University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Extension and Standardization of a Blockchain Interoperability API

Pascal Kiechl
Zürich, Switzerland
Student ID: 16-927-998

Supervisor: Eder Scheid, Christian Killer
Date of Submission: April 12, 2021

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Mit der Einführung des Konzepts der Blockchain (BC) und der anschliessenden Gründung von Bitcoin im Jahre 2009 hat ein Trend einer schnell wachsenden Anzahl alternativer BC-Implementierungen und -Plattformen begonnen. Aufgrund ihrer inhärenten Unterschiede sind verschiedene BCs in der Regel nicht in der Lage, weder ihre nativen Währungen noch die auf ihnen gespeicherten Daten auszutauschen. Daher wurden zahlreiche Interoperabilitätslösungen enworfen, darunter Bifröst, eine auf einem Notarschema basierende Interoperabilitäts-API. Die Prototyp-Implementierung von Bifröst funktionierte wie vorgesehen, wies jedoch eine Reihe von Schlüsselbereichen auf, in denen Verbesserungen möglich waren. In dieser Arbeit wurden neue Funktionalitäten hinzugefügt, um die an Bifröst gestellten Kernanforderungen verstärkt zu erreichen. Optionale Datenverschlüsselung wurde integriert, um den Benutzern mehr Kontrolle zu geben und die Sicherheit gespeicherter vertraulicher Informationen zu erhöhen, allerdings auf Kosten erhöhter Datengrösse, sobald Verschlüsselung angewendet wird. Die Möglichkeit, übergrosse Daten aufzuteilen, mit mehreren Transaktionen zu speichern und für die Rekonstruktion beim Abruf die einzelnen Teile ordnungsgemäss zu verfolgen, wurde implementiert, was die Flexibilität der speicherbaren Daten erhöht. Ausserdem wurde die Benutzerfreundlichkeit durch generische Fehlerbehandlung verbessert, was in Verbindung mit der Redundanz-Funktion die Robustheit von Bifröst erhöht. Es wurden Untersuchungen zu standardisierten Interoperabilitätsformaten durchgeführt, die als Inspiration für ein neues JSON-Schema dienten, welches Interaktion mit BC Interoperabilitäts-APIs wie Bifröst standardisiert. Schliesslich wurden Mittel zur sicheren Verwahrung privater Schlüssel untersucht, und obwohl letztlich keine sofort umsetzbare Lösung gefunden wurde, wurden die Optionen für zukünftige Entwicklungen von Bifröst in diesem Bereich geklärt.

With the introduction of the concept of blockchain (BC) and the subsequent launch of Bitcoin in 2009, a trend of rapidly increasing numbers of alternative BC implementations and platforms has begun. Due to their inherent differences, different BCs are generally incapable of exchanging neither their native currencies nor the data stored on them. Thus, numerous interoperability solutions have been envisioned, with Bifröst, a notary-scheme based interoperability API, being one of them. Bifröst's prototype implementation, though working as intended, had a number of key-areas where improvements were possible, thus, in this thesis, new features have been added to strengthen the core requirements imposed on Bifröst. Optional data encryption has been incorporated to give users more control and to increase the security of stored confidential information, at the cost of increased data sizes when encrypted. The capability to have oversized data split, stored with multiple transactions and tracked properly for reassembly on retrieval has been included, increasing the flexibility of the data that can be stored. Furthermore, ease

of use has been improved with the addition of generic error handling, whilst at the same time, in conjunction with redundancy, increasing the robustness of Bifröst. Research on standardized interoperability formats has been conducted and has served as inspiration for a new JSON scheme for interacting with BC interoperability APIs such as Bifröst. Finally, means to securely manage private keys have been investigated, and although ultimately no immediately actionable solution was found, the options for future developments of Bifröst in that area have been clarified.

Acknowledgments

I want to express my gratitude towards my supervisors, Eder Scheid and Christian Killer, for their agreement to act as such for this thesis. Eder Scheid in particular has been instrumental throughout the entire duration of this project, from defining the initial specification of the thesis all the way to the final presentation, providing insight and feedback along the way.

In addition to that, I want to thank Timo Hegnauer, for the excellent prototype implementation of Bifröst on which this work fundamentally relied on, as well as Fabian Küffer for proofreading parts of the thesis.

Finally, a big thank you to Prof. Dr. Burkhard Stiller and the Communication Systems Group (CSG) at the University of Zurich for providing me with the opportunity to write my thesis at their research group.

Contents

Chapter 1

Introduction

The world was first introduced to the concept of a blockchain (BC) in 2008 when the Bitcoin white paper was anonymously published under the pseudonym "Satoshi Nakamoto", which subsequently led to the creation of the Bitcoin "Genesis block" in early 2009 [15]. The launch of Bitcoin inspired the creation of numerous other BC platforms, some of which expanded the scope of their platforms beyond that of cryptocurrencies to enabling features such as *e.g.,* support for Smart Contracts (SC) [48].

With different BC platforms following different BC protocols and carrying different cryptocurrencies, the result are BC platforms whose respective cryptocurrencies and stored data cannot be exchanged directly [48]. BC interoperability is the area of research that deals with this issue, by enabling connections between different BCs in order for them to be able to access or modify the state of the connected BCs [48].

One such interoperability project is the interoperability Application Programming Interface (API) called "Bifröst", which aims to present users (developers) with a flexible, modular and easy to use interface, enabling them to store and retrieve data on a number of different BCs, without the need for them to be aware of the implementation details of the underlying BCs [48]. The Bifröst prototype has been developed by the Communication Systems Group (CSG) at the University of Zurich.

1.1 Motivation

Whilst the Bifröst prototype is functional as-is, there are a number of aspects that can be improved upon:

- All data is currently stored unencrypted on the selected BC. Giving users the option to instead have their data be encrypted first before it is stored on the BC would increase the privacy of the data.
- At current, the API does not verify whether the data a user wants to store on a BC can be stored within a single transaction (TX) or exceeds the maximal size and thus needs to be split over multiple TXs.

- The API at current is missing generic error handling.
- The management of addresses and private keys.

Additionally, there are more research-intensive aspects that also pose interesting research challenges:

- Identifying the necessary functions a BC interoperability API like Bifröst has to provide to handle all identifiable use-cases of the API in a BC platform agnostic fashion. In conjunction, a standard format (*e.g.,* in JSON) should be proposed, defining how data is sent to and received from Bifröst.
- Currently, private keys are stored in plain-text, rendering them vulnerable to attacks. Here, research can be conducted to identify ways to improve the security of the private key management.

Thus, this thesis presents the improvements performed in the existing Bifröst prototype implementation to tackle the aforementioned issues. Specifically, this thesis details the design and implementation of features for *(i)* user data encryption to ensure the privacy of their information, *(ii)* splitting data up onto multiple transactions to circumvent transaction data size limits, *(iii)* generic error handling and *(iv)* redundancy.

