Bucharest Academy of Economic Studies
Faculty of Cybernetics, Statistics and Economic Informatics
IT&C Security Master

DISSERTATION THESIS

Scientific coordinator:
Lecturer Ph.D. Andrei TOMA

Graduate:
Mihai TURCU

Bucharest
2022

Bucharest Academy of Economic Studies
Faculty of Cybernetics, Statistics and Economic Informatics
IT&C Security Master

Developing a distributed processing cloud platform using Fully Homomorphic Encryption

DISSERTATION THESIS

Scientific coordinator:
Lecturer Ph.D. Andrei TOMA

Graduate:
Mihai TURCU

Bucharest
2022

Declaration concerning content originality and responsibility assumption

I hereby declare that the results presented in this work are entirely the result of my own creation except where reference is made to the results of other authors.

I confirm the fact that any content used that originates from other sources (journals, books and Internet websites) is clearly referenced in the paper and is indicated in the list of bibliographic references.

Table of Contents

Contents

1. Introduction

Since the very outset, cloud computing has been flawed.

Currently, in a world that is evolving under the sign of Big Data, cloud privacy and processing conflict with each other. This derives from the need to decrypt data before processing it, which is implicitly impacting its confidentiality. However, a new frontier is emerging.

In an effort to resolve this dissension, fully homomorphic encryption represents a viable solution by safeguarding data at rest, in transit, and, uniquely so, during operations, empowering cooperative processing on sensitive, encrypted data.

In recent times, industries have had access to more data than ever before, allowing them to hone their processes and, therefore, offering greater added value. The true potential of data can be unearthed by analyzing it, for instance by using it as the input to a series of algorithms. These procedures often consist of complex pipelines of operations that require high computational effort and, as such, it is more scalable and beneficial for several machines to cooperate in a highly performant cluster, rather than performing all the processing on a single computer. Throughout the process of distributing the data and analyzing it, security should be of the utmost concern, as there are certain classes of data that are characterized by a confidential nature.

Traditional systems focus only on safeguarding two of the main areas of the data analysis lifecycle, by ensuring security at rest and in transit. However, when data is being handled and used, it mainly occurs in plaintext format, potentially after having applied some anonymization techniques to it. This type of usage could lead to data exfiltration and, implicitly, to bad actors gaining access to sensitive information, either through social engineering or data collection tools used by hackers.

In contrast, fully homomorphic encryption (FHE) is a scheme that allows processing to be done directly on encrypted data, ensuring that the only agent that can access the raw data and its further refined forms is a trusted party that possesses the secret, decryption key. By leveraging on the strengths of this paradigm, the drawbacks of sharing data between devices that partake in a distributed computing network can be minimized. For example, the previously mentioned data exfiltration threat can be mitigated by configuring a fully homomorphic system with the proper security parameters, strong enough to make it infeasible for attackers to decrypt the data. Likewise, social engineering threats are less impactful since the data that is being processed is in ciphertext format, and the agent doing the processing does not possess the decryption key.

A key domain in which FHE can be applied is cloud computing. So far, several investigations have tackled the introduction of this encryption scheme to the cloud sphere. However, the prospect of inducting it into the core of a secure distributed data processing outsourcing system with untrusted Internet agents has not yet been thoroughly explored.

Therefore, the primary focus of this thesis is to provide a deeper understanding and a robust proof-of-concept implementation of how FHE can be the main security element of a scalable data processing platform. Specifically, this thesis shows how, by relying on the always-encrypted state of data, the Internet can become a viable and boundless source of computational infrastructure.

To this end, the thesis describes how the benefits and properties of fully homomorphic encryption can be intertwined into a framework composed of several modules: a front-end platform that interacts with organizations in need of data processing and also with those wanting to offer computational capabilities, a back-end system which orchestrates the transmission of inputs and outputs between the two aforementioned groups and, last but not least, a web-based module which transforms even the machines of untrusted individuals into agents which can securely perform data computations.

Traditionally, website visitors have mainly been offered advertisements as a means of monetizing website content. However, the computer system envisioned in this thesis offers them the opt-in opportunity to instead provide the hardware of their device in order to process data chunks from time to time.

The proposed system envisions a mutually beneficial situation, in which website owners, further referred to as data processors, relate to individuals or companies, known as data suppliers, looking to outsource complex pipelines of transformations that they want to apply to payloads of data. The web platform intermediates data transfer between the involved parties and, additionally, oversees the processing of payments. The last part of the computer system is a browser compatible module, through which encryption operations and data computations are made. The module grants the computer system its scalability by offloading processing to website visitors, but it has also been designed with user experience in mind, by moving all heavy workloads onto a background thread, thus maintaining the unaltered consumption of website content.

In the following section of this thesis, related work in the field of fully homomorphic encryption is reviewed. Afterwards, the architecture of the computer system and the technical implementation of all the platform's modules are presented. Ultimately, conclusions are drawn, highlighting the assets of the platform, such as providing a means of outsourcing data processing in a secure manner and creating an alternative monetization scheme. Moreover, potential future developments are highlighted, envisioning how the system could evolve and stabilize as new and more performant fully homomorphic encryption schemes and implementations emerge.

2. (Fully) Homomorphic Encryption

2.1 Homomorphic encryption types

Homomorphic encryption can be classified into three classes of schemes, depending on the types of operations supported and the number of consecutive computations allowed: partially homomorphic, somewhat homomorphic, and fully homomorphic.

The first partially homomorphic encryption scheme was detailed in [1], then named "privacy homomorphism". In this paper Rivest, Adleman and Dertouzos attested that encryption schemes that enable operations to be performed directly on encrypted data were feasible and provided technical details and clear applications of how this paradigm would be useful in the financial industry. The key characteristic of partially homomorphic encryption schemes is that they allow only one type of operation, either additive or multiplicative, to be performed over the encrypted data with a potentially infinite number of computational iterations.

Further homomorphic encryption research in the following decades focused on overcoming the mentioned singular operation type limitation. As a result, in 2005 a new generation of homomorphic encryption algorithms appeared, coined somewhat homomorphic encryption. First described in [2], this type of scheme is characterized by the fact that it concurrently supports the usage of additive and multiplicative operations, a significant improvement over its partially homomorphic predecessor. However, the initial scheme described by Boneh et al. also has a shortcoming, as it can support only one multiplicative iteration while allowing an unbounded number of additive operations to be performed on the encrypted data.

In 2009, the homomorphic world witnessed a groundbreaking instance of innovation in the form of a feasible fully homomorphic encryption scheme, which had previously only been theorized. Craig Gentry's Ph.D. dissertation thesis [3] detailed how a practical implementation of fully homomorphic encryption could work. In contrast to partially homomorphic encryption, fully homomorphic encryption allows for any kind of operation to be evaluated, not limited to one type at a time. Moreover, Gentry also introduced a valuable new concept called bootstrapping, through which noise introduced by computations done on the ciphertext, which could lead to decryption errors, could be reduced, allowing for an unbounded number of operations to be run on the data while maintaining its integrity. The key weakness of first generation fully homomorphic schemes was that their performance was many factors below the equivalent plaintext computations. In the next years, new schemes emerged that offered refined performance, such as the BFV [4] scheme which replaced the bootstrapping technique with a learning-with-errors re-linearization technique. Moreover, some also added additional functionality, such as CKKS [5], which allows floating-point operations to be performed in a fully homomorphic environment.

Being the scheme that currently offers one of the highest processing speeds, the computer system presented in this thesis uses BFV as the default fully homomorphic scheme of choice for datasets which are composed exclusively of integer values. For data that contains floating-point values, the encryption module automatically detects this and switches to the CKKS scheme, in order to support this use case.

2.2 Usage in cloud data processing

There have been several recent studies in the field of outsourced data processing that have opted for the introduction of homomorphic encryption as a means of enhancing privacy and security. Of particular note is the scheme described in [6], in which the authors outline a lightweight, efficient computer system for secure data computations in the cloud. A key difference between their study and this thesis is that their encryption scheme of choice is a partially homomorphic encryption one. While this type of encryption still offers a potentially infinite number of computations to be done on the ciphertext, the drawback compared to FHE is that only one type of operation, either additive or multiplicative, can be performed over the encrypted data, which limits the flexibility and versatility of the computer system. The authors do provide valid reasons for this choice, as FHE is a nascent concept and current implementations are not yet production-ready in terms of speed or storage utilization. However, recent advances in this domain aim to improve these limitations, as will be noted in the next subsection. Another factor of dissimilarity is that in their scheme, the actor that handles the encrypted data processing and interacts with data consumers, coined "Access Control Server", is part of a strictly defined network of computers, as opposed to this thesis's more decentralized approach. On the one hand, their perspective leads to a potentially more trusted environment for clients, as they can choose which ACS to opt for. On the other hand, this choice limits the scalability of the system, since computations are limited to interactions with this closed network of specific devices. This thesis instead opts for an approach that parallelizes operations on datasets by splitting them into small chunks to be processed by a wide network of data processors, namely the devices of website visitors.

Another related piece of work comes from Liu et al. who designed in [7] a highly efficient privacy-preserving cloud data processing system, which uses trusted execution environments such as the Intel SGX to safeguard secret information in an enclave. While they do not use fully homomorphic encryption, a key benefit of their approach is that the execution environment of the Lightcom framework that they describe allows for robust confidentiality and data integrity guarantees. While the latter is a limitation that has yet to be solved in a practical manner in the homomorphic encryption domain, the former is mitigated by the very fact that data is processed in an encrypted state. Security-wise, fully homomorphic encryption can be a viable option in untrusted environments such as the Internet because of its stochastic nature. If FHE schemes were to use a deterministic approach, ciphertext-only attacks would be a possibility because bad actors in the form of website visitors would have unfettered access to ciphertext, originating from their status as data processors. However, the two schemes used in this thesis, namely BFV and CKKS, have a stochastic nature, and so this type of exploit has not yet been proven to be possible.

Relating to the execution environment of this thesis's computer system, an inquiry into the viability of data processing on the Internet was undertaken in [8]. The study showcases a master-worker model which enables the collaboration of globally interconnected volunteer-provided devices toward solving computational tasks. While the paper establishes a robust and fault-tolerant framework for mapping and reducing data, it does not tackle the security aspect sufficiently which, in contrast, is at the core of this thesis. By incorporating FHE within this untrusted network, data suppliers can be more inclined to put their trust in such a framework, knowing that their sensitive data is not at risk.

As previously shown, there is still room for new ideas and technical coverage in this domain and, as such, this thesis shall detail the development of a secure distributed data processing outsourcing system starting in the next section.

2.3 Limitations & future developments

In the final part of this literature review, future developments and solutions to current fully homomorphic limitations are discussed. A primordial concern ever since Gentry's major discovery has been performance, as fully homomorphic operations have traditionally been much more expensive and slow compared to plaintext ones. However, several recent studies have tackled this issue and have been able to improve performance drastically, by proposing new designs that allow for optimized, hardware-based operations or by utilizing existing resources such as the GPU. An example of the former is F1, described in [9] as a specialized accelerator capable of speeding up fully homomorphic encryption dramatically by up to 17412 times, which would massively improve the applicability and practicality of FHE. When it comes to this dissertation's computer system, however, this approach would require data processors to invest in such a device, and the assumption of website visitors having a dedicated accelerator in their system would, in most cases, not be able to be fulfilled.

However, a more feasible performance gain could be achieved through GPUs, which most website consumers own. For instance, Jung et al. [10] succeeded in optimizing the multiplication speed in the CKKS floating-point FHE scheme by up to 7 times, and its bootstrapping phases by 257 times compared to CPU implementations, by leveraging the GPU's speed while reducing main memory access on it. The authors also note that these newfound optimizations could be partially transferred to other FHE schemes.

