JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Vaultage: Automatic Generation of Secure Communication around Decentralised User-Managed Data Vaults

Alfa Yohannis*, Alfonso de la Vega†, and Dimitris Kolovos*
*University of York, United Kingdom
†University of Cantabria, Spain

ABSTRACT The storage of user data in centralised systems is a standard procedure followed by online service providers such as social networks. This requires users to trust these providers, and, to some extent, users are not in complete control over their data. A potential way to bring back this control is the promotion of user-managed data vaults, i.e., encrypted storage systems located in personal devices. Enabling communication between these data vaults could allow creating decentralised applications where users decide which data to share, and with whom. Nevertheless, developing such decentralised applications requires a considerable amount of work, as well as expertise in deploying secure peer-to-peer communication systems. We present Vaultage, a model-based framework that can simplify the development of data vault-based applications by automatically generating a secure communication infrastructure from a domain-specific model. We demonstrate the core features of Vaultage through a decentralised social network application case study, and we report on the findings of evaluation experiments that show Vaultage's code generation capabilities and some performance analysis of the generated network components.

KEYWORDS Data Privacy, Decentralised Data, Model-Driven Engineering, Generative Programming.

1. Introduction

A nearly ubiquitous prerequisite for users of contemporary online services is to grant service providers the right to manage and process their data. As a consequence, user data typically ends up stored in remote servers, where they have limited control of how it is processed or monetised. Most service providers use Terms of Service (ToS) to define how users' data will be processed 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 (Luger et al. 2013). As a consequence, users tend to ignore the terms and immediately jump to use the provided services (Steinfeld 2016).

There are regulations such as the European General Data Protection Regulation (GDPR) that establish a set of rules by which service providers must abide the processing, movement, and protection of personal data (European Union 2016). These regulations also provide several rights to the end-users, such as the right to be informed of the processing their data might receive, the right to object to specific data processes (e.g., personalised marketing), or the right to be forgotten (erasure of any user data). While the GDPR is a significant improvement in data privacy and protection, users still need to trust their service providers to comply with this and other relevant regulations. In this case, users' data is only protected by law (it requires legal enforcement) but not by the design of the technology (prevention). A breach in service providers' systems, which commonly are centralised, could lead to data misuse impacting millions of users (Cadwalladr & Graham-Harrison 2018; Krebs 2013; Gunaratna 2016; Stempel & Finkle 2017).

An alternative to data management centralised in the service providers' systems is to store users' data in personally-managed data containers that live in their own devices. Having data stored

JOT reference format:
Alfa Yohannis, Alfonso de la Vega, and Dimitris Kolovos. *Vaultage: Automatic Generation:of Secure Communication :around Decentralised User-Managed Data Vaults*. Journal of Object Technology. Vol. 21, No. 3, 2022. http://dx.doi.org/10.5381/jot.2022.21.3.a9

in personal devices grants users greater access control, and a clearer and more reliable way to remove any data item if they desire to do so. This kind of containers have been previously denoted in different ways, such as *data pods* in the case of the Solid framework (Solid 2017), or *data vaults* as in (M. Mun et al. 2010), which is the term we also use in this work.

On the other hand, such data vaults have to deal with several challenges, such as performance issues, or the complexity of developing trustworthy secure decentralised applications. Related to performance, nowadays personal devices (e.g., smartphones, laptops) come with enough computing power to host simple applications. Therefore, reusing these devices as the machines to run data vaults is reasonable, as seen in participatory sensing applications (Christin et al. 2011). Nevertheless, if implementing privacy-preserving services in centralised systems is already a complex endeavour (Senarath & Arachchilage 2018), developing decentralised applications is even more complex, since such applications have to deal with data distributed across data vaults, data access authorisation, and secure communication between multiple peers. Data unavailability and loss are likely to occur if devices are offline or lost/damaged. Addressing this complexity might increase the development costs of decentralised applications.

We present Vaultage, a model-based framework for the development of decentralised applications around user-managed data vaults. Vaultage allows modelling both the data to be stored in a data vault and the set of data services that external clients can request from a vault. From a model containing this information, Vaultage generates (1) a set of Java classes for the application's internal usage of the available data, and (2) a secure communication infrastructure that can be used to facilitate requests and responses between clients and data vaults of a specific application.

The core architecture of Vaultage was introduced in a previous workshop paper (Yohannis et al. 2020). As an extension of that paper, this work includes:

- An in-depth discussion of the motivation behind the development of Vaultage in Section 2.
- An improved and up-to-date description of Vaultage's features in Section 3, including new ones such as the direct communication modes that do not require using a central broker to relay messages between nodes, or the possibility to transfer other types of data (e.g., media files).
- An extended evaluation – in Section 5 – to assess Vaultage's expressiveness and code generation capabilities by creating several data vault-based applications. We have also carried out performance evaluation experiments to measure the performance implications of different communication aspects, such as encryption and the used communication mode.
- A discussion of the limitations and challenges of the proposed approach in Section 4.

Finally, Section 6 presents other related works and Section 7 comments on future lines of work and concludes this paper.

2. Background and Motivation

2.1. Data Privacy in Centralised Online Systems

Letting service providers have the responsibility of storing any related data to that service is a comfortable choice for the end-user, as they become free from the burden of its management and preservation. However, there are different privacy implications of that transfer of responsibility. For instance, as stated previously, the ways in which user data is processed and monetised by the service provider may be obfuscated behind difficult to understand terms of service (ToS) (Luger et al. 2013; Steinfeld 2016), to the point of inducing the creation of automated tools for understanding ToS assisted by machine learning and natural language processing techniques (Wilson et al. 2019). Also, even having a clear understanding of the ToS and regulations such as the GDPR (European Union 2016) that apply to certain data, there have been past examples of irresponsible uses of captured data (Cadwalladr & Graham-Harrison 2018).

Although there are several guidelines to develop applications featuring privacy by design (Cavoukian et al. 2009; European Network and Information Security Agency 2014), the implementation of such guidelines can be complex or in some cases directly overlooked because of, for example, developers poor understanding of privacy-oriented development principles, or because of companies lacking the enforcement of such principles (Senarath & Arachchilage 2018).

Another issue of storing sensitive data of thousands or millions of users in centralised systems is that such systems become lucrative targets for malicious attacks. When these attacks are successful (Krebs 2013; Wakefield 2014; Gunaratna 2016; Stempel & Finkle 2017), users can be affected by data breaches in which their personal information can be publicly exposed. As an extra problem users might face, there have been examples of companies not being clear enough when explaining the implications or the extent of data breaches affecting their users' data (Zou et al. 2019).

An approach that centralised service providers can follow to give control of this data is to store it in an encrypted form so that even the providers themselves cannot decrypt the data. The number of applications following this approach is increasing, including instant messaging platforms such as Signal[1] or email service providers like Tutanota[2]. Moreover, some advanced cryptographic techniques allow performing some privacy-preserving work over encrypted data without requiring or knowing how to decrypt it. For instance, homomorphic encryption (Potey et al. 2016) can be used to perform some data analysis processes over user-encrypted data without the need to know the encryption key, which can help maintain user data privacy. Solutions based on this encryption technique have been applied, among others, to recommender systems (Erkin et al. 2012) and medical data (Wood et al. 2020). Still, and despite the encryption, some users might be reluctant to pass service providers control of their data, in which case any approach allowing users to store data in self-managed systems would be a viable alternative.

[1] https://www.signal.org/
[2] https://tutanota.com/

2.2. Using Personal Devices in Decentralised Applications

The increase in features and computing power of personal devices (e.g., smartphones, tablets, laptop/desktop computers) has caused a surge of decentralised applications and services running in these devices during the last years. A clear example of this surge are participatory sensing applications (Christin et al. 2011), which use sensors found nowadays in current smartphones such as accelerometers, microphones, or location ones to capture information of interest to create data services. The capability of these devices can also be augmented by connecting external sensors, to provide extra measurements such as temperature or air quality (Sivaraman et al. 2013).