Furthermore, the thesis presents the insights gained from the research conducted into existing standardization efforts with regards to interactions with BC interoperability APIs and then proposes a JSON scheme for such interactions that is valid for both Bifröst as well as other similar APIs.

In addition, the thesis outlines the findings stemming from research into secure private key storage and explains how the different approaches are or are not applicable to Bifröst's current credentials architecture, as well as how a change in credentials architecture might help improving the private key security.

An evaluation of the implemented features is put forth, highlighting the impact they present on Bifröst's key requirements, as well as the practical validity of the chosen symmetric encryption scheme with regard to its performance, whilst exploring a solution for its impact on data size.

1.2 Thesis Outline

After the introductory chapter, the needed theoretical background information about BCs, BC interoperability and Bifröst is presented in Chapter 2. This is followed by Chapter 3, which discusses related work in form of other projects that deal with BC interoperability. Chapter 4 outlines both the changes made to the Bifröst prototype during the thesis, as well as the research conducted on the topic of standardisation. In Chapter 5, the evaluation procedure of the newly implemented or improved features are documented, alongside the evaluation results. Additionally, the findings regarding secure private key management are discussed. Finally, in Chapter 6, a summary of the thesis and possible directions for future work are presented.

Chapter 2

Background

In this part of the thesis, the theoretical background material necessary to understand Bifröst, BC and BC interoperability is discussed. The aim of this section is not to be exhaustive, but rather to provide the context within which this thesis is situated.

2.1 Blockchain

Fundamentally speaking, a BC is a so-called "distributed ledger" that, due to its nature, has certain properties, notably immutability, meaning any data stored on the BC is quasi-impossible to be mutated [5]. Additionally, BCs are decentralized, meaning they do not rely on a central authority to establish a "single version of truth" [5]. Consequently, BCs are also-called "trustless", which alludes to the fact that as a participant in a BC, neither trust in any other single participant, nor in a central entity, is required to trust the information held on a BC [60].

The concept of a BC first appeared in 2008, when an anonymous author under the pseudonym of "Satoshi Nakamoto" published the Bitcoin white paper with the title "Bitcoin: A Peer-To-Peer Electronic Cash System" [40, 15]. The "Genesis block" of Bitcoin was created a few months later [15]. Since then, BCs have become a subject of research for numerous companies [5].

2.1.1 Blockchain Technology

In [8], BC is defined as a "Peer-to-Peer" (P2P) distributed ledger in form of a chain of blocks, with a consensus mechanism deciding which blocks are to be appended. In order for this definition to make sense, a number of fundamental concepts and mechanics need to be introduced, which is the main focus of this section.

Peer-to-Peer (P2P) Network

The term P2P refers to the fact that in such a network, the peers, *i.e.,* the individual computers participating in the network can communicate directly with each other, without requiring specialized servers [45]. Such a network topology comes with a number of benefits like increased network connectivity, large amounts of easily available computing power and storage capacity as well as low costs [45]. In the context of BCs, the terms "peer" and "node" can be used interchangeably [8]. For the rest of this document, computers participating in a BC network will be referred to as "nodes".

Blocks and Transactions

Each block in a BC consists of a number of components: a number of transactions, a timestamp, a nonce, which is a random number used for the purpose of hash verification, as well as the hash of the previous block [41]. This results in each block being connected to the previous one, resulting in the chain of blocks mentioned at the start of this section [41]. Figure 2.1 illustrates how such blocks are connected [41]. Any changes to any of the blocks result in the hash of that block changing [41]. By appending new blocks to the BC, the already present blocks get more secure, resulting in them being "tamper resistant" [80].

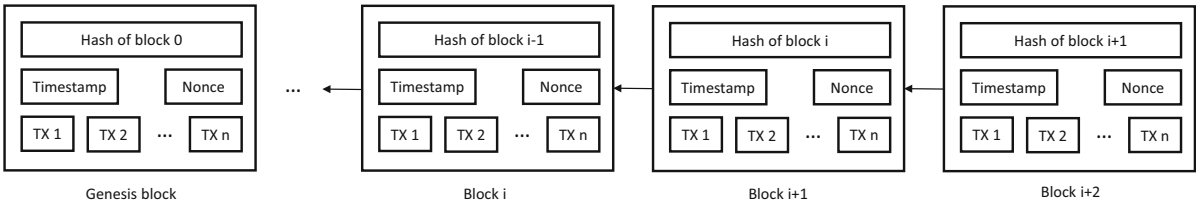


Figure 2.1: Chain of blocks [41]

A transaction is an operation on a digital asset which is included on the BC [8]. A digital asset can be different information, ranging from funds over information to services [8]. Generally speaking, a transaction results in data flow in the BC [8]. Constructing a transaction requires the specification of a number of properties, notably the destination of the operation, such that the aforementioned data flow can reach its destination [8].

Finally, before the transaction can be propagated across the BC network, it needs to be signed by the sender [8]. The signatures are created using asymmetric cryptography, meaning a pair of keys is used, with one being private to the signer, used to create the signature and with the other key being publicly available, used to verify the signature, which in turn allows the BC network to verify the authenticity of the transaction [76]. Such transactions are then propagated across the BC network and included in blocks by miners [8].

Nodes and Mining

Nodes are computers storing and maintaining a copy of the distributed ledger [8]. Nodes that, in addition to simply interacting with the BC network, are also involved in the

creation and validation of new blocks are called “mining nodes” or “miners” [1]. New blocks are created by miners by aggregating valid, yet currently unconfirmed, transactions into a “candidate block” [1]. After a candidate block has been created by a miner, it needs to be validated according to the consensus mechanism of the given BC [8]. Only once the candidate block has been validated can it be considered a valid block and can now be appended to the BC [8].

The validated block then needs to be propagated across the BC network, such that the other nodes in the BC network can update their respective copies of the BC [8]. Finally, the block needs to be confirmed, meaning all the nodes in the BC network have agreed on including it in the final version of the BC [8]. Once the block is confirmed, the transactions in the block are confirmed for a second time (the first time was when they were declared valid and included in the block by miner) [5].

Each time a new block is appended, the previous blocks, as well as the transactions contained within are reconfirmed [5]. Figure 2.2 depicts illustrates the sequence of stages that a transaction must traverse until it is confirmed, from its creation to the confirmation of a block.