Another limitation of current FHE schemes and implementations is that verifiable computation is difficult to achieve in an efficient manner, thus raising concerns regarding the integrity of processed data, since bad actors could choose to alter the computations applied to the data received as input. A recent study [11] seeks to remedy this by applying zero-knowledge SNARKs to ring computations, which are used in several FHE schemes including BFV and BFV. Their ring arithmetic proof framework, aptly named Rinnochio, aims to be generic enough to be applicable to a variety of environments. While they do not offer a practical implementation, leaving it as future work, their discoveries are crucial especially in untrusted environments, offering a means of verifying that FHE outputs are correct and have not been created through means other than the intended ones that the system provides, thus improving framework reliability.

Finally, a concern of particular importance and actuality is whether or not homomorphic encryption can withstand the test of quantum computing, which will most likely be adopted in the short to medium term. In [12], the authors design a quantum partially homomorphic scheme which is able to be used for data querying. What is more, the study also investigates how this scheme can be extended to a multi-user one, granting simultaneous access to the database to authenticated entities.

Compared to previous research, the study takes steps towards providing privacy to the users of the system, by using one-time encrypted input data which is transformed through quantum means and used within the database queries. Moreover, the databases themselves are also protected. This is realized through a mix of allowing the users to query only the data exposed by the database and authenticating the users prior to the execution of queries.

The authors also highlight the security and efficiency of their scheme, achieved through means such as protection against eavesdropping and honesty checks, but also the correctness of data queries which is granted by the usage of universal quantum circuits.

Ultimately, further inquries into enhancing performance and reducing the resources necessary for providing a robust security level are left as future work.

As has been noted in this subsection, fully homomorphic encryption is still a relatively new, largely volatile and constantly evolving subdomain of cryptography. As such, research is steadily bridging the gap towards a production-ready future, in which cloud systems can rely on this powerful new paradigm in order to more securely and efficiently outsource computations of all kinds.

3. Solution architecture

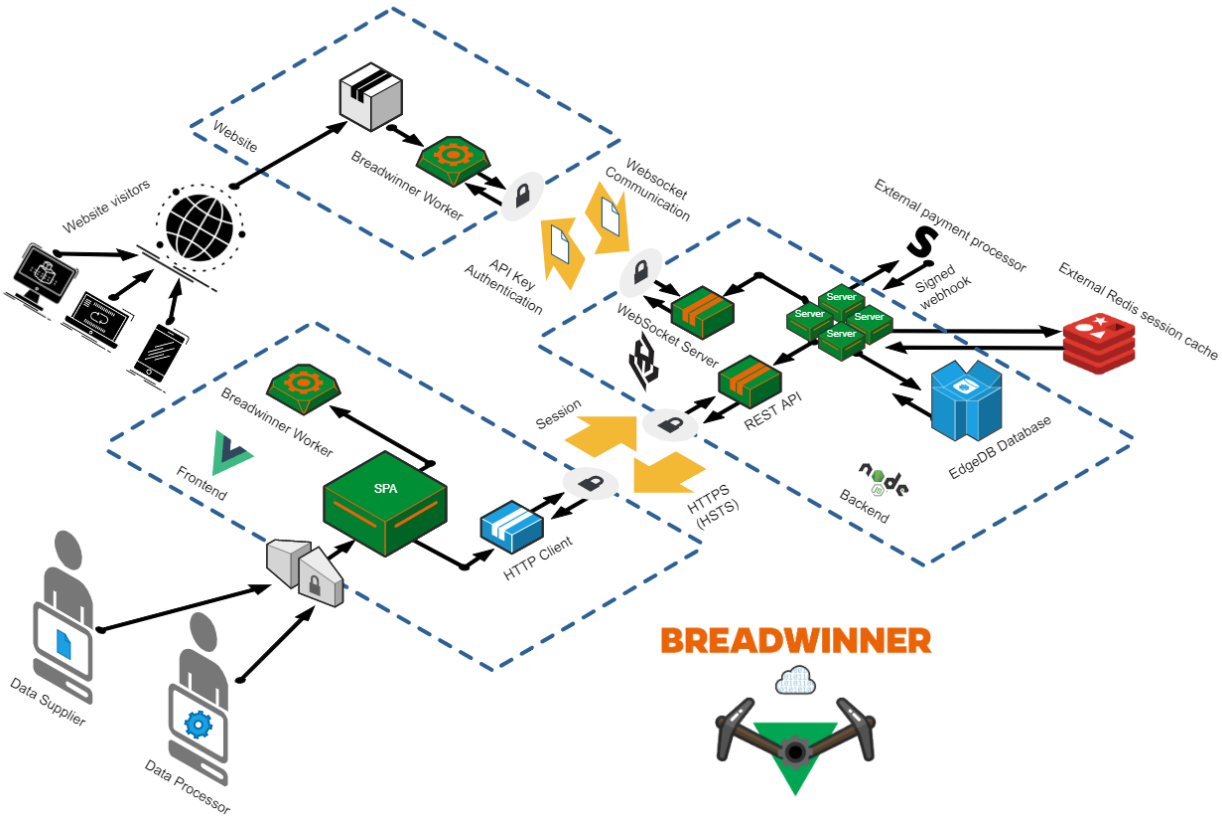


Figure 1. *Solution architecture.*

The above architecture showcases all the modules and actors participating in the envisioned computer system, nicknamed Breadwinner. In the following subsections, each of the solution's modules and the ways in which they collaborate are discussed.

3.1. Web module

This component is the core of the computer system and enables its main functionality. It represents a lightweight, browser-compatible and self-contained package that can be embedded into websites in order to receive chunks of homomorphically encrypted data to process.

In order to establish a WebSocket connection with the back end through which chunks can be obtained, clients must initialize the module with an API key that was previously generated in the front-end module. As a measure of mitigating the hijacking of API keys, each of them has a specific web hostname attached which disallows activation attempts originating from other domains, thus rendering the key unusable to unauthorized actors.

After initialization, the module sends processing requests to the server at a specified time interval. The server replies with a data chunk and the details required to process it such as a public key and a schema that contains the type and order of operations to apply. Additionally, the server relays to the client a randomly generated token, which distinguishes the client as

being authorized to send back to the server the output of processing the exact chunk that it has received. The data then undergoes processing through a calculator-like sub-module, which orchestrates the procedure by also taking into consideration the types of the operands of a certain operation, for instance being able to sum two arrays but also to add a plaintext value to an array.

After the processed output has been obtained, the client sends it together with the authorization token to the server, where it is saved and marked as processed.

Besides the calculator sub-module, the Breadwinner module also includes an FHE component that performs the specified operations on the data, but can also handle its encryption and decryption, the latter functionality being used on the front-end platform.

3.2. Front-end platform

The front-end module interfaces directly with the human actors involved. Data suppliers are the actors who wish to have their data processed in a secure manner and are allowed to upload datasets and form pipelines of operations that will be evaluated in a secure, homomorphic way. A BPMN collaboration diagram illustrating how data payloads are processed in the Breadwinner system is showcased in figure 2.

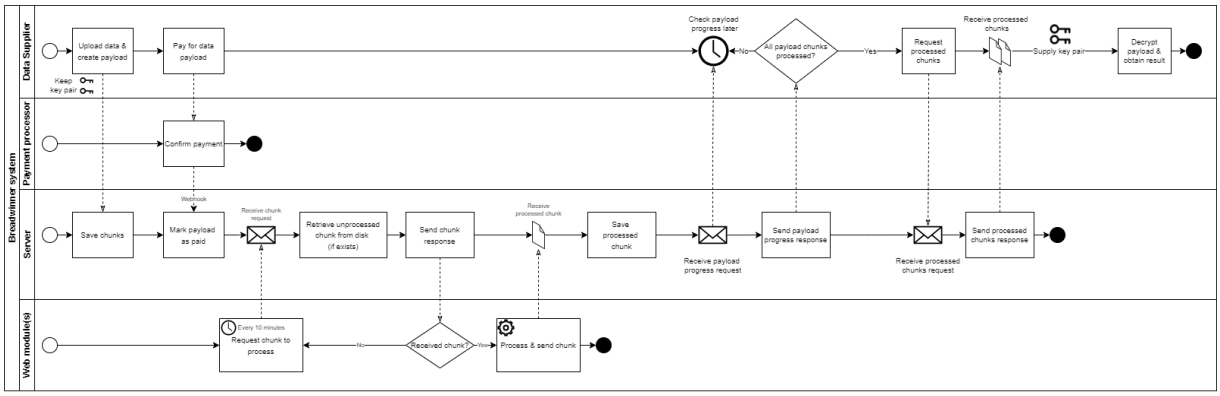


Figure 2. *Data processing collaboration diagram.*

Conversely, data processors are website owners who wish to tap into an alternative business model by signing up for an API key and embedding the secure data processing module into their websites. Thus, their clients will be able to choose between seeing advertisements or offering the computational resources of their device from time to time.

The security of the platform is ensured by a robust password policy, but users may also configure two-factor authentication for heightened security. Moreover, when data suppliers upload data payloads and a cryptographic key pair is generated, or when data processors generate API keys, the sensitive credentials never leave the client's device, and they are not stored on the server, in order to maintain their confidentiality.

As a means of illustrating the alternative monetization scheme created by this computer system, integration with an external payment system has been done. Data suppliers are required to pay a sum of money before their data payload is activated within the computer system and sent to web modules for processing. In contrast, data processors get paid automatically by the back-end module on a weekly basis for the number of processed data chunks that their websites have accomplished.

The price of an uploaded payload is dynamically calculated based on its size and complexity, as can be seen in the equation below.

$$pp = MIN(cp * nc + OP * no * nc, MPP)$$

Where:

- *pp = payload price*
- *cp = chunk price*
- *nc = number of chunks*
- *OP = constant, price per operation*
- *no = number of operations in pipeline*
- *MPP = constant, minimum payload price*

Equation 1. *Payload pricing formula.*

3.3. Back-end platform

This component is represented by the load-balanced servers, which each encapsulate two main sub-components.

The first one is an HTTPS server that communicates with the front-end module by authenticating users and serving and receiving information. For added security, the server sends an HTTP Strict Transport Security (HSTS) header to force secure communication on subsequent requests. Post-login communication is session-based, which empowers strict authorization based on the principal's properties. This means that an authenticated user, potentially an ill-intentioned or curious actor, can only access their own resources, since their identity is tied to their session, which is stored server-side. As such, retrieving the confidential resources and information of a separate user is strictly prohibited. The storage and retrieval of sessions are fast, thanks to a dedicated in-memory cache.

The second back-end sub-module is a WebSocket protocol server, which communicates with authenticated websites in order to receive processing requests from Breadwinner modules embedded within them. All WebSocket communication is encrypted, thanks to WebSocket Secure (WSS).

In addition, the server also manages payment-related operations. Similarly to the front end, it has been integrated with an external, dedicated payment system in order to ensure utmost security and global compliance with laws. Communication with this payment processor is bidirectional, as both incoming and outgoing requests use the HTTP protocol. However, in the case of the former, a special webhook endpoint has been defined on the server, in order to allow the external partner to inform the back end of events that occur, such as the successful completion of a payment or the configuration of a data processor's payment details. The webhook's events are signed, and so their source can be verified through signature validation, rendering the server immune to the impersonation of the payment server.

3.4. Partnered websites

The last, external, module of the computer system is represented by partnered websites, and implicitly their visitors. The high scalability of the system relies in no small measure on these actors, since they provide the infrastructure on which data processing is accomplished.

After registering an API key for their domain through the front-end module, data processors can easily install and integrate the Breadwinner module into their website. The only intervention required is initialization, which can be done by supplying the API key.

FHE data processing can be presented to website visitors as an alternative to advertisements. In this way, consent can be obtained from clients by placing the decision of seeing commercials or passively processing data into their own hands.

3.5. Database

The chosen database is of a relational type, as there are clear-cut relationships between the entities and all the data stored within the database is structured.