The main benefit of installing applications into already existing devices are the reduced costs when opposed to having to deploy and maintain physical infrastructures in place. Moreover, the constant motion of some personal devices such as smartphones can allow for extra use cases with respect to static infrastructures, such as being able to detect traffic jams or road bumps by the location and accelerometer sensors of a smartphone installed in a vehicle (Mohan et al. 2008).

On the other hand, capturing sensitive data such as location or voice recordings carries considerable privacy implications (Christin et al. 2011). Generally, participatory sensing applications use personal devices to obtain the relevant data, but then this data is sent to centralised servers where it is processed and put up to use. Consequently, thorough controls must be in place to check what data is captured and what local data processes are applied before being sent to a remote server (e.g., anonymisation, filtering, obfuscation). For instance, in (Maisonneuve et al. 2009) personal devices are used to measure ambient noise levels in different areas of a city. Instead of sending raw sound recordings for processing (which e.g., may contain personal conversations), sound intensity levels are calculated locally and sent instead. Ideally, the owner of a capturing device should be able to monitor and participate in this data control, which should be made as transparent and understandable as possible.

2.3. User-Managed Data Vaults

(M. Mun et al. 2010) proposed the concept of *personal data vaults* in the context of participatory sensing applications. These data vaults are intermediate containers where the captured information is stored prior to being shared with service providers. These containers are managed by the users, so they retain data ownership and can have a fine-grain control of which data is finally shared. The same authors later described an example of how to use personal data vaults in an application capturing and sharing location data (M. Y. Mun et al. 2014).

The objective of Vaultage is aligned with the work in (M. Mun et al. 2010), as it promotes the storage of data in user-managed data vaults, thus achieving true application decentralisation. These data vaults can be queried by external entities (e.g., service providers in a participatory sensing application), and the vault owners control whether and how these queries are answered. In the context of Vaultage, a data vault is composed of the following two parts:

1. **Data Schema**. Data vaults have a predefined schema of the data they can store, which depends on the application domain. For instance, a data vault-based social network app might store data about user posts and friends.
2. **Data Services**. Vaults provide data services that can be requested to access (or, in some cases, to provide) certain data. Continuing with the social network example, sending a friend request, or asking for the latest published posts could be valid data services. The set of services provided by a data vault is similar to the REST API provided by a web service (Pautasso 2014).

Therefore, any application wishing to include data vaults into its architecture must implement an infrastructure to persist the defined data schema and to enable communications to receive and respond to supported requests. The fact that data is stored in a decentralised way can make network configuration more complex (e.g., routing or firewall aspects). Also, asynchronous/parallel processing of requests and responses are required, which might cause synchronisation, locking, racing, and timing problems. While data vaults can in principle be applied to any application domain, commonalities of the infrastructure for data management and communication create an opportunity for code reuse and automatic generation (Brambilla et al. 2012), which is leveraged by Vaultage.

2.4. Running Example: Fairnet

We present Vaultage using as example a data-vault based social network, named Fairnet. In Fairnet, each user is the owner of a data vault, and they can communicate with other users by sending requests to their respective vaults. We describe next the two features that define a data vault as described in the previous section: its data schema and the data services that can be requested.

2.4.1. Data Schema

A Fairnet vault stores the following information of its user:

- As profile information, only the name of the user is stored.
- A list of accepted friends, of which we store their name. We also store their public identifier for communication in the decentralised network generated by Vaultage. As Section 3.1.1 explains later in the paper, this identifier is the public key of a key pair used for asymmetric encryption.
- A list of created posts. A post is composed of an id, a title, a text content, a timestamp, and a list of the identifiers of the files attached to the post. Posts are private by default, but they can be marked as public. A private post can only be accessed by the owner of the vault, while public posts can also be accessed by their friends.
- A list of files that are attached to a post. Each file has an identifier, a name and its byte contents.

2.4.2. Data Services

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

- addFriend(friendName: String): Boolean: ask for a confirmation to become a friend of another user. This request

includes the name that identifies the requester and can be answered with a positive or negative response. If the friend request is confirmed, the requester will be set as a trusted requester, and therefore, any following requests will be accepted.
- getPosts(): Post[*]: ask for the list of posts of a user[3]. Each post in the returned list contains the identifier and title fields, but not its content or the list of files that are attached.
- getPost(postId: String): Post: ask for the content of a specific post owned by the user. This vault will send the requested post with all its fields if the post is marked as public and the requester is a friend of the owner, otherwise, it will reject the request.
- getFile(fileId: String): File: ask for a file, which can be an image, video, audio, document, or other file types, attached by a user to one of their posts.

3. Vaultage

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

We present first the different aspects of the secure network architecture generated by Vaultage for each concrete data vault-based application. Then, we show how data vaults can be represented as models, which are used as input of a code generation process to obtain the presented architecture.

3.1. Network Architecture

Vaultage provides a secure communication infrastructure for clients to send requests to the data services of a vault. This communication is similar to how client-server REST architectures work (Pautasso 2014).

The following sections describe how messages are secured by encryption, and the two possible modes of communication between a client and a vault: brokered and direct messaging.

3.1.1. Encryption Decentralisation makes encryption a key aspect to secure communications. All requests to the data services of a vault and the associated response messages are secured with asymmetric encryption (Rivest et al. 1978), so every node that is involved in some sort of communication with the system is expected to own both a public and a private key. When a message is sent to a 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 verifying that a message comes from a legitimate source and shields the system against impersonation attacks.
2. A second encryption is performed on top of the first one, using this time the receiver's public key. This is to ensure that the contents of the message are only accessible by the receiver.

When a message is received, inverse decryption is performed, i.e., by using the private key of the receiver first and then the sender's public key. This double encryption mechanism ensures the authenticity of the sender and that only the expected receiver should be able to access the message contents. Currently, Vaultage uses RSA 2048-bit key-pair generation and ciphering provided by Bouncy Castle[4].

3.1.2. Brokered Messaging As data vaults are decentralised, we opted for using a relay communication system provided by a message broker, since users' devices do not usually have stable IP/DNS addresses where they can be reached by their peers and they may even be behind firewalls that prevent direct incoming connections. There are several message broker applications available, such as Apache ActiveMQ[5] (the one currently in use by Vaultage), Kafka[6], and Mosquitto (Light 2017).

In the case of ActiveMQ, any data vault wanting to receive messages has to subscribe with a representative identifier at the broker server, so that every message addressed to that identifier finally reaches the data vault. Vaultage ingrains into the network configuration the use of asymmetric encryption described in the previous section by using the public key of each node as its identifier at the broker server. Thus, every time a peer wants to send a message to a data vault, it only needs to send the double-encrypted message to the broker server using the vault's public key as destination.

Figure 1 shows how the brokered communication works using the Fairnet example. Each Fairnet user subscribed to an ActiveMQ broker as described in the previous paragraph, and it has access to a message queue, which the broker uses to deposit messages coming from other users. The figure depicts a request example where a user (Alice) is sending a friend request to another user (Bob). The request originates from Alice sending a friend invite to Bob through the Fairnet app (step 1). This action 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 user's name that sends the friend request ("Alice"). The message broker relays this request 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` boolean value to indicate that Bob has accepted the friend invite (step 5). As before, the message arrives at Alice's queue through the broker, and it is translated into a notification of Bob's acceptance of the friend request (step 6).

[3] For simplicity we don't consider paging results in this minimal example.