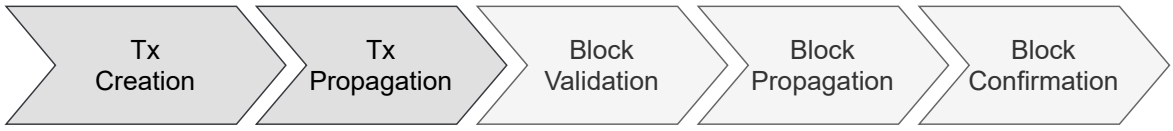


Figure 2.2: Transaction processing stages. Adapted from [8]

Transactions require a given number of accumulated confirmations before they can be considered final because the transaction might be included in a BC fork that will be pruned. For example, in the case of Bitcoin, a transaction needs to be confirmed 6 times to be seen as final [5]. Ethereum transactions meanwhile require 12 such confirmations [78]. The higher this number is, the less likely it is that a transaction will be reverted. The incentives for miners to partake in the mining process can differ from BC to BC, with public BCs often relying on a combination of block rewards and transaction fees [49].

Forks

Due to the decentralized nature of BCs, the copies held at the different nodes participating in the BC network may not always be equal [1]. For example, during the process of propagating a block across the BC network, it may be received by different nodes at different times [1]. Furthermore, multiple miners may broadcast blocks holding different transactions at almost the same time, resulting in a so-called “fork” [74].

This inconsistency between the different copies of the BC is then resolved by reaching a consensus among the nodes in the network on which version of the chain is the correct one [74]. The other versions that resulted from the fork are declared invalid [74]. An example of a BC with forks can be seen in Figure 2.3, where blue-filled squares means the BC and orange-filled squares represent the BC forks.

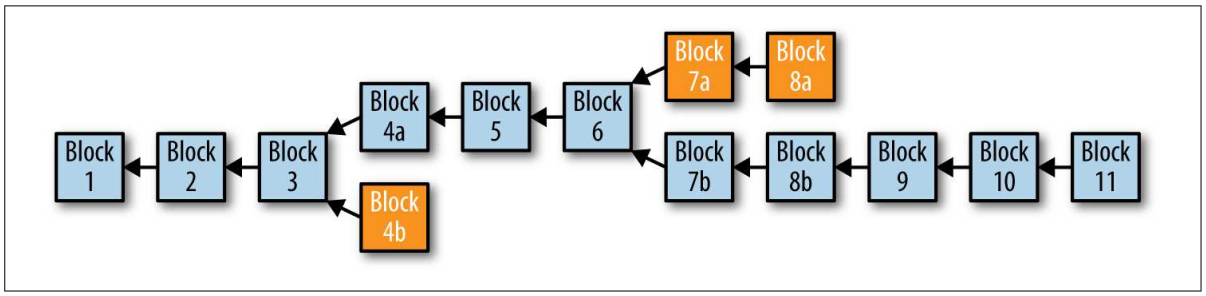


Figure 2.3: Blockchain with forks [1].

In the example of Bitcoin, the consensus on which version of the BC is valid, is reached based on which version is the longest, meaning it has the most amount of Proof-of-Work (PoW) put into it [1]. Assuming all nodes perform the selection according to this criterion, then there will ultimately be a globally consistent BC [1].

Consensus Mechanisms

At its core, a consensus mechanism is responsible for ensuring that each node participating in a BC network maintains the state of its copy of the BC correctly [49]. There exist a variety of consensus mechanisms, employed by different BCs [49]. The type of consensus mechanism running on a particular BC has a number of implications for the performance of that BC regarding scalability, speed of consensus finality, data consistency, latency for transaction confirmation, processing throughput, as well as computing power [76]. The next paragraphs provide an overview over two examples of such consensus mechanisms, Proof-of-Work (Pow) and Proof-of-Stake (PoS).

Proof-of-Work (PoW), originally introduced by Bitcoin, is the most widely used consensus mechanism. The idea is that each node uses its computing power to solve PoW puzzles, in order to construct blocks and participate in the mining process [22]. In [22] this is referred to as nodes essentially voting with their computing power. Notably, such a PoW puzzle is designed such that it is difficult to solve, yet once solved the result is easy to verify [37].

When a miner manages to solve such a PoW puzzle, he can then create a new block and forward it to the BC network, whilst the other nodes receiving the new block verify whether the included PoW fulfils the conditions imposed by the PoW puzzle [22]. If the PoW included in the block is valid, and the block itself is valid as well according to the criteria imposed by the BC in question, then the block is accepted and propagated further across the BC network [1]. Otherwise it is rejected and not forwarded further, ensuring that only valid blocks, including a valid PoW, are propagated [1].

With PoW, nodes vote with their computing power, whereas with ***Proof-of-Stake*** (PoS) their voting power is dictated by how much stake they own on the BC [22]. In effect, this means that a miner holding $X\%$ of coins on a BC with PoS can mine $X\%$ of the blocks on that BC [37]. As such, PoS can alleviate some of the issues that come with PoW, *e.g.,* the high power consumption resulting from the computing power required to solve PoW

puzzles [37]. Furthermore, as [8] show, PoS remains functional with a higher amount of all nodes controlled by an adversary than PoW and boasts better transaction throughput.

PoS does suffer from the "nothing at stake" problem, where miners are incentivized to mine on both forks, should a fork occur [11]. There exist however variants of PoS that are able to mitigate that issue [8].

2.1.2 Types of Blockchains

BCs can be categorized according to their "deployment type", which characterizes the read/write-permissions of a given BC. In this sense, the deployment type captures differences along these two dimensions, with the options "public" and "private" relating to the read-dimension and dictating which nodes have access to which information. Similarly, the options "permissionless" and "permissioned" indicate which nodes have which write-permissions, and therefore have the rights to append to the BC [49]. All in all, this leads to four possible combinations as indicated below:

- ***Public permissionless BC:*** Each node participating in such a BC has read/write access [49]. Sometimes also referred to as "classical" BC [8].
- ***Public permissioned BC:*** Each node in the BC network has reading access, but the ability to write to the BC is only granted to selected nodes [49].
- ***Private permissionless BC:*** Read/write capabilities are given to selected nodes in a closed network [49].
- ***Private permissioned BC:*** Read/write capabilities are granted by a central authority [49].

For example, Bitcoin is a "classical" BC, meaning it is categorized as a public permissionless BC in terms of its deployment type [8]. Hyperledger Fabric in contrast is an example for a private permissioned BC [31]

2.1.3 Smart Contracts

A Smart Contract (SC) is a computer program that executes a set of predefined actions, once a set of predefined conditions are met [8]. In the context of BCs, a SC is a transaction that holds instructions, which execute automatically given the necessary conditions are fulfilled [60]. Note that the decision whether or not a given condition is met may also be based on information from outside the BC, which the SC may access through the use of an "oracle" [60].

What sets a SC apart from a traditional contract is that it does not require the involved parties to trust that the other parties will uphold their part of the bargain [60]. SCs remove the need for such trust, since the code of the SC both defines the terms of the agreement

between the parties, as well as enforces the agreement by executing automatically in an autonomous fashion [60].