The central entity is the "User", which represents a generic abstraction of the computer system's users and contains account details, such as the username, hashed and salted password, role or two-factor authentication related secret.

The "Confirmation" entity has a 1:1 relationship with the "User" entity. It is created when a user signs up, sent as a URL to the user's email and used in order to finish the registration process. It contains a UUID, an expiration date after which it is no longer be usable, and a Boolean value which states whether the confirmation was used.

The two user subclasses contain role-specific information. "DataSuppliers" have a 1:m relationship with the "Payloads" that they upload, while "DataProcessors" contain linked payment account details and can have multiple API key child entities.

The "Payload" entity stores a user-consumed label and metadata about the encrypted data to be processed, such as the type of keys required to process the encrypted data, but also the schema containing the operations to be performed on the data. The schema is stored in a JSON format, and contains information such as the operations' types, operand names and plaintext operand values.

Payloads are formed of one or more "Chunks", small parts of encrypted datasets, which are sent to the Breadwinner module whenever one is requested. Chunks also store information about their processing status and which data processor completed their processing, so that they can be paid.

Payments store the URL at which they can be accessed online, but also details regarding their creation timestamp or current state. Data suppliers and payments are characterized by a 1:m relationship.

Within the database, no payment-related data is sensitive, as it only represents metadata from the external payment provider's system, such as UUIDs. Thus, all handling of confidential information, such as credit card details or extensive payment details, is delegated to the payment provider, maintaining its safety in the case of a local data breach.

Finally, API keys hold information such as their registered domain or a prefix that aids their corresponding data processor in properly identifying them. For all purposes, API keys, like passwords, are treated as sensitive credentials, and so they are not stored in plaintext format, but hashed.

The figure below showcases the entities of the EdgeDB database, their properties and the links between them.

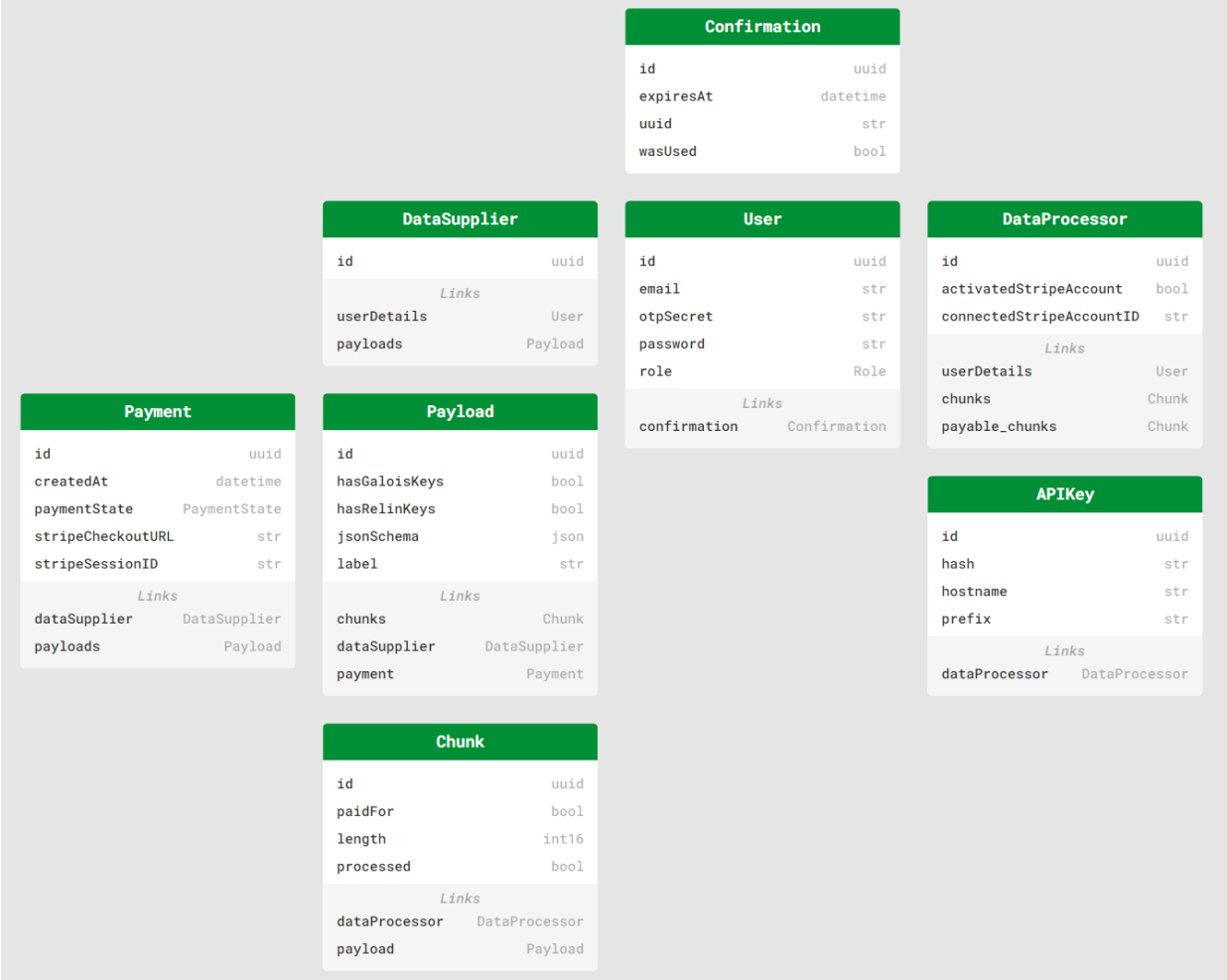


Figure 3. *Database schema.*

4. Technology stack

The used technology stack is modern and comprised of state-of-the-art programming languages, frameworks and libraries, chosen in order to achieve high performance, ease of use, security and correctness.

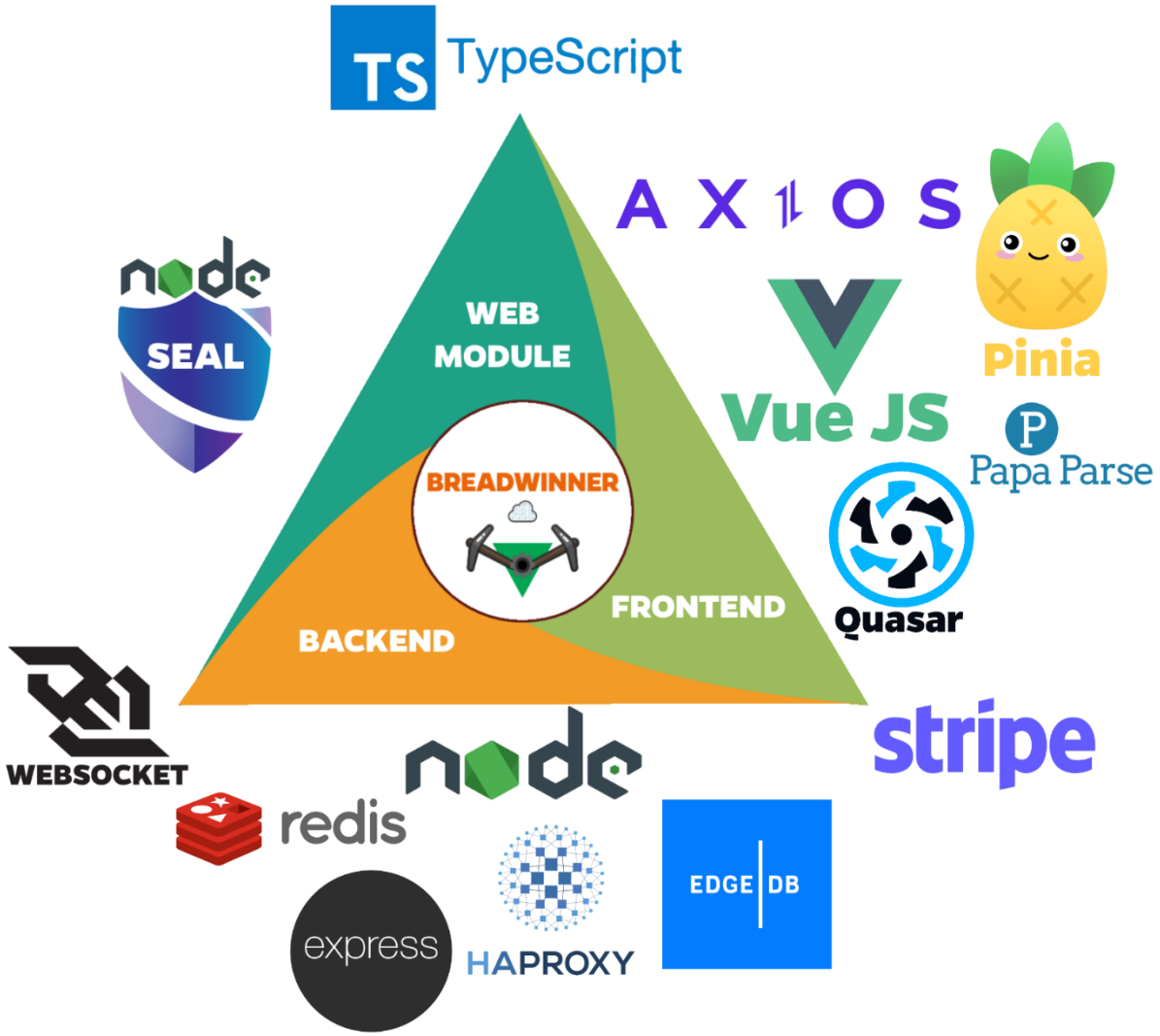


Figure 4. *Technology stack.*

4.1. TypeScript

TypeScript stands at the core of all the computer system's modules, as it is versatile and capable of being present in both a local, server context and a web one, by virtue of its transpilation to JavaScript. Its usage is central in overcoming one of JavaScript's most glaring issues, the fact that it is untyped. Through this superset of the base language, compile-time safety can be achieved, and additionally the readability, scalability and predictability of the source code is improved. Especially in the case of a library module, such as the web module of this computer system, using TypeScript provides users insight into the API and reduces onboarding time.

Furthermore, Typescript offers constructs which do not exist in the base JavaScript language. Such an example are enums, sets of constants which are logically grouped. They are used extensively throughout the application thanks to the readability that they provide, for instance in the case of managing user roles, as can be seen below.

```
export enum Role {
    DATA_SUPPLIER,
    DATA_PROCESSOR
}
```

4.2. Web module stack

The web module is lightweight, using default browser-provided technologies such as Web Workers for background thread access or WebSockets for communication with the back-end system. These being said, it does have one, crucial dependency, namely the FHE Node SEAL library [13].

Node SEAL

The library is written as a TypeScript abstraction that uses WASM (WebAssembly) in order to retain near-native performance and provide portability, which is crucial to the Breadwinner system as the web provides heterogeneous device types through its visitors. The WASM implementation represents a compiled version of the original C++ Microsoft SEAL library.

Several FHE scheme implementations are offered, such as BFV or CKKS. The former is preferred when all values are integers, whereas the latter is the scheme of choice in the case of fractional numbers.

By virtue of this library, several operations have been implemented in the web module as a proof-of-concept. For instance, users can add, multiply or subtract values from two arrays, but also perform the same operations or other ones, such as exponentiation, between an array and a desired plaintext value.

One of the main functionalities provided by the package is a context, which can be configured with several parameters in order to tweak the level of security or the maximum supported numerical value. Additionally, Node SEAL offers dedicated encoders and decryptors, but also a mechanism through which ciphertext values can be serialized to a Base64 string. Since encrypted data payload chunks must be saved when received from data suppliers and later distributed to the active web modules by the back-end system, the serialization feature is valuable and used extensively within the computer system.