[4] https://bouncycastle.org/
[5] https://activemq.apache.org/
[6] https://kafka.apache.org/

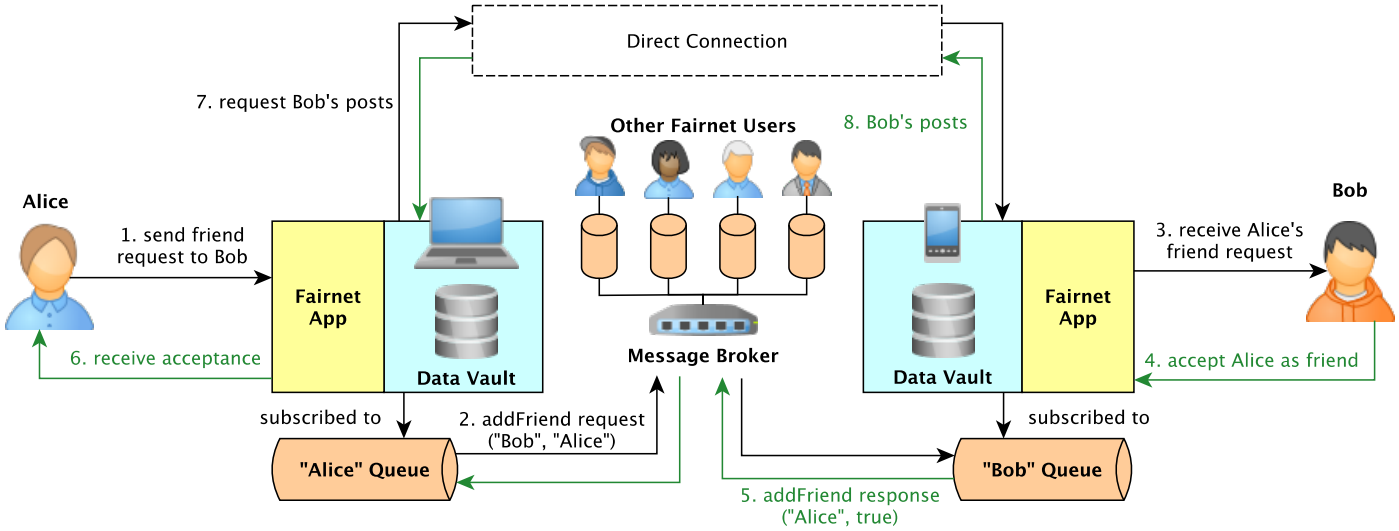

Figure 1 Brokered network architecture provided by Vaultage

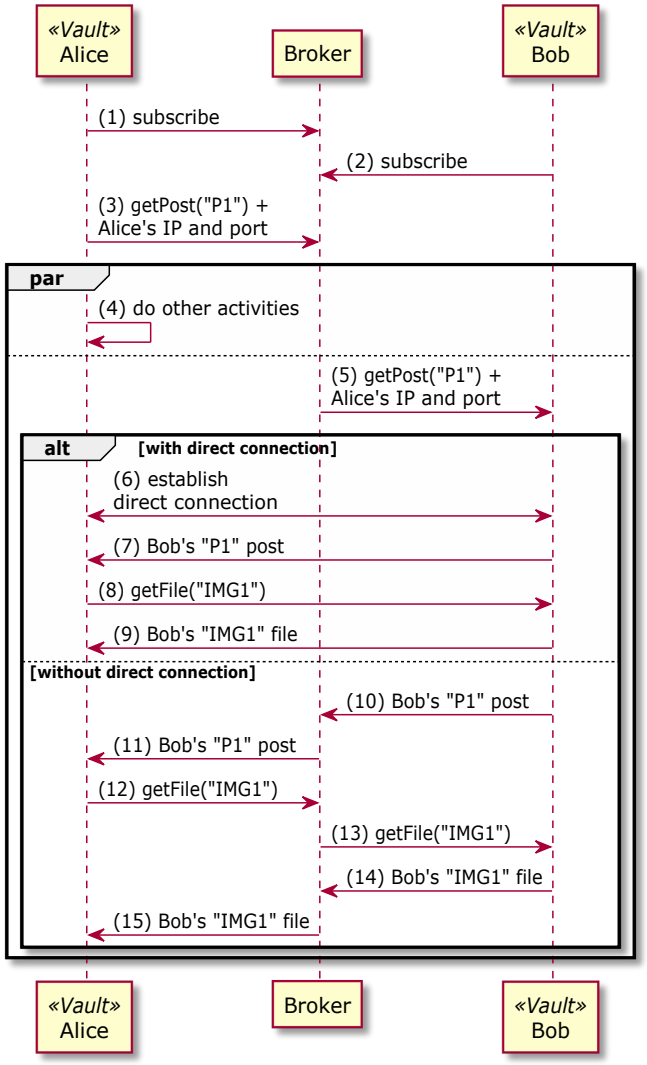

Figure 2 Direct messaging mechanism in Vaultage

3.1.3. Direct Messaging Vaultage also supports direct messaging for those cases where establishing a direct, broker-less connection between two nodes is possible. This mode of messaging is faster and more efficient, as there is no need to use an intermediate broker as a relay. For instance, in Figure 1, if Alice and Bob interchanged their connection details to allow direct communication, any further requests could be sent directly between their devices, such as the getPosts request that is depicted in steps 7 and 8 at the top of the figure. Direct messaging can also reduce the broker server's load, especially when the communication involves exchanging larger assets such as videos, audio, and images.

Vaultage makes direct communication possible by enclosing the IP and port of a node supporting this mode of communication in every message sent. This way, the receiving end could know the sender's address and then initiate a direct TCP/IP connection to send any subsequent messages.

Figure 2 depicts a scenario where Alice wants to access the post with id *P1* from Bob, as well as a file *IMG1* attached to that post. Assuming that Alice and Bob are already subscribed to the broker (steps 1 and 2), and that they are friends already, Alice starts by sending a request to get Bob's *P1* post through the broker (step 3). In addition to the request, the message also contains Alice's connection details – IP address and TCP port – that can be used to enable direct messaging. Due to the asynchronous nature of Vaultage, Alice could do other operations after sending the request (step 4) such as listening to any other incoming requests and responses, whether via a broker or directly from other requesters. After receiving Alice's request (step 5), Bob's vault uses Alice's network details to establish a direct connection (step 6), and then use that direct connection to send Alice the requested *P1* post (step 7). Using this connection facilitates the posterior interchange of the *IMG1* file through a getFile request that only involves Alice and Bob vaults (steps 8 and 9). If a direct connection cannot be established, Bob's Fairnet app automatically uses the broker to send back his *P1* post instead (steps 10 and 11), and it does the same for the

posterior file request (steps 12-15).

3.2. 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 services provided by the data vault (see Section 2.3) and conforms to the Ecore (Steinberg et al. 2008) metamodel presented in Figure 3. Note that the metamodel contains similar concepts to the ones found in UML class diagrams. Thus, we could have decided to reuse UML, including any specific terms such as vaults in a UML profile. Opting for the definition of a new metamodel is only an implementation choice based on our previous expertise and personal preference.

A Vaultage model contains one or more Vaults. A vault is composed of Features, which define the data schema; and Services, which determine the data services offered by each vault (see Section 2.3). Features can be either Attributes having a primitive type (defined as DataType following Ecore's denotation) or References, whose type is an Entity. An entity is a class-like composite type, which has a name and a set of features. The services of a vault have a name, a set of Parameters, and a return type. Parameters also have a type, and both parameters and services can use as type either an entity or a data type. This is managed through the Classifier term that again originates from Ecore terminology. Features, services and parameters can be multi-valued, which is managed with their *many* attributes.

As an example, Figure 4 shows the Vaultage model of Fairnet, which conforms to the metamodel of Figure 3. To limit the verbosity of representing such a model, we have used a mixture of a class and an object diagram: vaults and entity instances are represented as classes, including their original type (e.g., FairnetVault is a Vault, Post is an Entity). Features are represented as attributes for both vaults and entities, and vault services are represented as class operations. In the figure, a FairnetVault is a vault containing a name, and lists of Friends, Posts and Files, which compose the data schema of Fairnet. This vault has four operations, one for each data service defined in Section 2.4.

3.3. Code Generation

The Vaultage generator has been implemented using the Epsilon Generation Language (EGL) (Rose et al. 2008), and the generation templates produce code in the Java language. The generator takes as input a data vault model as described in Section 3.2, and produces classes for different concerns. The code generator takes care of providing all the scaffolding code required to manage the secure messaging system. This way, developers only need to care about specifying how vault requests and responses will be handled, while not having to control how these requests and responses reach their target nodes.

Figure 5 shows a class diagram containing the main classes generated from the Fairnet model of Figure 4. The classes of this diagram have been organised in three groups: (a) relevant classes from the Vaultage framework that have been manually programmed and provide general functionality such as message encryption or broker and direct messaging communication capabilities; (b) classes specific to the developed application that have been partially generated by Vaultage, but that need to be completed or instantiated by the developers to deliver application-specific functionalities; (c) fully generated classes, again specific to the application defined in the input model. We describe now the more relevant classes by their origin in the input model and their usage in the Vaultage framework.