SCs are characterized by three properties [60]:

1. ***Autonomy:*** Once a SC is deployed, there is no further interaction required between the SC and its initiator [60].
2. ***Self-sufficiency:*** A SC can manage its own resources by providing services against payment in funds and by expending funds on needed resources like processing power [60].
3. ***Decentralization:*** A SC is deployed on a BC and as such does not run an a central server, but instead are distributed on the nodes of the BC network [60].

An example for a SC might be an automatic payment for a specific good at a predefined price, such that the SC would automatically perform the payment once the good reaches the desired price [60].

2.1.4 Applications of Blockchain Technology

[60] draws a distinction between three different categories of BCs. The distinction is made according to the type of activities supported by the different categories of BCs, with "Blockchain 1.0" being focused on applications of the BC technology related to cryptocurrencies, "Blockchain 2.0" relating to contracts and "Blockchain 3.0" encompassing applications that go beyond finance and contracts, thus relate for example to the areas of government or culture [60].

Blockchain 1.0

BC 1.0 refers to the applications of the BC technology which relate to the concept of cryptocurrency [60]. This includes the decentralization of digital payment systems, which enable transactions that can be made with greatly reduced payment fees and can be received faster compared to traditional payment systems such as credit cards [60].

Blockchain 2.0

BC 2.0 according to [60] extends the the principle of decentralization from cryptocurrencies to markets in a more broad sense. Some of the areas of application in BC 2.0 include the interfacing of cryptocurrencies into the traditional banking and financial services, decentralized crowdfunding platforms, SCs and smart property, the concept of keeping a register and facilitating exchange of all sorts of assets, from information over health data to physical property [60].

Blockchain 3.0

With the internet creating connections between all humans, and serving as the underlying network for BC technology, BC 3.0 aims to take the model of decentralization yet another step further than BC 2.0 by applying advantages and possibilities of the BC technology to any kind of industry [60]. BC 3.0 could be used to facilitate transactions, resistance to censorship and transparency in politically restrictive countries [60]. On the issue of censorship, an exemplary project is "Namecoin", a decentralized domain name system with the aim to prevent centralized authorities, such as countries, to assume control of top level domains, thus preventing such authorities from seizing and redirecting URLs controlling the dissemination of information [60]. Namecoin essentially uses BC technology to implement a mechanism for free speech [60]. Further examples of BC 3.0 applications include decentralized digital identify verification services and decentralized governance services [60].

2.2 Blockchain Interoperability

As the number of BC implementations increases, the need for communication between them becomes crucial to avoid the creation of BC islands (*i.e.,* BC-based applications not exchanging data), as the technologies, protocols and functionalities of different BCs tend to not be implemented with such communication in mind [50]. Thus, BC interoperability is needed to facilitate communication between BCs for actions such as *e.g.,* cross-BC token transfers [50]. There exist three main methods of achieving such BC interoperability, namely notary schemes, sidechains and hash-locking [12].

2.2.1 Notary Scheme

In notary schemes, a entity or set of entities, trusted by all involved BCs, assumes the responsibility of verifying whether a given event happened on BC A or whether a given statement about BC A holds true and to then forward this information to BC B. The nature of such trusted entities may be active, meaning that they act automatically based on events happening on the involved BCs, or alternatively reactive, forwarding information only when requested [12].

The strength of notary schemes lies in the fact that they do not need any changes to the implementations of the underlying BCs, between which they facilitate communication, and as such, are technically simple to implement [12, 44, 48]. The disadvantage of using notary schemes is that they rely on the involvement of trusted third parties [44, 48].

2.2.2 Sidechains

In [4], it is defined sidechains as a BC that validates data from other BCs. This is achieved by using so-called Simplified Payment Verification (SPV), which allows for the verification

of whether a given transaction has occurred on a BC, without being required to download the entire BC.

To allow for the exchange of assets between BCs, the so-called "two-way peg mechanism" is used, pegging one sidechain to another one, resulting in a parent chain and a sidechain. This then allows one of the BCs to lock assets in a special output, resulting in an SPV proof, which allows for the creation of assets on the other BC according to some exchange rate [4]. It is worth noting that this two-way peg commonly allows for asset transfers in both directions [44].

Sidechains, in contrast to notary schemes, do not rely on trusted entities and as such enable trustless interoperability between BCs [44]. However, sidechains increase complexity and are required to support SCs, in order to process SPV proofs [44, 4]. Furthermore, there is high latency associated with asset transfers via sidechains, as there are both a confirmation period, as well as a contest period required to ensure resistance against denial of service attacks and to prevent double-spending [44].

2.2.3 Hash-Locking

The technique of hash-locking allows for cross-chain atomic operations, enabling exchanges between BCs [12]. Atomic in this context means that either the exchange happens in both directions or none of them [48]. Assuming a situation where two parties A and B want to make an exchange of assets from different BCs, the mechanism can be divided into the following five steps [48]:

1. Party A generates a key s, computes the hash of the key (`hash(s) = h`) and sends h to B
2. Both parties lock their assets in a SC on the respective BCs. The SCs have the ability to verify whether a given value v belongs to `hash(v)`.
3. A has $2 * X$ seconds to provide the key s to the SC that holds Bs locked assets, unlocking it in the process, resulting in a transfer of the assets to A. Should the key s not be provided within time, the SC returns the assets to B.
4. B has X seconds to provide key s to the SC holding A's locked assets, leading to the assets being transferred to B. Should the key s not be provided within time, the SC returns the assets to A.
5. A has X seconds to reveal the key s, leading to B learning s, enabling him to claim As items as described in the third step.

The hash-locking mechanism ultimately hinges on A to reveal the key s, meaning that B enables A to only reveal s if the exchange-rate between the different asset tokens has changed favorably within the time A has to reveal s [12]. Additionally, the involved BCs need to be capable of running the hash-timelock type of SC to implement the hash-locking mechanism [48].

2.3 Bifröst

Bifröst is a BC interoperability Application Programming Interface (API) that aims to provide users (*i.e.,* developers of BC applications) with a way to interact with a variety of different BCs without having to deal with the individual implementations of those BCs, by abstracting away the BC specific implementation details [48]. By eliminating the requirement for users to acquire knowledge of the respective underlying BC implementations, Bifröst removes restrictions standing in the way of innovative BC applications that rely on the interoperability of different BCs [48].

2.3.1 API Objectives

The primary objective of the Bifröst API is, as already alluded to, the creation of an interface that enables users to interact with a range of different BCs, whilst not being required to have knowledge of how the underlying BCs are implemented [48].