4.3. Front-end stack

The front-end stack contains technologies used for functionality but also for interface design, as this module of the system is the one that interfaces with the human actors of the computer system.

Vue JS 3

Vue JS 3 [14] is the framework used in the development of the front-end web application, together with the Quasar UI components library.

One of the key advantages of this framework and also a point that sets it apart from its competitors is that it offers a powerful reactivity system, using the Proxy design pattern in order to achieve this, namely the Proxy base JavaScript object. Vue offers special kinds of objects, known as “computed properties”, whose values depend on other defined variables, and so they are recomputed and cached when their dependencies change. Furthermore, “watchers” are an Observer type of function, and are called when their observed values are modified. Together, these constructs enable complex and clean flows.

There are many cases in which this kind of usage is present throughout the front-end module, but a prime instance of this is when a data supplier is configuring the pipeline that their data will traverse after being encrypted. Certain operations, such as addition, support a unary version, as is the case of summing the elements of an array. Conversely, other operations are categorically binary, such as subtraction, as is the case of subtracting a column of a dataset from another column or another intermediary result.

Through reactivity, these validations can be automatically applied whenever an operand is added or an operation is changed, thus simplifying the underlying code and hardening the application by minimizing the number of logical bugs that could appear.

Another advantage of Vue is the virtual Document Object Model (DOM) that it manages. Compared to the traditional DOM that is present in the browser, operations on the virtual one are less expensive, and they can also be executed in batches, reducing the number of necessary re-renders. Moreover, Vue makes sure to track the components that change and only re-render those, instead of the entire DOM, therefore improving performance and allowing the platform to include more complex views.

Pinia

Pinia is a store solution for the Vue ecosystem. By integrating it into a web application, Pinia can provide a means of sharing logic and data between components, offering a cleaner way of managing state throughout the application and preventing the extensive buildup of props being sent to components, also denoted as “prop drilling”.

An instance in which Pinia is instrumental is the case of defining operations within the payload creation process. As new operations can have as operands the results of previously defined operations, the component through which operations are defined can receive the entire pipeline from the store by mapping it, therefore being able to use its contents.

Papa Parse

This last library is transparent to the users. However, it provides an important functionality, that of being able to parse CSV format files into JavaScript objects. Through its configuration, CSV files uploaded to the web app are quickly processed on a background thread, in order to provide a better user experience.

4.4. Back-end stack

The final internal part of the computer system is represented by the back end. It is the central module that interacts with both the front end and the web modules. As such, its technology stack is more varied, out of the need to support multiple protocols.

Node.js

At its core, the server is written in TypeScript and Node.js based, and thus executed in Chrome's proprietary V8 engine [15]. This allows it to execute compiled, machine code which is highly efficient.

Another key advantage of Node.js is one of its core mechanisms, the event loop. It allows for non-blocking IO operations despite being single-threaded, by queueing operations and diligently communicating with the host system in order to react through callbacks to tasks being finalized. This is quite impactful for the Breadwinner computer system and was a key factor in deciding to use this runtime environment, as the server must be able to handle a large amount of file reading and writing when saving and distributing data payload chunks. The figure below showcases the cycling of event loop tasks.

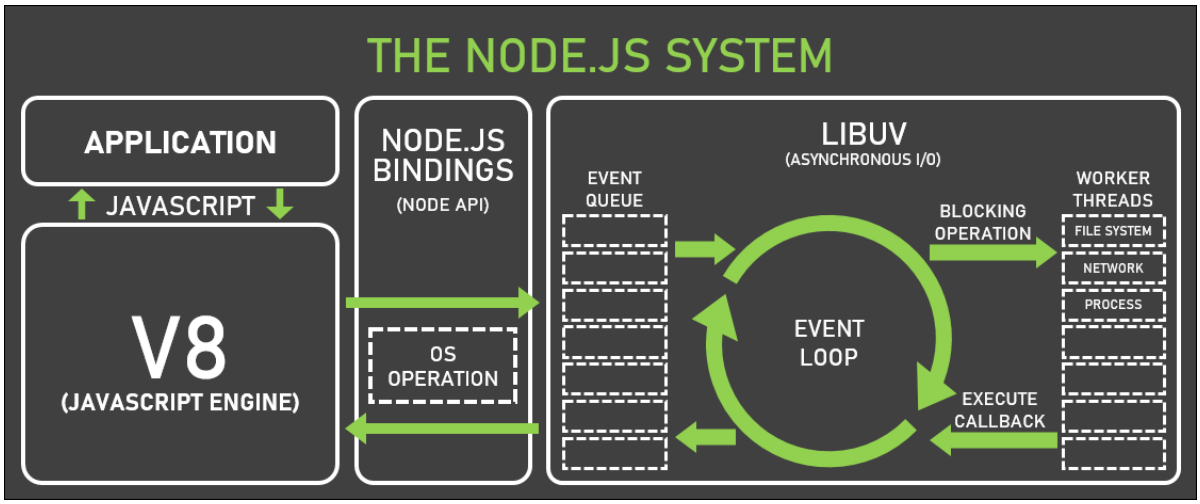


Figure 5. *Node.js event loop. Source: https://www.tutorialandexample.com/node-js-event-loop*

Finally, a productivity advantage of Node is its rich package ecosystem, offering performant and well-tested libraries for many purposes, as will be exemplified further. A potential disadvantage of using libraries is the introduction of third-party vulnerabilities, but this can be mitigated by rigorously auditing them and by using trusted, well-established packages. For instance, after installing the back-end system's dependencies, Node Package Manager (NPM) was able to detect 4 vulnerabilities, which could be fixed after running the command *npm audit fix*. This updated the vulnerable libraries' versions to new, secure ones.

Express

The *express* package is at the core of the API exposed by the server to the front-end module. It enables the server to define endpoint routes, which can receive HTTP requests and reply with responses. Express directly facilitates the communication between the front-end and back-end modules, which is done securely through HTTPS. Middleware can also be defined on a per-route basis, which allows for the interception of requests and the execution of specific logic.

In the case of the Breadwinner system, there are several middleware defined, which handle authentication, session management and CSRF token validation. Middleware logic can also short-circuit the execution chain of a request, which is useful in the case of declining unauthenticated requests on routes which should be available only to users that have logged in.

Finally, a related library named “routing-controllers” was used together with Express in order to allow for a class-based syntax definition of routes, which aids in grouping up related routes and allows their configuration via annotations.

Redis

Several packages, such as *redis* and *connect-redis* are used in order to connect to a Redis server that has been set up on a local Ubuntu virtual machine. Redis [16] enables swift access to session information, thanks to its in-memory storage of data. In a real-world scenario, in a production environment, the external approach of running the Redis server on a different machine would be more beneficial, leading to better performance since it would not compete for resources with the back-end server. Moreover, while not exemplified in this proof-of-concept implementation, Redis also allows for both horizontal and vertical scalability thanks to its Cluster topology.

Express-session

This package offers an express middleware which allows for the management of sessions within the server’s context. Sessions are established when users log into the web application. Before logging in, users are assigned a pre-session. Sessions interact with the *csurf* package in order to be able to generate CSRF tokens for a particular user’s session.

A specific case is that of avoiding login CSRF exploits in which a bad actor tricks a legitimate user into logging into the attacker’s account, potentially uploading sensitive information.

Sessions are also used in order to strictly limit user access to their own resources. This is accomplished by retrieving the ID of the user whose resources should be accessed directly from the session object injected by the middleware, thus disallowing any user from accessing another user’s data.

Finally, *express-session* integrates with the previously mentioned Redis server in order to efficiently store and retrieve session data to and from an in-memory cache.

Class-validator

In order to ensure the integrity of the computer system, all validations that are done on the front end are mirrored on the back-end system, to prevent potential invalid hand-written requests created by bad actors which tampered with the front-end web application.

To this end, the *class-validator* package allows for the abstraction of request data payloads by extracting them into classes, where their properties may be annotated for validation purposes. In the case of validation failure, the request is rejected with a 400 Bad Request HTTP response code.

WS

This package is used for setting up the WebSocket Secure server to which web modules can communicate. It allows for deep configuration, such as delegating the HTTP to WebSocket protocol upgrade process to the server. This is advantageous because it allows for the enactment of an authorization procedure before accepting connection requests, as is the case of the API verification procedure done when web modules wish to connect to the back-end server.

Stripe

The *stripe* package is the official library published by the payment processor for JavaScript applications. Within the Breadwinner back end, it provides functionalities such as creating payment sessions when data payloads are uploaded, or the creation of managed Stripe accounts for Data Processors, through which they can automatically receive payments for their processing. This integration allows for a proof-of-concept implementation of how monetization may be achieved within an outsourced data processing system.

Argon2

For the protection of user passwords within the system, the Argon2 algorithm has been used, its implementation being provided by the NPM package with the same name. It is the winner of the Password Hashing Competition and recommended by OWASP as a state-of-the-art password storing algorithm [17]. Specifically, within the back-end system the Argon2id variant is used, which grants resistance against both side-channel and GPU-accelerated attacks. It allows for the storage of hashed and salted passwords within the database, which discourages the usage of rainbow tables.

Edgedb

This package allows the server to interact with the database, through which it can manage entities. The back-end system relies on EdgeDB [18], a new database solution offering a novel graph-relational implementation. Compared to the ubiquitous Object Relational Mapping (ORM) tools traditionally used by projects in order to interact with the database, EdgeDB offers a lower-level solution that retains the high-level usability, while offering superior performance, comparable to hand-written, optimized SQL queries. Based on the database's schema, previously showcased in section 3.5., a tool named *edgeql-js* generates a custom query builder that can be used to run complex queries against the database. Furthermore, the query builder is fully type-safe, thanks to TypeScript.

An example of the query used in order to create an API key for a data processor can be seen below.

```
await queryBuilder.insert(queryBuilder.APIKey,
{
 prefix: apiKeyString.slice(0, 5),
 hash: generateSHA512(apiKeyString),
 hostname: payload.hostname,
 dataProcessor,
}).run(db);
```

Dotenv

In order to follow the best practice of not storing sensitive credentials within the source code, such as the Stripe API & webhook keys, or connection credentials, the *dotenv* package has been

used. This allows for the creation of a .env file within the project's structure in which key-value pairs can be defined as environment variables. Upon initializing the server and calling *dotenv's* corresponding function, the local .env file is read and the variables are injected into the process, making them available to be used within the entire web application. Finally, the .env file has been purposefully ignored from the Git version control system, in order to avoid tracking it and its credentials anywhere other than the machine on which the server runs.

Otplib & qrcode

Together, these packages allow for the implementation of a two-factor authentication security system. The first package, *otplib*, is used for creating time-based one-time password (TOTP) seeds, which are stored within the back end and within a dedicated authenticator application chosen by the user. These seeds are then used to generate TOTP tokens that are required upon each login into the application after the user activates the two-factor mechanism.

The second package, *qrcode*, is used in order to generate a data URI which contains an image representation of a QR Code, which is sent to the user to be scanned into an authenticator app.

Nodemailer

Nodemailer empowers the server to connect to SMTP providers and to send emails automatically. This helps the Breadwinner system to send confirmation emails to users upon signing up, which they can use in order to finish setting up their account.

Node-cron

This final package is used in order to set up scheduled tasks at certain moments in time, defined through cron syntax. For instance, the reimbursement of data processors is done at midnight at the start of each week, while the deletion of unpaid payloads that have been abandoned for at least an hour is scheduled to run at the start of every hour.

Haproxy

Finally, while website visitors improve the scalability of the system through the computational infrastructure that they provide, the back-end system must also be able to scale.

Therefore, on the Ubuntu VM on which Redis also resides, a Haproxy load balancer was set up, in order to distribute both HTTPS and WSS requests to multiple servers.

