[14] https://www.cloudlocker.eu/en/index.html

[17] Nili Steinfeld. 2016. “I agree to the terms and conditions”: (How) do users read privacy policies online? An eye-tracking experiment. *Computers in Human Behavior* 55 (2016), 992 – 1000. https://doi.org/10.1016/j.chb.2015.09.038

[18] European Union. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* 59 (May 2016). https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC

[19] Alexander Wood, Kayvan Najarian, and Delaram Kahrobaei. [n.d.]. Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics. *ACM Comput. Surv.* 0, ja ([n. d.]), 1. https://doi.org/10.1145/3394658