- **Entities** of the data vault model are transformed to plain old Java object classes (POJOs). These classes contain the entity features as basic Java attributes and references, and their only methods are getters and setters to access these features. Friend, Post, and File are the generated classes from the respective entities in the model.
- **Vault** classes contain the features defined in the input model in the same way that the entities described above. Additionally, vaults have methods for the provided data services. In the figure, there is a single vault class (FairnetVault), which belongs to the partially generated classes. This class is partially generated because its service methods (e.g. addFriend, getPosts) are initially empty, and it is the task of the Fairnet developers to provide their implementation. These implementation classes are only generated once, so that developers do not have to manually create them from scratch. Basically, when a Fairnet vault receives a request message, such as addFriend, the response is generated by invoking the addFriend method of the FairnetVault class, and this response is packed and sent in a response message to the requester. Lastly, part of the functionality of the FairnetVault class is included into the FairnetVaultBase superclass. For instance, the signatures for the service methods must not be modified by developers because other generated code depends on that, so these signatures are included into FairnetVaultBase. There is yet another superclass from Vaultage, denoted Vault, that contains generic functionality such as the management of the encryption key pairs.
- **Remote Vault** classes allow sending requests and responses to a concrete vault type. At the right part of the figure, there is a RemoteFairnetVault class, which can be used to send a request to a concrete FairnetVault by means of its public key (Section 3.1 explains how this key is used as an identifier in the broker communication). This class has a pair of methods for each data service of the vault. The first method allows to send a request to that service of a remote vault (e.g. addFriend), and the second method is used to respond to a received request (e.g. respondToAddFriend).
- **Request Handlers** are responsible for processing request messages and invoking the appropriate vault methods to obtain a response. The response to a received request is sent back through a RemoteFairnetVault instance, as described in the previous paragraph. There is a request handler for each vault (e.g. FairnetRequestHandler), and it is fully generated.
- **Response Handlers** determine how a client (e.g. a Fairnet user requesting a service from a remote vault) processes

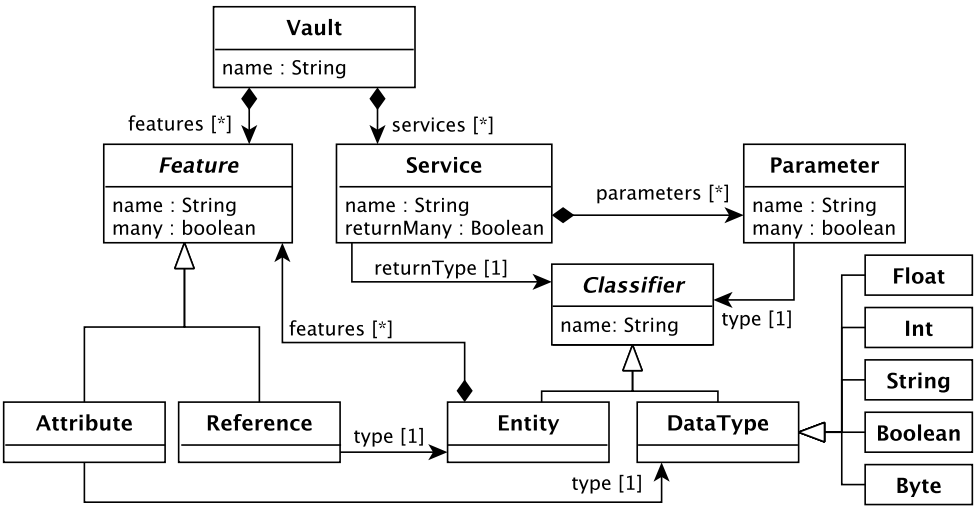

Figure 3 Vaultage metamodel depicted in Ecore

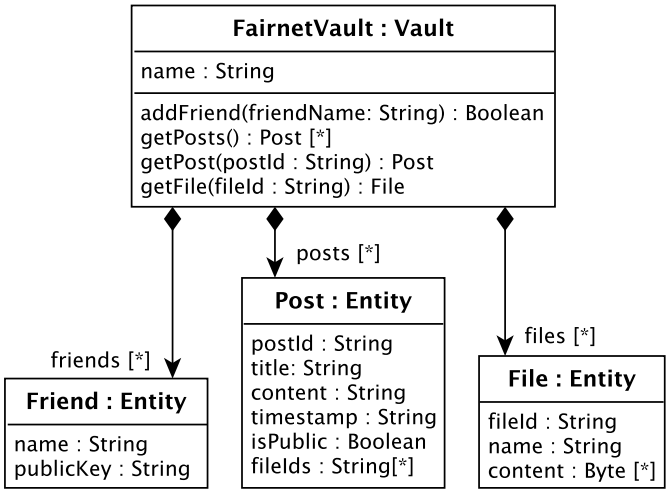

Figure 4 Fairnet model conforming to the metamodel of Figure 3

the returned response. For instance, in step 6 of Figure 1, Alice received Bob's acceptance of her friend request. As the response is positive, the Fairnet app automatically adds Bob as one of Alice's friends. If the response was negative, though, this addition would not take place. That logic has been defined in the Fairnet app by extending the AddFriendResponseHandler and implementing its run method, which has a result boolean parameter that provides the outcome of the original friend request.

– A **FairnetBroker** is generated for launching the Apache ActiveMQ broker server. It allows certain modifications, such as configuring the port in which the broker server is started.

4. Limitations

In this section, we highlight some limitations of our approach. While the inclusion of personal data vaults can improve the privacy, security and users' control over any personal data used by the application (Henman & Dean 2004; M. Mun et al. 2010; Ladjel et al. 2019), it does not come without challenges:

1. **Availability**. Currently, Vaultage stores user data on single devices, such as laptops or smartphones. Such an approach makes it less reliable than storing it on a cloud service that supports replication, and it can limit the availability of data stored in vaults to peers (e.g. when the devices are off or without internet connection). Adding support for synchronisation between devices could mitigate this by enabling users to maintain redundant copies of their vaults in multiple devices. Synchronisation also opens the possibility to acquire specific devices that are always connected under users control, in the same way that CloudLocker[7] promotes for providing personal cloud storage.

2. **Data Veracity**. While asymmetric encryption allows securing the communication between sender and receiver (see Section 3.1.1), the veracity of certain data cannot be certified by a single user of an application. For instance, if Fairnet allowed users to give likes to other users' posts, it is not possible to verify the real number of likes that a post has by simply asking their owner, as they might provide a wrong value in search of personal gain (Seo et al. 2019). Another example of important application data that requires verification is the reputation of a user in an online auction market (Pavlou & Ba 2000). Such data might require additional certification, such as cryptographic signatures by the users that liked a post, or an external blockchain-based registry of user likes (Hawlitschek et al. 2018).