HTTP requests are distributed to the involved servers using the "Round Robin" algorithm, which ensures that consecutive requests will be routed to different servers, evenly spreading out the load. However, in the case of WebSocket, the session of a web module remains open with the original server to which the WebSocket connection request was first routed, maintaining a so-called "sticky session". Thus, throughout the lifecycle of the WebSocket session, messages from the same client will be routed to the same server.

When using Haproxy, the two main options of handling SSL are SSL termination and SSL passthrough. Both have their benefits and drawbacks in terms of security and efficiency.

On the one hand, by using SSL termination, the load balancer acts as an edge device and handles the encryption and decryption of messages. This means that SSL-related computation may be offloaded from servers to the machine on which the balancer runs, therefore improving their

performance, especially if using a dedicated accelerator. However, a drawback of this approach is that the messages are transmitted between the load balancer and servers using plain HTTP, leaving them vulnerable to sniffing exploits.

On the other hand, SSL passthrough allows each individual server to handle the SSL process, ensuring end-to-end encryption between the client and the server and improving security. The drawback of this approach is that packet inspection is not available, therefore smart load balancing is not achievable since data cannot be examined.

Considering the fact that the main objective of this thesis's computer system is ensuring security, the second option has been chosen, as it does not expose data in its journey from the load balancing device to the server.

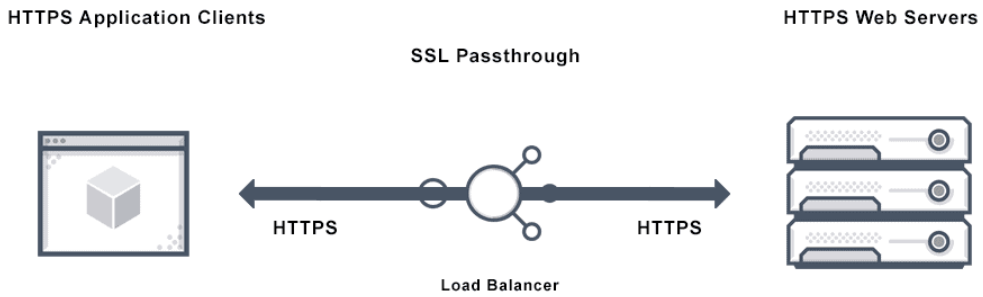


Figure 6. *SSL Passthrough. Source: https://avinetworks.com/glossary/ssl-passthrough/*

5. Solution implementation

In the following subsections are highlighted the implementation details of each module of the computer system.

5.1. Web module

The embeddable web module's source code is TypeScript-based and is ultimately transpiled to JavaScript. The module itself is offered to website owners as a package. For the purposes of this thesis, the package is hosted locally, as opposed to installing it from a manager such as NPM, yet its usage remains similar to a hosted library's, simply having to specify its disk location instead of a semantic version in the package.json file of the application's project. Within the web module, several classes collaborate, as shown in the diagram below, each having a specialized role in order to respect separation of concerns.

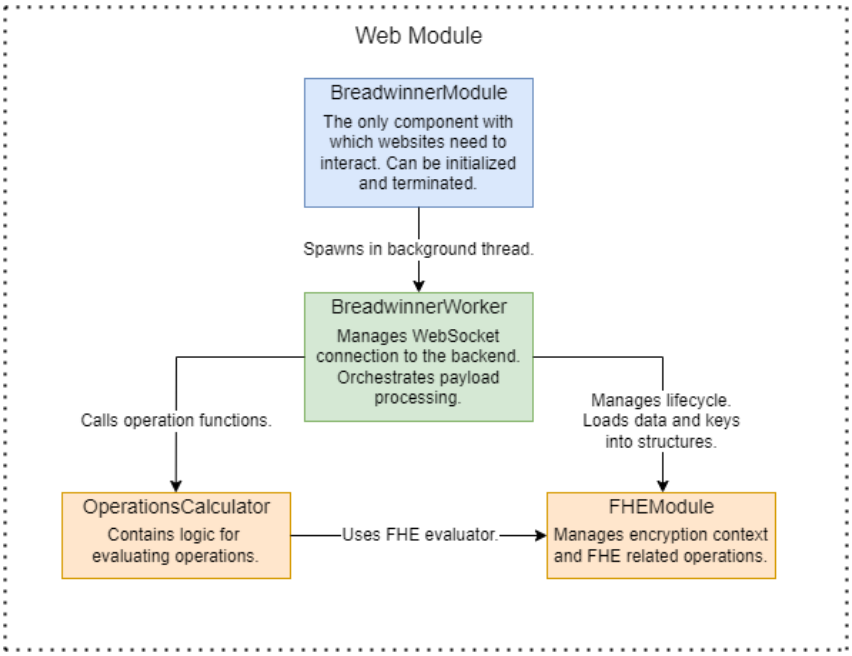


Figure 7. *Web module classes.*

BreadwinnerModule

The entry point of the module is a singleton class named *BreadwinnerModule*. It contains a simple API, exposing two methods through which the module can be initialized or terminated. Additionally, it contains two private properties: one named *instance* which is used for the implementation of the singleton design pattern, and another named *worker*. The latter one is a Web Worker instance, used in order to migrate all processing to a background thread, as FHE data processing should be transparent to the end-user and should not impact their browsing experience.

When the module is activated using an API key, a Web Worker background thread process is created, and a message containing the API key is sent to it. Through a listener, it receives a clone of the API key string message, which it uses in order to initialize the second main class of the web module, the *BreadwinnerWorker*.

BreadwinnerWorker

Upon being initialized, this class attempts to connect to the back-end system via WebSocket Secure (WSS) by sending it the API key in an application-specific header named *Sec-WebSocket-Protocol*. This is more secure than using a query parameter, which would be more likely to be logged throughout its lifecycle. When the connection request reaches the server, it is treated as a handshake upgrade request to the WebSocket protocol. In order for the connection to succeed and be opened, the server validates the received API key by hashing it using the SHA-512 algorithm and matches the hash together with the *host* header value against the keys stored in the database. If a corresponding match is found, the connection type is switched and upgraded from HTTPS to WebSocket Secure. Otherwise, the connection is dropped. Therefore, there is no situation in which a WebSocket connection is opened between web modules and the server without proper authentication being used.

After the connection has successfully been opened, the web module worker instance requests its first chunk to process. This request can either be fulfilled, if the server holds any chunks that still require processing, or an empty response can be sent. If the web module does not receive a payload to process, it simply skips the processing part and attempts to obtain a chunk after a set amount of time. On the contrary, a processing payload that is received from the server contains a verification token, ciphertext data in Base64 format, the length of the encrypted data, and the operations schema in JSON format, detailing FHE schema to be used and the operations that should be applied homomorphically to the data. Each operation contains a result type, a type, representing the actual name of the operation i.e. ADD, SUBTRACT, MULTIPLY, DIVIDE or EXPONENTIATION, but also the operands, which can either be numbers or arrays and have a unique key. The TypeScript interface used for payload operations is showcased below.

```
interface OperationDTO {
    type: OperationType;
    operands: Operand[];
    resultType: OperandTypes;
}
```

Finally, the processing payload contains the public key necessary to perform computations, but may also contain certain specialized keys, used only for specific operations. For instance, Galois keys are used in order to compute the sum of an array's elements, while Relinearization keys are used within the exponentiation operation and after multiplication operations, in order to reduce the noise induced to the ciphertext by the processing. Due to their size and in order to reduce network traffic and redundant computations for all parties involved, the keys are generated only if the payload pipeline includes any of the previously specified operations.

The process of processing a chunk begins by parsing the JSON schema to a JavaScript object. Afterwards, the *FHEModule* specialized class is initialized with the FHE schema to be used, and the received public key is loaded. An operands and results map is then created, which may receive *CipherText* and *PlainText* Node SEAL class instances as values and has a string field key as keys. In order to populate this map, the encrypted data received, which corresponds to a CSV format file column, is loaded into a *CipherText* instance and added to the map, its field key being prefixed with the letter *d*, indicating its data-related role.

Afterwards, all operations are iterated over and, if any plaintext values are used, the corresponding values are loaded into arrays that match the length of the encrypted data and encoded into a *PlainText* instance. These instances are then added to the map, their field key being prefixed with the letter *p*. In the next step, the aforementioned specialized keys are loaded into their Node SEAL data structures.

Subsequently, the operation evaluation process may commence. A loop over the operations begins its execution, and a specific method of the *OperationsCalculator* class is called, providing its needed parameters.

The result of each operation is stored in the map, in order to allow further operations to use it as an operand, if they so desire.

After all operations have been evaluated, the last operation's result is extracted from the map and serialized to a Base64 string, to prepare it for its destination, the back-end system. Before this happens, however, a cleanup sequence is initiated, in which each *CipherText* and *PlainText* instance from within the operands and results map is deallocated, so as to free the memory that it occupies. Finally, the FHE module's deallocation method is also called, to reclaim the memory of any internal structures such as encoders, keys, or the encryption context itself.

After the end result of the processing has been obtained, it is sent to the server, together with the verification token. The server stores the output and marks the Data Processor on whose behalf processing was accomplished, in order to be able to offer them monetary recognition at a later date.

FHEModule

The *FHEModule* class contains logic pertaining to the encryption context, being able to generate keys, encode or decode values into Node SEAL compatible structures, but also to encrypt or decrypt data or ciphertext. It supports the usage of either the BFV FHE scheme in the case of integer values or the CKKS FHE scheme for fractional numbers.

OperationsCalculator

Finally, this last subcomponent contains the logic for processing operations, receiving input parameters and returning output *CipherText* values. The exposed interface is simple, as it is structured in the form of a collection of functions, one for each of the supported operations. The functions can automatically collect a variable number of operands, through the usage of a JavaScript specific rest parameter, and the exact way in which the received operands should be processed is locally determined through validations.

5.2. Front-end platform

The front-end platform directly interacts with the two main kinds of human actors. As such, it must comply with several criteria, including security, performance, coherence, and, last but not least, ease of use.

The main sub-modules of the front-end platform are authentication, account confirmation, payload management, API keys management, two-factor authentication settings, and payments account management.

The navigation structure diagram below showcases how these submodules are accessed and which roles each of them supports.

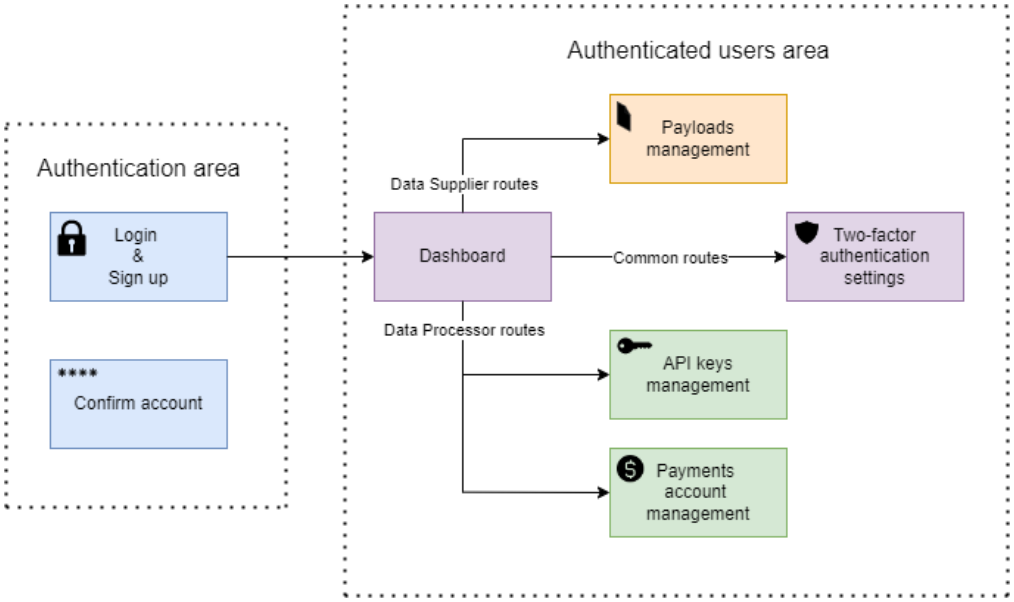


Figure 8. *Front-end platform navigation structure.*

The client-server communication takes place in HTTPS format and is authenticated using session cookies containing opaque tokens. Compared to some JSON Web Tokens (JWTs), these cookies are more secure, as they are bereft of any extractable credentials, containing only a randomly generated value that links the cookie to server-side stored data. Furthermore, terminating a malicious actor's session is easier when using session cookies, the only action necessary being that of removing the associated session's data from the server's cache. JWTs are stateless and therefore harder to invalidate since they are generated with a fixed expiration time. Thus, their cancellation would require the maintenance of a server-side data structure or another means, increasing complexity and reducing some of the benefits they provide. The key use case in which JWT usage would be advantageous is third-party service interoperability, and it could be a potential design choice in a computer system that requires such integrations.