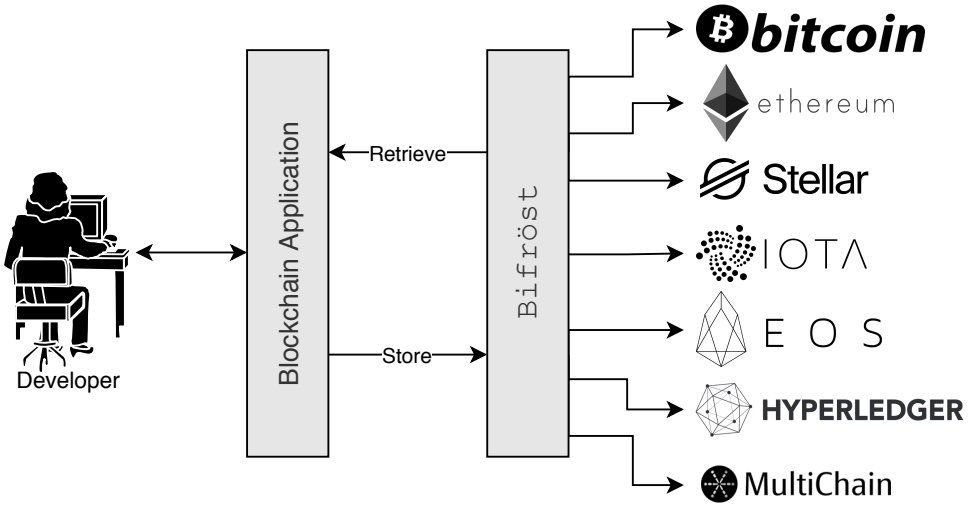


Figure 2.4: Using the Bifröst API [48]

Furthermore, a number of requirements were imposed on the design of Bifröst API, namely ***flexibility***, allowing users to store any arbitrary data on the supported BCs, provided it is formatted correctly, ***modularity***, enabling easier implementation of adapters for additional BCs, as well as ***ease of use***, which reflects part of the main goal of exposing simple API functions and abstracting the complexity of the underlying technical workings [48]. Additionally, the nodes are executed on BC Remote Procedure Call (RPC) servers, which provide an isolated an reproducible environment; thus, further increasing ease of use [48]. Figure 2.4 illustrates how BC applications interact with the Bifröst API [48].

2.3.2 Architecture

The architecture of Bifröst consists of three main components: *(i)* the API itself, *(ii)* the adapters to the different BCs, and *(iii)* a database [48]. Note that Bifröst uses a notary scheme to facilitate interactions with multiple BCs [48].

Figure 2.5 depicts an overview of the Bifroöst API, as well as the function flow for "store", one of the exposed API functions [48]. The rest of this section serves the purpose of providing a description of Bifröst's architecture.

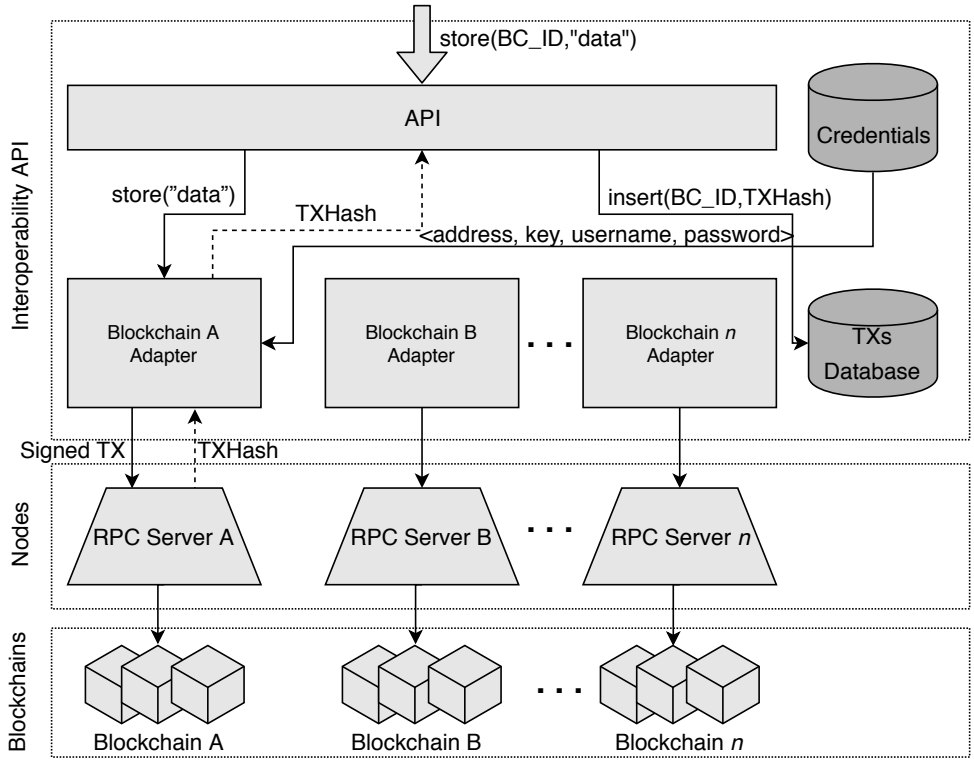


Figure 2.5: Bifröst architecture and store function [48]

Blockchain Adapters

The adapters have the task of converting the supplied user input to a format which conforms with the required format of the respective underlying BC, meaning that they create a transaction and forward it to the respective BC network [48]. In this sense, the adapters perform the first two stages of transaction processing shown in Figure 2.2. When the `retrieve` function is called, the adapter will not create a transaction but instead is responsible for retrieving the data from the BC [48].

Currently, Bifröst supports adapters for seven different BCs, as can be seen in Figure 2.4, meaning each supported BC has its own adapter and consequently, adding support for a new BC requires writing a new adapter [48]. Note that besides the different BC adapters there is also a PostgreSQL adapter, which can connect to a traditional database instead of to a BC [48].

Database

The role of the database is three-fold [48]: Firstly, it manages a list of the BCs that are supported by an adapter [48]. Secondly, it is responsible for storing the credentials of different users [48]. Note that this table may hold multiple credentials for each users, as the credentials are not universal to all BCs, due to their different credentials requirements [48]. Lastly, the database maintains a list of all transactions that have been completed on any of the BCs [48].