3. **Security**. Vaultage uses private and public keys to decrypt/encrypt data and the public keys as the identifiers of users. Once a malevolent actor obtains the private key of a user (e.g. Bob), this actor could impersonate the original user

[7] https://www.cloudlocker.eu/en/index.html

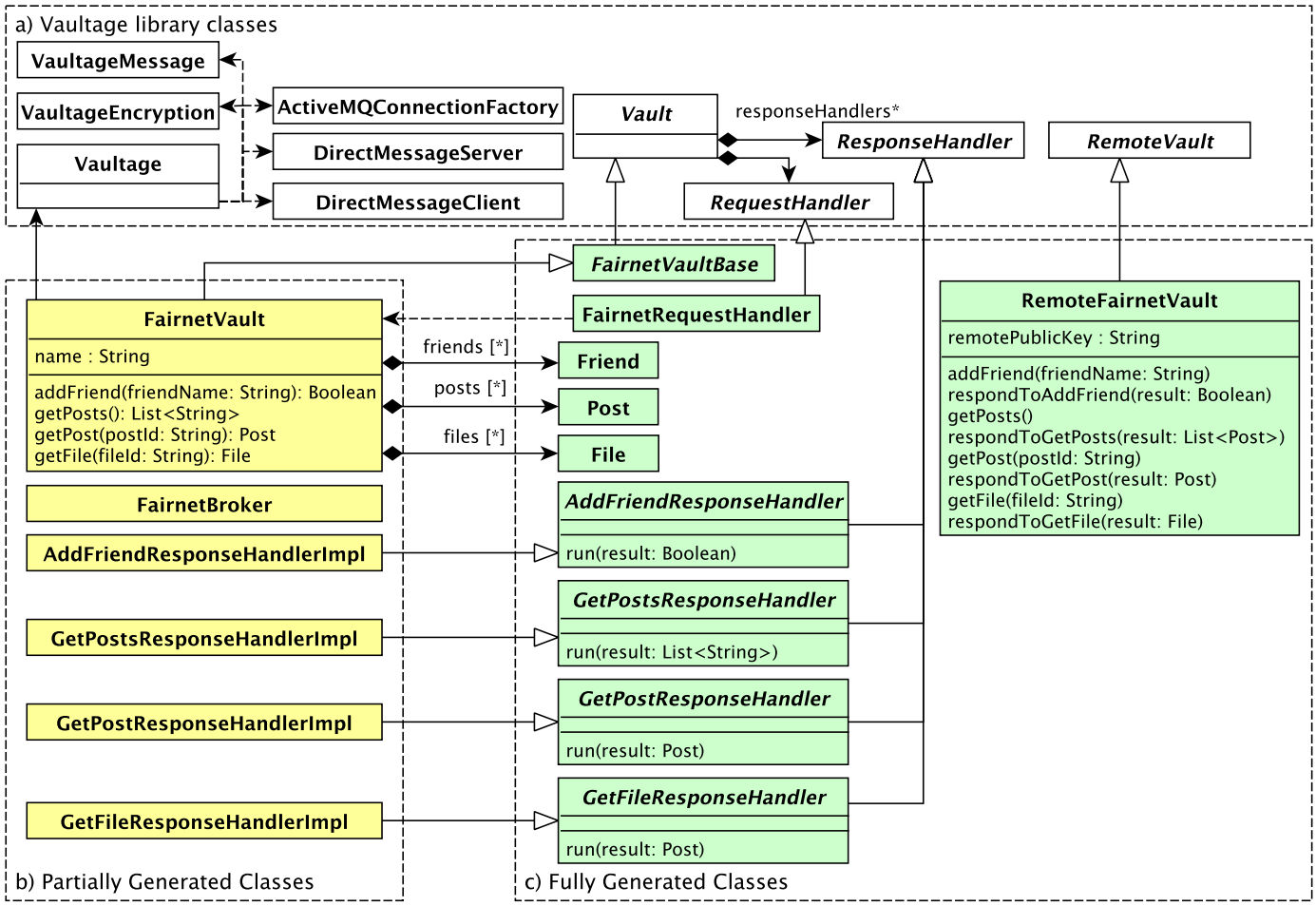

Figure 5 A simplified class diagram of the classes that are automatically generated by Vaultage for the Fairnet example (domain-specific wrappers (green) and skeleton code(yellow)) and core classes of Vaultage (white).

in the network. To fix such an issue, in the case of Fairnet, Bob needs to create a new account and tell his friends that his previous account has been hijacked so that they can remove the old account and add Bob to the new one. Centralised systems address these issues by allowing to change the login details, i.e., the user's password. However, this approach also comes with its own risks. Once a centralised system is breached (Stempel & Finkle 2017; Gunaratna 2016), the malicious actor might be able to access many users' data and take over their accounts. Decentralised applications are less attractive for cyberattacks, as in most cases breaching a single user is not as profitable as obtaining the data of thousands or millions of users. Another way in which malicious users can attempt to affect the communications of a Vaultage app is by flooding the broker server with a lot of fake requests. For these issues, including multi-factor authentication (Dasgupta et al. 2017) can increase the security of decentralised apps, as it does for centralised ones. Such an authentication can allow changing compromised key pairs at an account level, and it can permit a broker to only accept requests coming from authenticated users. Recent works also show that it is possible to use decentralised authentication (Lux et al. 2020), so that a centralised authentication server does not store the credentials of all users of a Vaultage application.

4. **Control**. Data that has been transferred to a requester is no longer under the control of the data provider. For example, Alice can do malicious activities with Bob's posts and files (e.g. pictures) that she obtained via requests to Bob's data vault. In this case, Bob cannot control any of Alice's actions. This issue is inherent to any sharing of data through online services, and it could be mitigated by establishing understandable terms of use within the application so that data misuses that violate such terms can be legally pursued.

5. Evaluation

In this section, we describe our evaluation of Vaultage. We start by presenting different data vault applications that showcase the capabilities of Vaultage. Then, we report on experiments we carried out to measure the overall performance of Vaultage's communication. We also highlighted the performance impact of certain aspects of the communication, such as using brokered or direct communication, and the cost of double message encryption.

Application	Model Lines	Generated Code Lines	Ratio
Fairnet	24	447	1:19
Pollen	20	338	1:17
Synthesiser	8	256	1:32

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

5.1. Data Vault Applications Generation

We have created three 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]. The examples of this paper are contrived and somewhat minimal to help readers understand how Vaultage works and to show how Vaultage can be applied to different contexts. With Vaultage's modelling and code generation capabilities, adding more services to these examples would be trivial, as developers would just need to define the new services at model level and most of the implementation code related to these would be auto-generated.

We used the cloc[9] tool to measure 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 were excluded from these counts.

Each data vault application has a specific objective that makes it different to the others. Since Fairnet was already introduced in Section 2.4, the following sections only discuss Pollen and Synthesiser.

5.1.1. Pollen This application is an example of using Vaultage purely for its communication capabilities. Pollen is a polling application that uses Multi-Party Computation techniques (MPC) (Goldreich 2002) to perform aggregated polls securely and without leaking the answers of individual participants. For instance, it is possible to ask a set of participants some questions (e.g. a 1-to-10 rating about a government decision, an opinion on 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, Pollen's main benefit is a non-trivial communication problem that has been very useful for polishing Vaultage's auto-generated message handling architecture so that it becomes easy to use for data vault application developers.

Figure 6 shows the Vaultage model for Pollen. Vaults in Pollen are the users who participate in MPC polls. There are two types of supported polls: number polls are answered by giving a single number (e.g. what is your age/salary?), while multi-valued polls require selecting from a set of options (e.g. select from 1 to 5 your satisfaction with the garbage collection system in your neighbourhood).

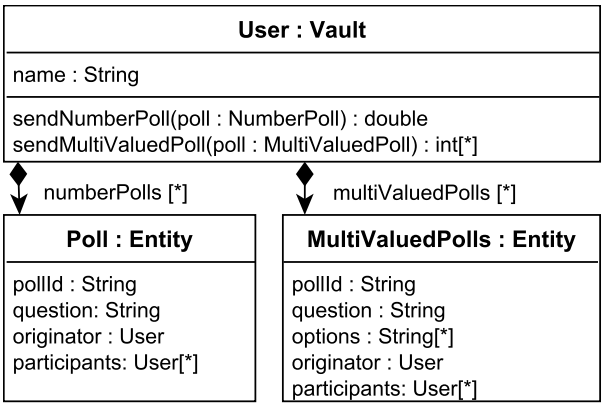

Figure 6 Pollen's Vaultage model.

An example of a number poll is shown in Figure 7. Let's say that Alice wants to obtain the average salary between Bob, Charlie, and her. To do that, she initiates a poll by sending a sendNumberPoll request to the first of the three participants, which in this case is Bob. When Bob receives the request, instead of immediately answering it, it sends a new request to the next participant in the list (Charlie) as part of the multi-party computation calculation. This process continues until the poll reaches again its originator, which in the example happens when Charlie sends back the question to Alice. Then, instead of answering with her real salary, Alice returns a large fake number as the response. What Charlie does then is responding to the intermediate request sent by Bob, for which Charlie adds his real salary to the fake number provided by Alice. This way, Bob receives a response from Charlie but he is not able to discern Charlie's salary from the aggregated value. When Bob receives this response, he aggregates his real salary and responds back to the original request from Alice. So, Alice receives an aggregate number resulting from adding the fake value and both Bob and Charlie's salaries, so Alice is not able to determine the specific salary of any of them (and these salaries would be more obfuscated as the number of participants grows). The last thing that Alice needs to do is subtract the original fake value and add her own salary, so she can calculate the average salary from that result.