Authentication

When users first access the web application, they can observe a card containing two tabs, allowing for login or registration. Within the registration view, they can create an account with a certain role, depending on their goals within the Breadwinner computer system. As such, they can either be a data supplier or a data processor. The sign-up process is simple and efficient, in order to retain users more easily, and it is presented in Appendix 1.

After signing up, they must confirm their account and choose a password by following the link sent to their provided email address. The confirmation link contains a special, large, randomly generated UUID token, which is submitted together with the desired password to the server, in order to assign the password to the user corresponding to the identified confirmation.

Once they login into the application, a user's path diverges depending on their role. The side navigation menu allows users to choose which view they want to interact with, presenting them with role-specific actions. The routing system implemented in the application's logic also

authorizes route navigation, ensuring that users have the required role for that route. If they do not, they are redirected to a default route.

Data payloads management

For data suppliers, the main type of action that they can perform is data payloads management. Within the main view, they can view a card-based list of payloads that they have defined, containing information such as the label of the payload, the number of chunks that form the data payload and the total number of rows that were included in the payload.

Depending on the state of a payload, a different action or indicator is shown in the center area of each payload card. When a payload's processing is in progress, a loading bar appears that indicates to the user the percentage of chunks that has been processed thus far. On the contrary, if a payload has been fully processed within the system, a decryption button appears.

A floating action button allows users to commence the data payload creation process. It is composed of four steps: uploading a .csv dataset, creating payloads and operation pipelines, confirming the submittal of the payloads and, ultimately, performing payment by using Stripe.

When the dialog window appears, a request goes out to the back end to check if the data supplier has a payment that is already in a pending state. If so, the process skips to the fourth step, in order to allow them to resume the payment of the previously configured data processing. Otherwise, the users may upload a .csv file, which allows them to proceed to the next step.

The second step involves creating payloads and configuring operation pipelines for them. Multiple payloads can be created based on the same dataset, each having a separate pipeline.

Below is a screenshot showcasing the configuration of a payload's operation pipeline. The pipeline is configured to compute the average salary of a list of teachers. The dataset's source is this repository: https://github.com/Apress/data-analysis-and-visualization-using-python. This data, if it were real, could be considered sensitive information, depending on other variables that would be included in the dataset. Thanks to fully homomorphic encryption, however, the data is handled safely.

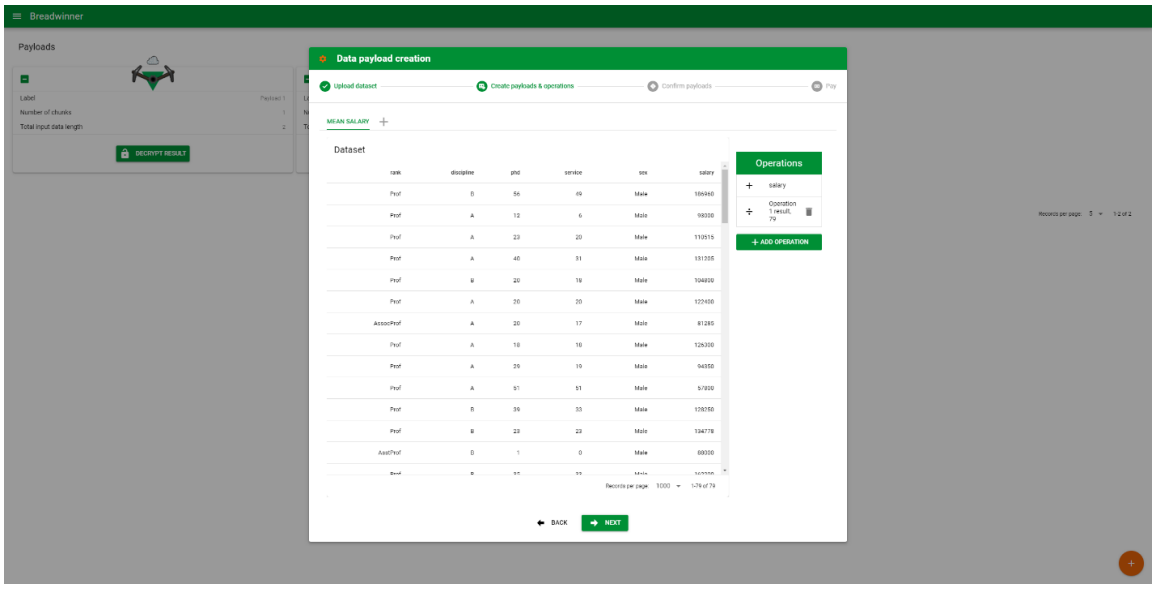


Figure 9. *Data supplier configuring data payload to compute a mean salary.*

By double clicking a payload tab name, its name was changed from the default to a more descriptive name – “Mean salary”.

Upon submission, payloads undergo a chunking process which splits them into smaller parts that are then encrypted. This procedure is crucial, as chunks can be sent one at a time to website visitors for processing. During encryption, the appropriate FHE scheme is chosen based on whether or not column values are whole numbers. After encryption, the platform bundles up the key pair used for encryption in JSON format and gives the user the possibility to download it, in order to use it when decrypting the final result. The key pair is not stored server-side.

In the final step, the user can pay for their processing via the external payment processor, Stripe. All payment details are handled by Stripe, ensuring safe handling and compliance.

After some time, when all data chunks have been processed, the data supplier can obtain the decrypted result of their payload by supplying the previously mentioned key pair. The decryption process then commences, and the result, either an aggregated value or an array of values, depending on the operations performed on the data, is given to the user in JSON format. The decryption source code is provided in Appendix 2.

API keys management

The main type of action available for data processors is the management of API keys. These tokens are crucial in the activation of the Breadwinner web module, and they must be correctly configured with the web domain on which the module is to be used. After creation, the user is presented with a dialog window from which they can retrieve the generated API key, as it is hashed server-side and its plaintext version can never be accessed again after closing said window.

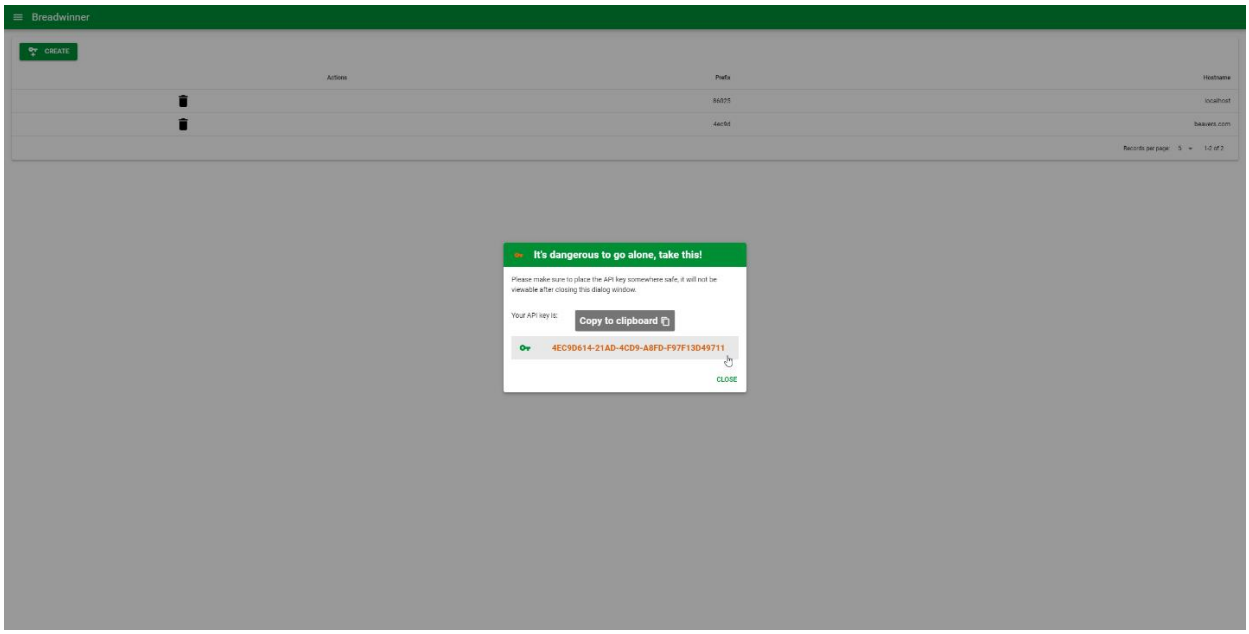


Figure 10. *API key creation*

API keys are displayed in a table, which allows data processors to view a prefix of the API key, in order to be able to identify distinct ones, but also the attached hostname of an API key. Finally, an action can be used in order to delete an API key from the system.

Payments account management

Data processors can also configure their external payments account with their payment and business details via Stripe, by accessing a URL provided by the server once their external account has been created. This is crucial in order to be able to receive weekly payments, depending on the number of chunks processed by their website. All payment related details are stored on Stripe's secure servers, the only field that is replicated within the back-end system's database is an account id, which is used when performing weekly payments to the corresponding data processor.

Two-factor authentication settings

An action that is role-agnostic is the management and setup of two-factor authentication. In this view, users can configure settings related to their account's two-factor authentication.

The content of the card presented to the user in the two-factor authentication management view is variable, depending on whether or not they have activated it previously. If previously activated, two-factor authentication can be deactivated at any point through the press of a button. Otherwise, the user is presented with a view with two steps. The first step allows them to request the issuance of a trial TOTP seed from the back-end server, and the second step requires them to scan a QR code which they can input within a third-party authenticator application.

After inputting their first valid code, two-factor authentication is activated for their account and required upon every login attempt. During the login process, if a user has activated two-factor authentication, after inputting the password they will see an additional dialog in which to input a TOTP, and they will be considered logged in only after inputting a valid one.

The screenshot below presents the two-factor authentication activation view.