API

The API is responsible for exposing functions to the user and as such serves as the point of entry for interactions with Bifröst [48]. A total of three functions are accessible to users [48]:

- The function `store(text, blockchain)` allows the users to store data in string form on a BC. The first argument receives the data, whilst the second argument is used to specify the BC on which the data is to be stored. The function returns the transaction hash, once the data is stored on the desired BC and the transaction has been confirmed.
- With `retrieve(transaction_hash)` users can retrieve data which they stored in the transaction specified by the transaction hash. Note that it is not necessary to specify the BC on which the data was originally stored, since the API uses the database to automatically identify the corresponding BC.
- Finally, `migrate(transaction_hash, blockchain)` allows users to migrate data stored in the specified transaction to a different BC. Note that data existing on the BC can not be deleted. Thus, this function simply creates a copy of the data and stores it on the specified BC. The data which is to be copied is retrieved by using the `retrieve` function listed above.

2.3.3 Prototype Evaluation

For a detailed evaluation of the Bifröst prototype with regard to performance, security and supported data size, the respective sections in [48] can be consulted. This section simply aims to quickly illustrate the issue of how private keys are managed in the prototype, as it is of high importance to this thesis and then mention a number of functionalities that form a set of features that could potentially be added in later iterations of Bifröst.

The private keys used to sign transactions are stored on the database of the Bifröst prototype in a plain text format [48]. This means that should a hypothetical attacker gain access to the server, he would immediately have access to the credentials of all users. Thus, the way the private keys are handled currently poses a security risk [48].

As for potential features to be added, they include the handling of generic errors, the capability to compare the size of the data to be stored with transaction size limits and splitting the data up into multiple transactions if needed, as well as the option to have data be encrypted before it is stored on a given BC.

Chapter 3

Related Work

Given the growing importance of BC interoperability, caused by the increasing number of BCs - as of March 2021, [13] lists close to 9'000 different cryptocurrencies - numerous projects have sprung up that have their own approach to providing BC interoperability. The aim of this chapter is to give an overview over a range of different interoparability projects from all three of the different interoparability schemes mentioned in Section 2.2.

3.1 Hash-Locking Projects

3.1.1 Komodo

Komodo's AtomicDEX aims to enable cross-chain exchanges of cryptocurrencies via atomic cross-chain swaps [6]. Komodo uses so called "liquidity provider nodes" to improve liquidity by actively trading in cryptocurrencies [6]. Notably, Komodo enables anyone to become such a "liquidity provider" through providing an extensive setup tutorial [33].

3.1.2 Interledger

Interledger is designed to facilitate "secure payments across multiple asssets on different ledgers" [28]. Notably, Interledger is neither a BC, nor a token, nor a centralized service, but rather a "standard way of bridging financial systems" drawing inspiration from the architecture of the internet [28].

Interledger's architecture description uses the metaphor of a graph, where each node participating in the Interledger network is a vertex in the graph and each edge between two vertices are "accounts" between two parties [28]. This then enables the classification of parties into two types, *(i)* those with only one account, able to trade funds with other parties to which they have a connecting vertex, and *(ii)* parties with multiple accounts,

called “connectors” that enable trades between any paries they connect to [28]. The incentive structure for connectors is built on fees for facilitating such trades, with competition between connectors balancing out the fees charged by the connectors [28].

As a side-note; Interledger used to initially implement a notary scheme to enable interoperability, but due to notary schemes requiring trust in the notary, thus posing a potential security risk, the project has moved over to a hash-locking scheme [30].

3.1.3 Wanchain

Wanchain enables cross-chain transactions between popular public chains, such as Ethereum and Bitcoin, as well as between different private BCs or even between private and public BCs [75].

Wanchain enables the registration of assets from other BCs through their built in asset template resulting in a new type of asset associated with the respective foreign asset [75]. This new type of asset is then used to facilitate exchanges between Wanchains original token and the registered foreign asset [6]. The exchange itself happens via a “Locked Account” scheme, which locks the assets in place during the transfer [48].

3.2 Notary Scheme Projects

3.2.1 Herdius

Herdius at its core is a decentralized exchange platform [26]. Notably, Herdius aims to achieve its vision of a “truly decentralized exchange” via a concept they call “Distributed Virtual Wallet Network” (DIVIWA), a distributed, virtual wallet that can store private keys in a convenient, yet secure way [26].

This is done by introducing for each user a single “Herdius key” that is used to secure the keys used to access BCs [26]. The Herdius key itself is secured by splitting it and encrypting its parts, subsequently distributing it across the Herdius network [26]. Once the key is needed, it is reassembled via a multi-signatory threshold signature scheme [26].

3.2.2 Accenture

Accenture’s approach to BC interoperability follows a notary scheme approach, where a trusted node, a so called “interoperability node” connects to “gateway nodes” of different BCs [64]. This interoperabilty node is then tasked with taking the necessary steps to facilitate *(i)* “Value transfer”, meaning the transfer of assets from one BC to another whilst maintaining protection against double spending and *(ii)* “Active state”, which refers to a piece of data existing on both BCs and being fully synchronized between the BCs [64].

Note that Accenture's interoperability approach targets permissioned BCs, but there are mentions that a similar approach may be derived that is applicable to permissionless BCs [64].

3.3 Sidechain Projects

3.3.1 Aion

Aion positions itself as a "multi-tier blockchain network" that enable BCs that participate in the network to perform transactions amongst each other [52]. As such, Aion removes the need for a participating BC to establish a sidechain for all other connected BCs by acting as a central hub through which these transactions are routed [48]. Note that the Aion hub is a BC, called "Aion-1" and is the base implementation of the "Connecting network" [52].

Aion's architecture differentiates between "Connecting networks" and "Participating networks" [52]. Participating networks are networks that fulfill a set of requirements, which make them eligible for integration with a connecting network [52]. Notably, these participating networks are not limited to being BCs, as *e.g.,* oracles may also act as participating networks [52]. Connecting networks then use communication protocols, so called "Bridges" to communicate with participating networks, and in turn enable communication between different participating networks [52].

3.3.2 ARK

The ARK Platform uses so called "SmartBridges" to enable interoperability between a given BC and Ark [36, 2]. Furthermore, ARK provides for users to create a new BC with a token as a push-button solution, with the newly created BCs being automatically SmartBridge compatible [36]. There are two kinds of SmartBridges, "Protocol-Specific SmartBridges", which enable communication between BCs that utilize ARK Core Technology, and "Protocol-Agnostic SmartBridges", facilitating communication with BCs that rely on a different set of underlying technologies [2].

Note however, that in order for a BC to make use of SmartBridges, it needs to be contain certain computer code in its source code [67]. This means that for BCs not built via the push-button tools provided by ARK, modifications to their source-code are required [48].

3.3.3 BTC Relay

BTC Relay is considered to have been the first Bitcoin sidechain [30]. BTC Relay is an Ethereum SC which implements a Bitcoin SPV light wallet method, storing Bitcoin block headers, consequently enabling the verification of payments done on the Bitcoin BC [20].