5.1.2. Synthesiser This is a performance testing tool that allows evaluating the architecture generated by Vaultage. Figure 8 shows the Vaultage model of Synthesiser. Vaults in Synthesiser are denoted as workers, which store no data, and offer two data services: increment and getTextSize. An increment request provides a number as a parameter, and it is always responded with the following number (i.e. it adds one). This request is useful to perform stress tests with multiple workers receiving requests at the same time. For instance, in one of the implemented experiments workers receive a number of tasks to perform, each of these consisting of sending an increment request to another

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

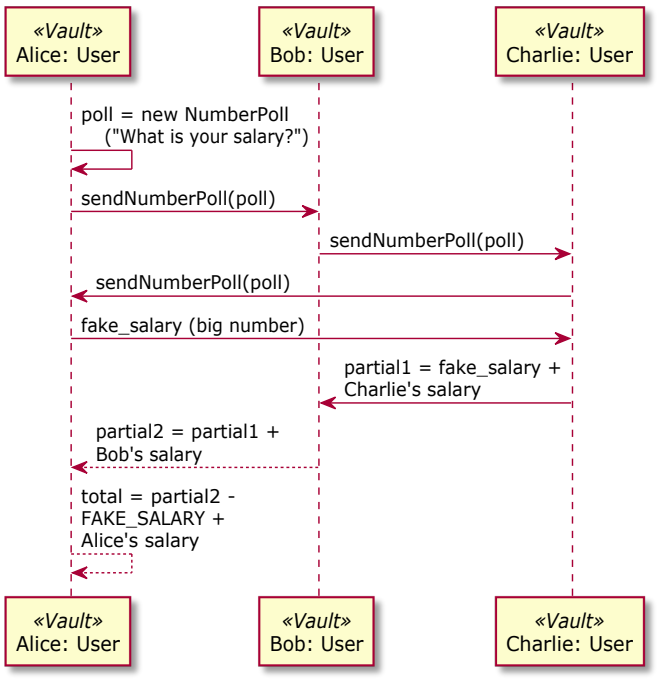

Figure 7 Example of an aggregated salary survey in Pollen.

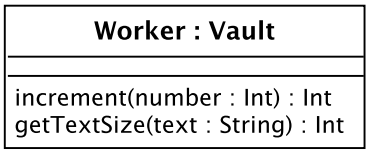

Figure 8 Synthesiser's Vaultage model.

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. Figure 9 shows an example of a worker requesting 10 tasks to another worker. 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. Also, 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. The other request, getTextSize, simply returns the size of the String text received as a parameter. This request is useful to test how the Vaultage network responds against messages of increasing size by controlling the size of the text parameter.

5.2. Vaultage Performance

We evaluated the performance of the vaults generated by Vaultage in a controlled network setup. We have profiled the time spent in the different aspects of the communication process, such as encryption and brokered/direct messaging. We have also included variations in the profiled experiments by modified the number and size of the messages.

The controlled network setup for the evaluation is depicted

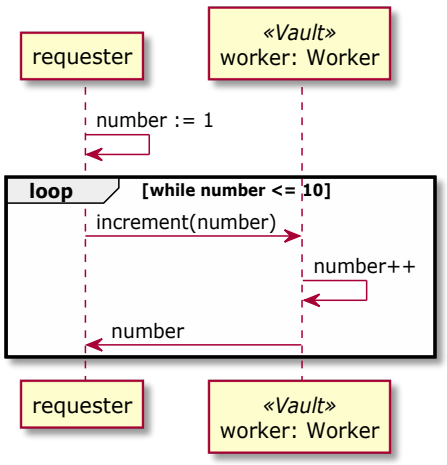

Figure 9 Requester sending 10 tasks to a Synthesiser worker

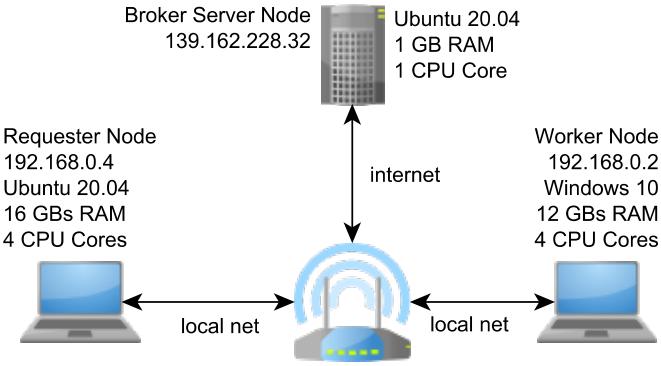

Figure 10 The controlled network setup for performance evaluation.

in Figure 10. This setup is composed of three nodes: the first node is the broker server node that runs Apache ActiveMQ on a cloud computing service, and the second and third nodes run Synthesiser worker vaults (see Section 5.1.2). Both the second and third nodes start a fixed number of workers in parallel. Workers from the second node act as requesters that send tasks to the workers in the third one. The broker node runs on a cloud system with Ubuntu 20.04 as operating system, one core of an AMD EPYC 7601 32-Core Processor 2.2 GHz, 1 GB RAM, and Java OpenJDK 8. The requester node runs with Ubuntu 20.04 operating system as well, a 4-core Intel(R) Core(TM) i5-7200U CPU @ 2.5 GHz, 16 GB RAM DDR4, and Java OpenJDK 11. The worker node runs Windows 10 in a 4-core Intel(R) Core(TM) i7-6500U CPU @ 2.5 GHz, 12 GB RAM DDR3, and again Java OpenJDK 11.

5.2.1. Stress Test The objective of this evaluation was to measure the total time required for a set of requester workers to get the results of a fixed number of assigned tasks. We used 3 requesters on the second node and 3 workers on the third node, with the message broker running in the cloud node. This experiment used the increment Synthesiser request, so an initial number was incremented as many times as request operations

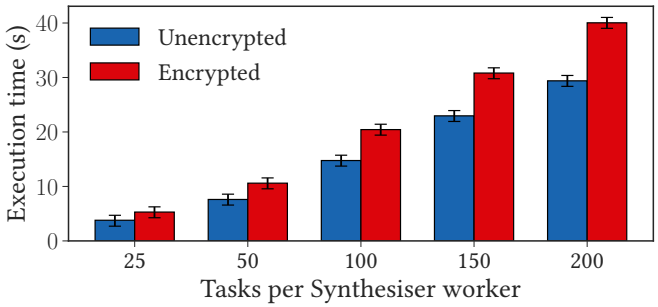

Figure 11 Completion times of Synthesiser worker tasks in brokered/direct and encrypted/plain messaging with an increasing number of operations.

Message Size	Brokered		Direct	
	Encrypted	Plain	Encrypted	Plain
60 KiB	1.91	0.18	1.59	0.02
120 KiB	3.60	0.26	3.10	0.04
180 KiB	5.21	0.34	4.59	0.04
240 KiB	6.89	0.40	6.11	0.05
300 KiB	8.53	0.47	7.56	0.06
1.5 MiB	42.57	2.00	37.02	0.18

Table 2 Seconds required to process a large text message request with respect to the encrypted/plain and brokered/direct messaging aspects

were sent by a requester. The work size ranged from 50 tasks per requester worker up to 250 tasks. To ensure our measurements were trustworthy, each Synthesiser run was run 7 times per work size, but we only took the last 5 values to reduce the warm-up effects of initial executions. Then, 95% confidence intervals for the times of each work size were calculated. The averaged values and their confidence intervals are depicted in the bar chart of Figure 11. Based on the values of these intervals, the measurements were stable across all work sizes.