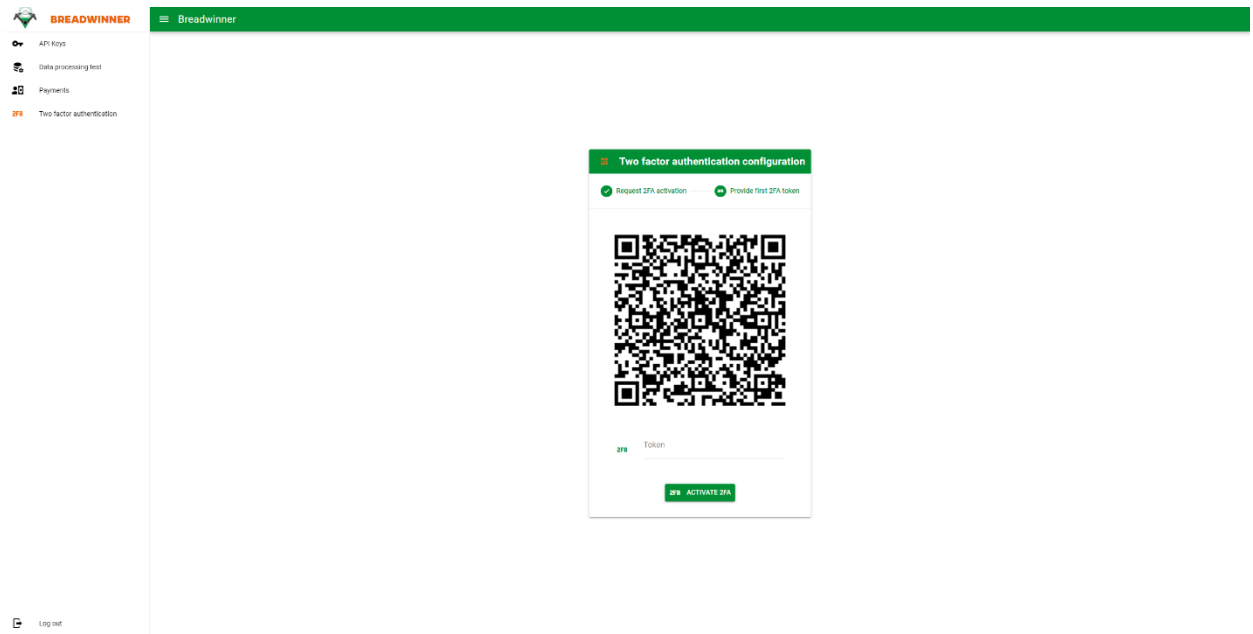


Figure 11. *Two factor authentication activation*

5.3. Back-end platform

The server interacts with both previous modules, representing the central link that bridges all the internal and external actors of the computer system.

Several libraries and extensions have been used in order to harden the system. For instance, the express-session library allows for cookie-based session management. In order to improve the performance and scalability of this approach, which is one of its most pronounced disadvantages when compared to a stateless alternative, an external Redis cache server was set up on an Ubuntu virtual machine.

Another means of safeguarding client-server communication is through the use of Cross-Site Request Forgery (CSRF) tokens, specifically through implementing the Synchronizer Token pattern recommended by OWASP for stateful software [19]. Tokens are generated server-side and sent to the user upon logging successfully, but also refreshed before state-altering POST and DELETE requests, and they must be provided when executing these requests, by including them in a designated header.

Database implementation

EdgeDB is the database of choice that resides on the back-end system. In order to communicate with it, the servers must use a query builder that allows for typed select, insert, update or delete operations.

Entities are defined using the EdgeDB Schema Definition Language (SDL). Compared to traditional relational database implementations, entities are described using an object syntax. Relationships between entities are represented using links. For instance, the link below is used within the definition of the Payload entity, in order to establish the relationship between payloads and chunks.

```
multi link chunks -> Chunk {
     constraint exclusive;
}
```

The 1:m cardinality of the relationship is specified by using the *multi* keyword. By specifying the exclusive constraint, a chunk can only be attached to a payload at a time, thus modelling the business logic requirement correctly.

While modeling a relationship, it is one-way by default, meaning that the chunks of a payload can easily be accessed by querying the appropriate property. However, finding the parent payload of a chunk requires a different approach.

To solve this issue, a virtual, special kind of link can be defined, named *backlink*. While backlinks are not represented in the database physically as a property, therefore not occupying additional storage space, they allow for the bidirectional retrieval of relationships that is required by some use cases. An example of a backlink that allows for the selection of a chunk's payload can be seen below.

```
link payload := .<chunks[is Payload];
```

Another useful functionality is that of computed properties. They allow for the reuse of querying logic and can be accessed as a property of the entity on which they are defined.

Moreover, they are lazily evaluated and thus do not incur performance penalties. A crucial way in which computed properties aid the Breadwinner back end is the retrieval of data processors that need to be paid. By using the computed property within a query, filtering the data processors which have passed the minimum number of processed chunks threshold for payouts is done on the database side, therefore eliminating the retrieval of redundant data. The computed *payable_chunks* property which resides on the *DataProcessor* entity is showcased below.

```
link payable_chunks := (
      select .chunks
      filter .paidFor = false
)
```

Communication with the front-end platform

One of the two main communication avenues with the outer world is done through HTTPS with the front-end platform. As such, the Express.js library is used in order to expose an API containing endpoints that can receive and relay information from and to the system's users.

By calling these endpoints through the front-end platform, authorized actors can asynchronously perform operations such as uploading and retrieving data payloads, creating API keys or revoking them, setting up their two-factor authentication, or configuring their payments account.

Several abstract classes of source code files exist, each with their own responsibility: controllers, services and repositories.

Endpoints are grouped into *controllers*, depending on the entity whose resources they allow access to. For instance, the *PaymentController* contains a HTTP POST method endpoint which allow for the creation of payments for the uploaded payloads, and a GET method endpoint that allows the front-end module to receive the Stripe Checkout payment session URL that a data supplier should access in order to finish paying for data processing.

The logic contained within controllers is minimal and related to routing and authorization via middleware, as the bulk of the business logic and functionalities is contained within *services.* In turn, they use *repositories*, which contain logic specifically related to the database layer in the form of CRUD data queries.

An important flow within the computer system that requires the cooperation of the front-end and back-end modules is the creation of data payloads.

When receiving one or several payloads from a data supplier, together with their operation pipeline schemas, they are saved within the database. The query builder allows for the seamless composition of queries, allowing for the simultaneous insertion of the payload's details, but also the bulk insertion of its chunks' details, and for the payload's data supplier to be selected and assigned to it. This composition allows for performant queries, as there is only one round trip necessary to the database, minimizing the latency caused by context switches.

The query which inserts the data payload is showcased below.

```
const savedPayload = await queryBuilder
            .insert(queryBuilder.Payload, {
                label: payloadDTO.label,
                jsonSchema:
queryBuilder.json(payloadDTO.jsonSchema),
                hasRelinKeys, hasGaloisKeys,
                dataSupplier:
queryBuilder.select(queryBuilder.DataSupplier, (dataSupplier) => ({
                    filter: queryBuilder.op(dataSupplier.id, '=',
queryBuilder.uuid(userId)),
                    id: true,
                })),
                chunks: queryBuilder.set(
                    ...payloadDTO.chunks.map((chunkDTO) =>
                        queryBuilder.insert(queryBuilder.Chunk, {
                            length: chunkDTO.length,
                            paidFor: false,
                            processed: false,
            }))),}).run(db);
```

After the database persist operation, the encrypted data chunks and keys that have been received from the front end must be saved locally on the server's filesystem. This approach is chosen instead of saving them within the database for multiple reasons. Storing the files in the database would lead to slower queries, as the table becomes bigger and more bloated. Moreover, storing the files on the filesystem could be more easily transitioned to a dedicated storage solution, such as a cloud storage bucket, by simply replacing the path of the file with the URL to its location. Finally, a crucial advantage is the fact that the database layer can focus on performing queries, its specialized tasks, while the server leverages the underlying filesystem for IO operations.

This approach of storing files is even more beneficial, thanks to the Node.js event loop. When chunks and keys are written to files, this is done through JavaScript *Promise* objects, in an asynchronous way. This allows for writing the files in parallel by blocking the function's execution on a *Promise.all* construct, which waits for all file *Promises* to be fulfilled i.e. for all files to be persisted. Additionally, the server is free to serve other requests while the filesystem is managing the writing process.

After receiving a response from the server, the client initiates the payment procedure by sending a request to the corresponding endpoint on the *PaymentController* back-end class. The server then retrieves all the payloads that belong to the user and have not been paid and creates a Stripe session and a payment entity. The session is configured using several options, such as the ways in which the payment could be done – only payment by card is allowed in this case, or the URLs to which Stripe should redirect the user upon payment. The session is also attributed an item with a quantity of one and a price computed by using the payload pricing formula showcased in section 3.2. Finally, some metadata is attached to the session in the form of the id of the first payload included in this payment.

After the client successfully performs the payment, Stripe sends an event to a webhook on the server through which it notifies the system of the successful processing of the payment. The event is signed, and its signature is verified using a webhook secret that the server possesses, in order to avoid impostors posing as Stripe. The server then updates the state of the payment that has been processed from *pending* to *paid*. This final step concludes the payload creation process and allows for the chunks of this payment's payloads to be distributed to web modules.

A potential alternative to using Stripe for handling user payments and payouts, which could be investigated in future work, would be that of using cryptocurrency and blockchain technology.

After all data chunks have been processed by the web modules, the data supplier can enter their payloads management page and request the decryption of their payload. This requires them to upload the key pair containing a private and public key, which was initially given to them upon uploading the payload.

The server then replies with the encrypted chunks, which are decrypted on the front-end platform by using the FHE submodule of the web module. Thus, no credentials leave the front-end client, and the decryption process is undergone safely.

Communication with the web modules

The second type of communication takes place between the server and Breadwinner web modules through WebSockets. The server manages an in-memory map of WebSocket sessions, which is populated when a web module correctly authenticates itself by supplying an API key matching the domain on which it is being used. The WebSocket upgrade process is highlighted in Appendix 3.

Through the WebSocket sessions, chunks are sent to web modules, together with a cryptographically secure UUID token that is verified upon reply, which partially protects the integrity of the system against trying to submit invalid or unauthorized data to the back-end system.

In turn, web modules respond with the processed versions of the chunks, which are stored by the server and retrieved when a data supplier wishes to decrypt the result. The limitation with this is that the computations performed on the output data are not verifiable non-interactively, without consuming the same amount of processing power that has gone into their processing. As highlighted in the literature review section 2.3., this is a topic of ongoing research.

Task scheduling

The back-end server runs two scheduled tasks periodically. The task scheduling is implemented using the *node-cron* package and uses the cron syntax for specifying the time at which tasks should be run.

The first scheduled task's role is to delete payloads whose payment has been left in a pending state, and that are at least one hour old. This task runs every hour, on the dot, and is defined using the following syntax:

```
// Every hour, on the dot.
cron.schedule('0 * * * *', async () => {
        ...
});
```

The task begins by retrieving the payments matching the criteria, together with their attached payloads and chunks. Furthermore, the Stripe session id stored on the payment object is used in order to request the expiration of that session. After a session has been expired, attempts to navigate to Stripe's website and to perform the payment will fail.

Finally, the associated chunk files are removed from the server's storage, and the corresponding payload and chunk entities are deleted within the database.

The second scheduled task relates to the payment of data processors for the chunks that they have processed. This task runs once every week, at midnight on Monday, as showcased below.

```
// Every week, at 00:00 on a Monday.
cron.schedule('0 * * * 1', async () => {
        ...
});
```

A database query is run, which identifies the data processors who have activated their Stripe accounts and have also passed the required number of unpaid processed data chunks. This threshold is computed by dividing the minimum supported Stripe payment value by the payment offered per chunk.

For each of the payable data processors that satisfy the criteria, their payable chunks are updated so as to be marked as paid, ensuring no duplicate payments for the same processing. Additionally, a Stripe transfer is created, in order to make a payment from the Breadwinner Stripe account to the data processor's account. The value of the payment is dependent upon the constant monetary value of a chunk, but also the number of processed chunks belonging to a data processor.

6. Conclusions

Fully homomorphic encryption is a worthy contender that can solve the ongoing opposition issue between data privacy and data processing within the domain of cloud computing. Through its unique property of enabling third parties to perform processing on behalf of the data owners, while not decrypting the data and imperiling its confidentiality, it paves the way for a more secure and operational Internet.

At the outset of this dissertation thesis, the prime idea proposed for further investigation was the technical viability of creating a distributed processing cloud platform.