So called "Relayers" submit Bitcoin block headers to the BTC Relay SC, which earns them a fee whenever one of their submitted headers is used to verify a payment [20]. This is the incentive-structure that enables the BTC Relay to be autonomous [20]. Note that the interoperability provided by BTC Relay is one-way, meaning that whilst it allows Ethereum to process Bitcoin block headers, it does not provide Bitcoin with the ability to process Ethereum block headers [48].

3.3.4 Cosmos

Cosmos is in essence a "network of many independent blockchains" that run in parallel [34]. These BCs are also called "zones", with the first BC in the network being the so called "Cosmos Hub" [34]. The Cosmos Hub has connections to other zones and is responsible for tracking different token types in the connected zones as well as keeping score of the totals of the different tokens, which it does by being constantly updated with recent block commits from the connected zones [34].

This way, two zones that are both connected to the Cosmos Hub can transfer tokens amongst each other in a secure and fast way, with all the transfer going through the Cosmos Hub, thus not requiring any liquid exchange having to occur directly between the two zones [34].

3.3.5 Polkadot

Polkadot in its approach to BC interoperability is similar to Cosmos [48]. Polkadot calls itself a "true multi-chain application environment", where not only tokens, but any kind of data can be communicated between participating BCs and even between BCs of different deplyoment types (as discussed in chapter 2.1.2) [77].

In Polkadot the hub to which the participating BCs, called "Parachains" or "Parathreads" depending on their payment model, is called "Relay chain" [77]. The payment model is the aspect where Polkadot differs from Cosmos in its approach, then whereas Cosmos does not place any costs on BCs wanting to join the Cosmos network, joining the Polkadot network requires the joining BCs to stake a large amount of their respective tokens [48]. This gives Polkadot leverage to exert more authority over the connected BCs in order to *e.g.,* punish misbehaviour [48].

3.3.6 Rootstock

Rootstock aims to complement the Bitcoin platfrom by provding a SC platfrom which supports turing-complete SCs, giving Bitcoin miners access to the SC market [35]. Furthermore, since Rootstock is also "compatible with Ethereum standards", Ethereum users are granted access to a SC platform compatible with their own, where the native currency is Bitcoin, which comes with beneftis such as *e.g.,* a larger user-base [35].

From the perspective of BC interoperability, Rootstock is a two-way pegged Bitcoin sidechain [35]. Since two BC platforms that support turing-complete SCs are required to form a trustless two-way peg, Rootstock uses what they call a collection of "Semi-Trusted Third Parties" (STTP), which are tasked with validating SPVs, as the Bitcoin platform cannot do so itself since it does not support turing-complete SCs [35].

3.4 Discussion

Given the collection of related works, a number of conclusions can be drawn. First, these related works tend to stem from the industry rather than academia, or at the very least there are no peer-reviewed papers that focus on them. Table 3.1 shows which works have a peer-reviewed scientific paper directly associated with them and which ones publish a white paper. Papers that mention a given work without an in-depth discussion of the theoretical underpinnings and technical specification are not seen as associated directly with the work in the context of this categorization.

To determine the publication type, Google search was used. The search inputs where "X" as well as "X white paper", where X is to be substituted for the respective work's name. If the search result shows *e.g.,* a conference paper or an article released in a scientific journal, then an examination is done on whether work X is discussed in depth in said publication. If so, it is considered peer-reviewed for the purpose of this categorization. If not, then based on whether a white paper is found or not, the publication type is set to "n.a" or "White Paper". The exception to this would be a work that presents extensive information on a business / marketing level which does not qualify as a white paper. For such works the publication type is set to "Business Paper".

Table 3.1: Comparison of Related Work

Work	Interoperability Scheme	Publication Type
Komodo	Hash-Locking	White Paper [32]
Interledger	Hash-Locking	White Paper [63]
Wanchain	Hash-Locking	White Paper [75]
Herdius	Notary Scheme	White Paper [26]
Accenture	Notary Scheme	Business Paper [64]
Bifröst	Notary Scheme	Peer Reviewed [48]
Aion	Sidechain	White Paper [52]
ARK	Sidechain	White Paper [2]
BTCRelay	Sidechain	n.a
Cosmos	Sidechain	White Paper [34]
Polkadot	Sidechain	White Paper [79]
Rootstock	Sidechain	White Paper [35]

Second, though anecdotal, the author has noticed a seeming lack of interest in notary scheme based solutions. This does seem to align with the fact that Interledger has abandoned its initial notary scheme approach for a hash-locking scheme due to security concerns [30].

If this holds generally and is not simply due to a sampling bias, then this could be seen as an indicator that notary schemes, due to requiring a trusted third party, thus introducing a security risk, are not seen as sustainable long term solutions for BC interoperability when compared to the other available interoperability schemes.

Chapter 4

Extending and Standardizing Bifröst

This chapter serves as a report for all of the work done on Bifröst by the author, both in terms of code and research. As such in Section 4.1 will contain information on how the interaction with Bifröst is standardized, how the architecture of Bifröst allows for standardization of BC adapters and a documentation explaining how adapters for additional BCs can be added. Section 4.2 outlines the design decisions and challenges faced when adding the new features to Bifröst, whilst Section 4.3 details the Python implementation for each of the added features.

4.1 Bifröst Standardization and Documentation

4.1.1 Standardization of Interaction

This section revolves around the standardization of how data is sent to and received from a BC interoperability API such as Bifröst. This includes both researching already existing standardization efforts, as well as proposing an actual JSON format which, wherever possible, is based on said efforts.

Existing Standardization Efforts

[27] draw a distinction between two kinds of standards. "Backward-looking standards" are standards that concern themselves with existing implementations and make an effort to extract a formal standard from them [27]. "Forward-looking standards" on the other hand do not make the attempt to deal with specific existing implementations, but rather treat BCs as "black-box implementations" and focus on standardizing interactions between different black-box implementations, which includes interoperability [27].

Furthermore, [27] establish that the time is not yet ripe for creating backward-looking standards, as there still are disagreements within the BC community about fundamental

aspects, such as what exactly constitutes a BC and what features a BC needs to have. Instead, focus should be put on forward-looking standards [27].

[23] draws a similar conclusion by pointing out that the BC technology has not matured enough to be standardized and that standards may act prohibitively against further developments of the technology. [16] meanwhile caution particularly against standards related to the technical aspects of BCs. In the author's view this aligns with the suggestion by [27], as standardization of the technology, as well as standardization of the technical aspects of BCs both do not align with the "black-box implementation" approach and hence would fall into the category of backward-looking standards.

A number of sources appeared to be promising leads regarding insights that might inform a JSON format for interacting with BC interoperability APIs. [25] propose the introduction of a common standardized "minimal operations set" that should be implemented by all BCs. Such a minimal set of operations could then be compared against the API functions exposed by Bifröst *e.g.,* with regard to the goal of the operation or the parameters it permits. This then could yield information about further functions that could be added in future iterations of Bifröst or parameters that allow for the introduction of additional features. [25] unfortunately do not present a concrete minimal set of operations, meaning such a comparison is not possible.