The figure shows the weight that encrypting transmitted messages has in the total time required for a Synthesiser run to complete, both in brokered and direct messaging. The results show that, on average, plain messaging takes up to 42% and 49% less time in brokered and direct messaging, respectively, over the time required to complete a Synthesiser run with encryption. This kind of experiments could also be useful to compare different encryption approaches, e.g., measuring the cost of using greater RSA encryption key lengths. In Figure 11, it can also be noticed that there is a consistent penalty of using brokered messaging over the direct one. The penalties are 47% and 54% for encrypted and plain messaging respectively.

5.2.2. Increasing Message Sizes Test The objective of this experiment was to profile the time used by the vaults when handling messages of increasing sizes, again with respect to the encrypted/plain and brokered/direct aspects of the messaging. To do so, we used the other data service provided by Synthesiser workers, getTextSize. We constructed request messages containing texts with sizes ranging from 60 KiB to 1.5 MiB. The results can be found in Table 2. We can see that encryption takes the majority of the message time, as plain messages take from 90.6% at 60 KBs to 95.3% at 1.5 MBs less time to complete in brokered communication and from 98.2% at 60 KBs to 99.5% at 1.5 MBs in direct communication. With respect to the communication mode, and on average, sending large messages through the broker server is 1.15 times slower in encrypted mode and 7.51 times slower in plain mode than using direct messaging.

Referring again to the results of Table 2, if we aim for the 2-second rule (Nah 2004), which is the maximum waiting time if we want to keep a user engaged with a system, then an encrypted brokered message with a size of around 60 KBs would meet the threshold. For unencrypted brokered messaging, a 1.5-MB message is still acceptable for the standard. In contrast, plain direct messaging can still handle a message with a size of more than 22 MBs in 2 seconds if it is linearly projected based on the data.

5.2.3. Threats to Validity In terms of threats to validity, there are factors in our evaluation environment setup (Figure 10) that can affect the performance of Vaultage in real-world conditions. For instance, differences in computing power of each node (e.g., processor, memory, etc.) and the speed of network connection can translate into different results to the ones of our evaluation. Also, while deploying the broker server in a remote cloud node is a realistic scenario, the lack of control over how the cloud provider administers this node might be affecting the obtained results. We tried to mitigate these issues by executing the experiments several times and at message sizes large enough to make the results more robust against the effects of variance and noise.

6. Related Work

Providing users with full control of which personal data they want to share with third parties is a service currently offered by several applications. A good example of these is Solid (Solid 2017), 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 where pods are stored, so it allows avoiding centralised backends and opens the possibility for users to store their data locally. However, Solid pods need to be stored in a location with a publicly accessible IP/DNS, which makes them unsuitable for running on mobile devices and laptops, and the data is not encrypted by default. We plan to study the potential benefits of supporting some Solid components in Vaultage, such as its user authentication infrastructure. Applications following similar approaches are Digi.me[10], CozyCloud[11],

[10] https://digi.me/
[11] https://cozy.io/en/

or CloudLocker[12], 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 (Basin et al. 2011; Perez et al. 2013; Gerking & Schubert 2019), which are general concerns of any network infrastructure. However, less effort has been put into the automatic generation of secure network communication interfaces such as the ones provided by Vaultage. We have only found one application aiming to generate this kind of communication-related code, in the context of the 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 (Berrouyne et al. 2019) allows defining 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, model-to-text transformation is performed to generate the network code to deploy in the system's IoT devices, freeing engineers from having to deal with low-level network details. While Vaultage does not offer a way to specify fine-grained network constraints, it is a general-purpose framework that supports solid encryption mechanisms for communication between more complex nodes than the usual IoT devices.

This work is an extension of our previous workshop paper (Yohannis et al. 2020). Some of the new features we added are the direct communication mode between vaults, which allows a vault to send a message directly to another vault without the relay of a central broker, and the support for transferring data in other types (e.g., media files). We also extend the previous paper by adding an in-depth discussion on the motivation behind this work and a description of Vaultage's features. We expand the evaluation of the expressiveness and code generation capabilities of Vaultage by including new use cases (Pollen and Synthesiser). We also add a load test to measure the implication of message sizes on Vaultage's performance in different communication aspects. Lastly, we discuss the limitations and challenges associated with the proposed distributed architecture.

7. Conclusions and Future Work

We have presented Vaultage, a framework intended to support developers when creating data vault applications. Vaultage offers automatic code generation of decentralised data vault networks, including brokered and direct 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 integrate Vaultage with payment wallets and cryptocurrency networks to allow vault owners to monetise their data services. Also, we plan to study how to enable users to concisely query and aggregate data stored in multiple data vaults from a high-level query language or interface. We will also 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).

[12] https://www.cloudlocker.eu/en/index.html

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

Basin, D. A., Clavel, M., & Egea, M. (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* (pp. 1–10). ACM. doi: doi:10.1145/1998441.1998443

Berrouyne, I., Adda, M., Mottu, J., Royer, J., & Tisi, M. (2019). Cypriot: framework for modelling and controlling network-based iot applications. In C. Hung & G. A. Papadopoulos (Eds.), *Proceedings of the 34th ACM/SIGAPP symposium on applied computing, SAC 2019, limassol, cyprus, april 8-12, 2019* (pp. 832–841). ACM. doi: doi:10.1145/3297280.3297362

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-Driven Software Engineering in Practice* (Vol. 1). (Issue: 1 ISSN: 2328-3319) doi: doi:10.2200/S00441ED1V01Y201208SWE001

Cadwalladr, C., & Graham-Harrison, E. (2018). *Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach.* https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election.

Cavoukian, A., et al. (2009). Privacy by design: The 7 foundational principles. *Information and privacy commissioner of Ontario, Canada*, *5*, 12.

Christin, D., Reinhardt, A., Kanhere, S. S., & Hollick, M. (2011, November). A survey on privacy in mobile participatory sensing applications. *Journal of Systems and Software*, *84*(11), 1928–1946. doi: doi:10.1016/j.jss.2011.06.073

Dasgupta, D., Roy, A., & Nag, A. (2017). Multi-factor authentication. In *Advances in user authentication* (pp. 185–233). Cham: Springer International Publishing. doi: doi:10.1007/978-3-319-58808-7_5

Erkin, Z., Veugen, T., Toft, T., & Lagendijk, R. L. (2012). Generating private recommendations efficiently using homomorphic encryption and data packing. *IEEE Trans. Information Forensics and Security*, *7*(3), 1053–1066. doi: doi:10.1109/TIFS.2012.2190726

European Network and Information Security Agency. (2014). *Privacy and data protection by design - from policy to engineering.* (Tech. Rep.). LU: Publications Office. Retrieved 2021-05-21, from https://data.europa.eu/doi/10.2824/38623

European Union. (2016, May). 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*. Retrieved from https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC

Gerking, C., & Schubert, D. (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* (pp. 61–70). IEEE. doi: doi:10.1109/ICSA.2019.00015

Goldreich, O. (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)

Gunaratna, S. (2016). *LinkedIn: 2012 data breach much worse than we thought.* https://www.cbsnews.com/news/linkedin-2012-data-breach-hack-much-worse-than-we-thought-passwords-emails/.

Hawlitschek, F., Notheisen, B., & Teubner, T. (2018). The limits of trust-free systems: A literature review on blockchain technology and trust in the sharing economy. *Electronic Commerce Research and Applications*, *29*, 50-63. doi: doi:https://doi.org/10.1016/j.elerap.2018.03.005

Henman, P., & Dean, M. (2004). The governmental powers of welfare e-administration. In *Australian Electronic Governance Conference 2004.* Melbourne, Australia: Department of Political Science, University of Melbourne.

Krebs, B. (2013). *Adobe Breach Impacted At Least 38 Million Users.* https://krebsonsecurity.com/2013/10/adobe-breach-impacted-at-least-38-million-users/.