To this end, a scalable computer system architecture was first envisioned. The scalability of the solution derives from its ability to integrate Internet devices as actors participating in the core processing loop, by embedding web browser compatible modules into websites. Leveraging the always-encrypted state of data which has been homomorphically encrypted, this approach uses the WebSocket protocol to transmit chunks of data to authenticated websites in order for them to be processed. All processing is done on a background thread, which negates the impact on the browsing experience for website visitors.

The system nourishes a mutually beneficial loop between the two kinds of human actors that directly interact with it. Data suppliers represent the entities who manifest the need to have their data processed according to a well-defined set of operations and specifications, which they can configure within the front-end platform. In return for a sum of money, their data is encrypted using fully homomorphic encryption, and securely passed on to the processing nodes of the system. When the data output is ready to be accessed, data suppliers are the only actors who can decrypt and benefit from the plain output, since they are the only party that ever has the private key in their possession.

In order to be able to fulfill these needs, the system recruits website owners, referred to as data processors. Their role is to provide the processing infrastructure of the system, by embedding the web module into their websites, thus allowing the system access to their website visitors' devices. In return for this, data processors are rewarded by the system, being paid money for each chunk processed through their website. As such, outsourced processing is presented as an alternative monetization source to advertisements.

Throughout the implementation of the system, best practices have been followed so as to ensure the utmost security for all parties involved. This was accomplished through measures such as strict authentication and authorization procedures, the usage of cookies and tokens for securing sessions and data transmission between clients and the back-end system, sensitive credential protection through hashing and salting and, last but not least, two factor authentication integration.

A limitation of the computer system is the inability to verify the correct transformation of inputs into outputs through homomorphic operations on the encrypted data without the need for complete re-computation by the verifying party, which would render the previous processing redundant. This issue, coined verifiable computation, is a topic that is still being addressed within the domain and literature, and a research paper which details a possible way in which this could be remedied was briefly presented in the literature review section of this thesis.

While this thesis's goal was to assess the viability of a technical implementation for a secure distributed processing system based on fully homomorphic encryption, the financial feasibility of such a project depends on several variables, such as hosting and storage prices, but also the ability to compete with the alternative monetization scheme, namely advertisements. The computer system was, however, integrated with a payments platform for practical reasons, in order to showcase how its monetary side could be managed and automated. Thus, the economic implementation of this system would require robust investigation and market research, making it ideal to be tackled in future research.

As fully homomorphic encryption is a nascent technology, its high storage and computational effort requirements make it multiple factors slower than operating on plaintext data, impeding its actual applicability in real-life scenarios. However, as has been highlighted in the early sections of this thesis, efforts are underway to remedy the performance gaps, through the creation of new, specialized hardware able to more efficiently perform homomorphic operations. At the same time, new generations of algorithms are constantly being iterated and refined upon, tackling existing security vulnerabilities while also improving speed metrics. Moreover, the security and usability of homomorphic encryption in a world which has access to quantum computing remains to be addressed, both theoretically and practically.

While the computer system envisioned and developed during the course of this dissertation thesis is configurable and could be used and deployed in its current state, boundless upgrades could be made, and several alternative design decisions could be taken into consideration.

For instance, a central characteristic of fully homomorphic encryption which is also present in this system is the large size of encrypted outputs and cryptographic keys used for performing operations. Depending on the parameters used by the system, each of the aforementioned objects could exceed storage values upwards of hundreds of megabytes. The introduction of a Web 3.0 storage solution, the Interplanetary File System (IPFS), could be a potential means of offloading this data, as it is stored in an encrypted state and would not be decryptable by nodes which replicate it.

Another upgrade which could be made to the system would be the migration to an authentication scheme which does not require the usage of passwords. One such mechanism is Fast Identity Online (FIDO), a technique based on public-key cryptography, which is contemporarily being discussed and standardized.

In conclusion, this dissertation thesis has proven the technical feasibility of a distributed outsourced processing computer system, by presenting a highly scalable architecture based on Internet devices which are authenticated and offer their processing capabilities to the computer system through a lightweight, web-compatible module. An interdisciplinary study which analyzes the economic lucrativeness of such a project is a topic that is left as future work.

Bibliography

[1] R. L. Rivest, L. Adleman and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation,* 1978.

[2] D. Boneh, E.-J. Goh and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," in *Second Theory of Cryptography Conference*, Cambridge, Massachusetts, 2005.

[3] C. Gentry, "A fully homomorphic encryption scheme," 2009.

[4] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," 2012. [Online]. Available: https://ia.cr/2012/144. [Accessed June 2022].

[5] J. H. Cheon, A. Kim, M. Kim and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *23rd International Conference on the Theory and Applications of Cryptology and Information Security*, Hong Kong, 2017.

[6] Q. Wang, D. Zhou and Y. Li, "Secure Outsourced Calculations With Homomorphic Encryption," *Advanced Computing: An International Journal (ACIJ),* 2018.

[7] X. Liu, R. H. Deng, P. Wu and Y. Yang, "Lightning-fast and privacy-preserving outsourced computation in the cloud," *Cybersecurity,* 2020.

[8] F. Costa, L. Silva and M. Dahlin, "Volunteer Cloud Computing: MapReduce over the Internet," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Anchorage, 2011.

[9] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert and D. Sanchez, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption (Extended Version)," *Computing Research Repository,* 2021.

[10] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon and Y. Lee, "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems,* 2021.

[11] C. Ganesh, A. Nitulescu and E. Soria-Vazquez, "Rinocchio: SNARKs for Ring Arithmetic," 2021. [Online]. Available: https://ia.cr/2021/322. [Accessed April 2022].

[12] H. Zhu, L. Wang and C. Wang, "Privacy-Enhanced Multi-User Quantum Private Data Query Using Partial Quantum Homomorphic Encryption," *International Journal of Theoretical Physics,* 2021.

[13] Morfix Inc., "Node SEAL Documentation," 2022. [Online]. Available: https://docs.morfix.io/. [Accessed 2022].

[14] E. You, "Vue.js Documentation," 2022. [Online]. Available: https://vuejs.org/guide. [Accessed 2022].

[15] OpenJS Foundation, "Node.js Documentation," 2022. [Online]. Available: https://nodejs.org/api/documentation.html. [Accessed 2022].

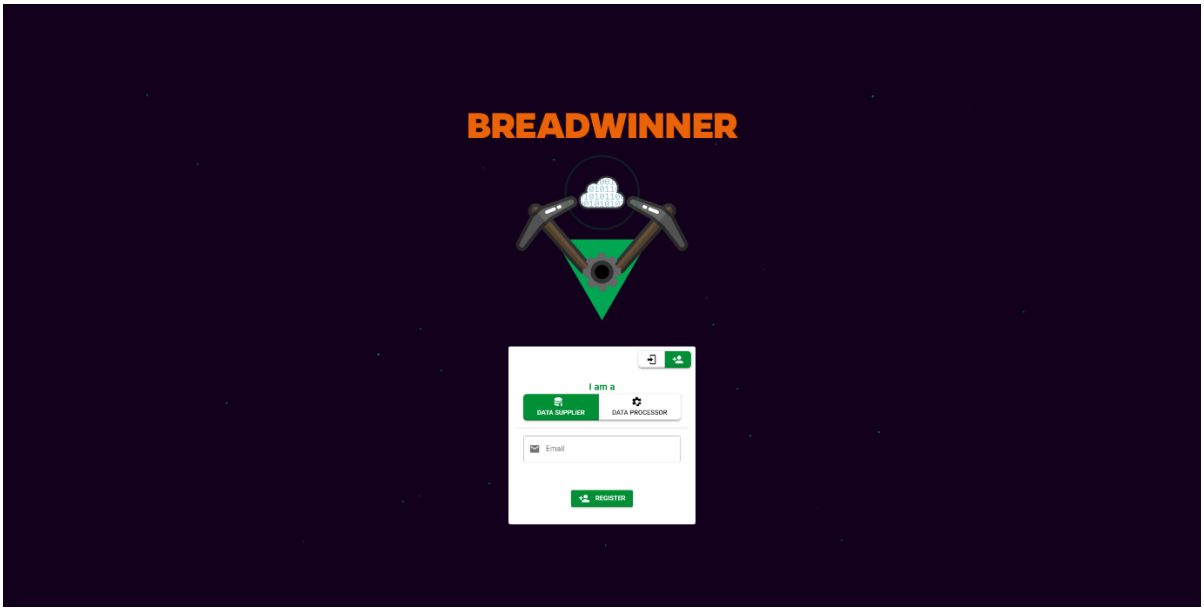
[16] Redis Ltd., "Redis Documentation," 2022. [Online]. Available: https://redis.io/docs/. [Accessed 2022].

[17] OWASP Foundation, "Password Storage Cheat Sheet," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. [Accessed 2022].

[18] EdgeDB Inc., "EdgeDB Presentation," 2022. [Online]. Available: https://www.edgedb.com/. [Accessed 2022].

[19] OWASP Foundation, "CSRF Vulnerability Prevention," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html. [Accessed 2022].

Appendix 1 – Breadwinner signing up process



Appendix 2 – Decrypting data on the front end

```
// Fetch some info about payload in order to complete the processing
const response = await getPayloadDecryptInfo(
      this.userDetails.roleSpecificId,
      this.payloadToDecryptId
      );
const payloadDecryptInfo = response.data;

// In the case of an array type end result, we should only concatenate the
resulting arrays from each chunk that was processed.
// In the case of a number type end result, we must sum all the numbers and
divide by the number of chunks.
this.keyPair = JSON.parse(await this.uploadedKeyPairFile.text());
await FHEModule.initFHEContext(payloadDecryptInfo.schemeType);
FHEModule.setKeyPair(this.keyPair);

let plainTextResult = null as null | number | Array<number>;

switch (payloadDecryptInfo.endResultType) {
      case OperandTypes.ARRAY: {
            console.log('End result type is an array');
            plainTextResult = [];

            for (const chunk of payloadDecryptInfo.chunks) {
               const response = await getProcessedChunkOutput(chunk.id);
               const cipherTextArray = FHEModule.seal!.CipherText();
               cipherTextArray.load(FHEModule.context!, response.data);
               const decryptedArray = FHEModule.decryptData
                                                  (cipherTextArray)
                                                .slice(0, chunk.length);
               cipherTextArray.delete();

               plainTextResult.push(...decryptedArray);
            }
            break;
       }
       case OperandTypes.NUMBER: {
            console.log('End result type is a number');
            plainTextResult = 0;

            for (const chunk of payloadDecryptInfo.chunks) {
               const response = await getProcessedChunkOutput(chunk.id);
               const cipherTextArray = FHEModule.seal!.CipherText();
               cipherTextArray.load(FHEModule.context!, response.data);
               const decryptedArray = FHEModule.decryptData(cipherTextArray);
               const decryptedNumber = decryptedArray[0];
```

```
                cipherTextArray.delete();
                plainTextResult += decryptedNumber;
            }
            break;
        }
}

console.log('final result', plainTextResult);
FHEModule.deallocate();
```

Appendix 3 – WebSocket upgrade request handling on server

```
httpsServer.on('upgrade', async function upgrade(request, socket, head) {
    try {
        const apiKey = request.headers['sec-websocket-protocol'];

        if (!apiKey || Array.isArray(apiKey)) {
            throw new NotFoundError('Received invalid API key');
        }

        if (request.headers.host) {
             const dataProcessorId = await checkAPIKeyValid(
                apiKey as string,
                request.headers.host.substring(0,
request.headers.host.indexOf(':'))
            );

            wss.handleUpgrade(request, socket, head, function done(ws) {
                socketMap.set(ws, {
                    dataProcessorId,
                    payloadToken: '',
                });
                wss.emit('connection', ws, request);
            });
        } else {
            throw new NotFoundError('Host header not present.');
        }
    } catch (error) {
        console.log(error);
        if (error instanceof NotFoundError) {
            socket.write(error.message);
        }
        socket.destroy();
    }
});
```