Ladjel, R., Anciaux, N., Pucheral, P., & Scerri, G. (2019, August). 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. Retrieved from https://hal.archives-ouvertes.fr/hal-02269203

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

Luger, E., Moran, S., & Rodden, T. (2013). Consent for all: revealing the hidden complexity of terms and conditions. In W. E. Mackay, S. A. Brewster, & S. Bødker (Eds.), *2013 ACM SIGCHI conference on human factors in computing systems, CHI '13, paris, france, april 27 - may 2, 2013* (pp. 2687–2696). ACM. doi: doi:10.1145/2470654.2481371

Lux, Z. A., Thatmann, D., Zickau, S., & Beierle, F. (2020). Distributed-ledger-based authentication with decentralized identifiers and verifiable credentials. In *2020 2nd conference on blockchain research applications for innovative networks and services (brains)* (p. 71-78). doi: doi:10.1109/BRAINS49436.2020.9223292

Maisonneuve, N., Stevens, M., Niessen, M. E., & Steels, L. (2009). NoiseTube: Measuring and mapping noise pollution with mobile phones. In I. N. Athanasiadis, A. E. Rizzoli, P. A. Mitkas, & J. M. Gómez (Eds.), *Information Technologies in Environmental Engineering* (pp. 215–228). doi: doi:https://doi.org/10.1007/978-3-540-88351-7_16

Mohan, P., Padmanabhan, V. N., & Ramjee, R. (2008). Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems - SenSys '08* (p. 323). Raleigh, NC, USA: ACM Press. doi: doi:10.1145/1460412.1460444

Mun, M., Hao, S., Mishra, N., Shilton, K., Burke, J., Estrin, D., … Govindan, R. (2010). Personal data vaults: A locus of control for personal data streams. In *Proceedings of the 6th international conference.* New York, NY, USA: Association for Computing Machinery. doi: doi:https://doi.org/10.1145/1921168.1921191

Mun, M. Y., Kim, D. H., Shilton, K., Estrin, D., Hansen, M., & Govindan, R. (2014, June). PDVLoc: A Personal Data Vault for Controlled Location Data Sharing. *ACM Transactions on Sensor Networks*, *10*(4), 1–29. doi: doi:10.1145/2523820

Nah, F. F.-H. (2004). A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, *23*(3), 153-163. doi: doi:10.1080/01449290410001669914

Pautasso, C. (2014). Restful web services: Principles, patterns, emerging technologies. In *Web services foundations* (pp. 31–51). New York, NY: Springer New York. doi: doi:10.1007/978-1-4614-7518-7_2

Pavlou, P. A., & Ba, S. (2000). Does online reputation matter? an empirical investigation of reputation and trust in online auction markets. *AMCIS 2000 Proceedings*, 77. Retrieved from https://aisel.aisnet.org/amcis2000/77/

Perez, S. M., García-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., & Cabot, J. (2013). Model-driven extraction and analysis of network security policies. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, & P. J. Clarke (Eds.), *Model-driven engineering languages and systems - 16th international conference, MODELS 2013, miami, fl, usa, september 29 - october 4, 2013. proceedings* (Vol. 8107, pp. 52–68). Springer. doi: doi:10.1007/978-3-642-41533-3_4

Potey, M. M., Dhote, C., & Sharma, D. H. (2016). Homomorphic encryption for security of cloud data. *Procedia Computer Science*, *79*, 175 - 181. (Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016) doi: doi:https://doi.org/10.1016/j.procs.2016.03.023

Rivest, R. L., Shamir, A., & Adleman, L. (1978, February). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, *21*(2), 120–126. doi: doi:https://doi.org/10.1145/359340.359342

Rose, L. M., Paige, R. F., Kolovos, D. S., & Polack, F. A. C. (2008). The epsilon generation language. In I. Schieferdecker & A. Hartman (Eds.), *Model driven architecture – foundations and applications* (pp. 1–16). Berlin, Heidelberg: Springer Berlin Heidelberg.

Senarath, A., & Arachchilage, N. A. G. (2018, June). Why developers cannot embed privacy into software systems?: An empirical investigation. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (pp. 211–216). Christchurch New Zealand:

ACM. doi: doi:10.1145/3210459.3210484
Seo, Y., Kim, J., Choi, Y. K., & Li, X. (2019). In "likes" we trust: likes, disclosures and firm-serving motives on social media. *European Journal of Marketing*. doi: doi:https://doi.org/10.1108/EJM-11-2017-0883
Sivaraman, V., Carrapetta, J., Hu, K., & Luxan, B. G. (2013, October). HazeWatch: A participatory sensor system for monitoring air pollution in Sydney. In *38th Annual IEEE Conference on Local Computer Networks - Workshops* (pp. 56–64). Sydney, Australia: IEEE. Retrieved 2021-05-23, from http://ieeexplore.ieee.org/document/6758498/ doi: doi:10.1109/LCNW.2013.6758498
Solid. (2017). *The Solid Project.* https://solidproject.org/.
Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *Emf: Eclipse modeling framework*. Pearson Education. Retrieved from https://books.google.co.uk/books?id=sA0zOZuDXhgC
Steinfeld, N. (2016). "i agree to the terms and conditions": (how) do users read privacy policies online? an eye-tracking experiment. *Computers in Human Behavior*, *55*, 992 - 1000. doi: doi:https://doi.org/10.1016/j.chb.2015.09.038
Stempel, J., & Finkle, J. (2017). *Yahoo says all three billion accounts hacked in 2013 data theft.* https://www.reuters.com/article/us-yahoo-cyber/yahoo-says-all-three-billion-accounts-hacked-in-2013-data-theft-idUSKCN1C82O1.
Wakefield, J. (2014). *eBay faces investigations over massive data breach.* https://www.bbc.co.uk/news/technology-27539799.
Wilson, S., Schaub, F., Liu, F., Sathyendra, K. M., Smullen, D., Zimmeck, S., … Smith, N. A. (2019, February). Analyzing Privacy Policies at Scale: From Crowdsourcing to Automated Annotations. *ACM Transactions on the Web*, *13*(1), 1–29. doi: doi:10.1145/3230665
Wood, A., Najarian, K., & Kahrobaei, D. (2020, August). Homomorphic encryption for machine learning in medicine and bioinformatics. *ACM Comput. Surv.*, *53*(4). doi: doi:https://doi.org/10.1145/3394658
Yohannis, A., de la Vega, A., Kahrobaei, D., & Kolovos, D. (2020). Towards model-based development of decentralised peer-to-peer data vaults. In *Proceedings of the 23rd acm/ieee international conference on model driven engineering languages and systems: Companion proceedings.* New York, NY, USA: Association for Computing Machinery. doi: doi:https://doi.org/10.1145/3417990.3420043
Zou, Y., Danino, S., Sun, K., & Schaub, F. (2019, May). You 'Might' Be Affected: An Empirical Analysis of Readability and Usability Issues in Data Breach Notifications. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1–14). Glasgow Scotland Uk: ACM. doi: doi:10.1145/3290605.3300424

About the authors

Alfa Yohannis is a Research Associate in the Department of Computer Science at the University of York. His research interest is in software engineering, particularly decentralised systems, model-based software engineering, change-based models, software engineering education, and gamification. He is also a lecturer in the Department of Information Technology at the Universitas Pradita, Indonesia, and an awardee of the LPDP scholarship. You can contact him at alfa.yohannis@york.ac.uk.

Alfonso de la Vega is a Research Associate at the Department of Computer Science of the University of York (United Kingdom), and an external researcher from the University of Cantabria (Spain). His current research revolves around model-driven software engineering topics, with a special emphasis on model visualisation and comparison. He received his PhD from the University of Cantabria in 2019, where he studied how to incorporate the benefits of model-driven engineering and domain-specific languages into the data manipulation, storage and analysis fields, with a special focus on reducing the complexity of conventional analysis processes. You can contact him at alfonso.delavega@unican.es.

Dimitris Kolovos is a Professor of Software Engineering in the Department of Computer Science at the University of York, where he researches and teaches automated and model-driven software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-driven software engineering platform, and an editor of the Software and Systems Modelling journal. He has co-authored more than 150 peer-reviewed papers and his research has been supported by the European Commission, UK's Engineering and Physical Sciences Research Council (EPSRC), InnovateUK and by companies such as Rolls-Royce and IBM. You can contact him at dimitris.kolovos@york.ac.uk.