[6] name the Ethereum ERC's as "de facto standard" for standards concerning general interoperability. However, at the time of writing this, most proposals are still in their draft phase [73]. More importantly, they are intrinsically tied to the Ethereum BC, making them not that useful as Bifröst explicitly has the goal of abstracting away from underlying BC implementations.

[6] furthermore mention the programming languages "HSL" and "DAML", which abstract away from specific BCs by providing a generic BC model, as well as ledger protocols "Interledger Protocol" and "Web Ledger Protocol" which aim to enable common BC interactions. However, the inspection of their respective documentations ([24], [17], [29] and [53]) did not yield any information that was directly applicable to Bifröst.

The Web Ledger Protocol documentation comes closest to what is needed by providing a number of examples for how requests for storage events are structured in JSON format [54]. Inspection of these provide insight into how requests could be structured for the construction of specific supported object types with the `setObject` field as visible in the first example listed in [54]. This is not directly applicable to Bifröst in its current state, but might potentially be of interest for future versions.

In conclusion, to the author's knowledge, there is no project that has proposed a JSON format for request and response bodies of calls to BC interoperability APIs. Furthermore, according to the author's research detailed here, there is no existing BC standardization effort which could directly inform how such JSON formats should be structured, besides what is generic to requests and responses for all types of applications. The research did however yield an approach to specifying object types (see Web Ledger Protocol storage examples [54]) which is incorporated into a new JSON format proposed in this thesis.

Proposal of a JSON Format for Interoperability APIs

The JSON formats themselves are relatively simplistic, as not a lot of data and parameters have to be communicated. A format for both request and response in case of success will be shown for each of the three currently available API functions (`store`, `retrieve` and `migrate`). Finally, a format common for all API functions for responses in case of errors is shown and the errors that may occur are listed with their respective status code.

The JSON format for the store operation is shown in Listing 4.1. The overall structure of the request is identical for all three API functions, with the requests at the highest nesting-level consisting of the same two fields, `parameters` and `data`.

The `parameter` nesting (lines 3 - 8) contains fields for all those parameters that are of a generic type, such as *e.g.,* the `blockchain` identifier which is an integer or the `password` which simply is a string.

The `data` (lines 9 - 16) field meanwhile is responsible for housing those elements of the request which do not fulfill the necessary criterion to be stored as a field in the `parameter` nesting. The `data` field holds an array into which an arbitrary number of objects may be written. The structure of those objects is the aspect that gives this JSON format its flexibility.

Each of those object has to define two fields, `type` and `fields`, with the former indicating the type of the object, and the latter holding all the fields which the object type specified in `type` requires.

With this format then any kind of data can be transmitted whilst adhering to a well-defined but generic overarching nesting-structure. This then means that should Bifröst in the future *e.g.,* accept different data types besides a generic string for storage, the way the `data` field is structured will allow those data types to be passed without changes to the JSON format being required.

Imagine for example a future iteration of Bifröst that allows users to store phone contacts. The object in the `data` array then could have its `type` set to "phone_contact", with `fields` containing a number of predefined fields such as `name`, `last_name`, `phone_number` and `country` for the respective country prefix.

Additionally, the specified `type` tells the API which entries it should expect within a data object's `field` nesting, allowing for potentially different procedures based on the type of object received.

Finally, as the format is generic, it is not only usable in the context of Bifröst but it may potentially be adapted to other interoperability APIs too, no matter what kind of data or parameters the require users to send to their API.

Listing 4.1: JSON for `store` request and success response

```
1 # REQUEST
2 {
3   "parameters"   : {
```

```
4       "blockchain"   : 1,
5       "redundancy"   : "True",
6       "password"     : "password as string",
7       "multiple_tx"  : "False"
8     },
9     "data"   : [
10      {
11        "type"   : "string",
12        "fields"   : {
13          "value" : "data in string format"
14        }
15      }
16    ]
17  }
18
19  # RESPONSE SUCCESS
20  {
21    "status_code" : 200,
22    "data"   : [
23      {
24        "type"   : "transaction_hashes",
25        "fields"   : {
26          "value" : [
27            "0xb2057d719cc25b46886d3e274b385e ..."
28          ]
29        }
30      }
31    ]
32  }
```

As for the response, the philosophy is the same, with a response in case of a successful operation always returning both the `status_code` (line 21) and a `data` array (lines 22 - 31) that adheres to the same structure already outlined when discussing the request.

Note that in the case of the request shown here, the object held in the `data` array is a normal string and could potentially also be put into the `parameters` nesting, but for the sake of illustrating the format has been put into the `data` array. One could potentially make an argument that it should be stored in the `data` array in any case, as it is the "main input" for the `store` API function, but that is something which ultimately boils down to the personal preferences of the developer(s) of a given API and is ultimately irrelevant to how the format functions, hence the argument will not be discussed further.

Furthermore, Note that this setup with the `data` array is inspired by the storage examples shown in the Web Ledger Protocol documentation [54], as it is essentially a simplified variant of their `setObject` structure, and forms the main take-away from the research conducted into existing standardization efforts.

Moving on to `retrieve`, the JSON format for its request and success response are shown

in Listing 4.2. The structure of the request is identical to the one discussed for the `store` operation. Instead of a string it now takes a transaction hash as its `data` input, hence the different `type` specification (line 8).

Listing 4.2: JSON for `retrieve` request and success response

```
1  # REQUEST
2  {
3    "parameters"   : {
4      "password"   : "password as string"
5    },
6    "data"   : [
7      {
8        "type"   : "transaction_hash",
9        "fields"   : {
10         "value" : "0xb2057d719cc25b46886d3e274b385e  ...""
11       }
12     }
13   ]
14 }
15
16 # RESPONSE SUCCESS
17 {
18   "status_code" : 200,
19   "data"   : [
20     {
21       "type"   : "string",
22       "fields"   : {
23         "value" : "data in  string  format"
24       }
25     }
26   ]
27 }
```

In the response meanwhile the status code is returned as is the case for all responses. The `data` field once again adheres to the structure discussed previously and in the case of the `retrieve` API function simply returns a data object of `type` string.

The structure of the JSON format for the `migrate` function can be seen in Listing 4.3. The migration process internally functions as a retrieve process followed up by a store process. The JSON format reflects that by consisting of a request body whose `type` specification of the `data` is equal to the one for the `retrieve` function and a response that is equal to that of a `store` function.

Listing 4.3: JSON for `migrate` request and success response

```
1  # REQUEST
2  {
3    "parameters"   : {
```

```
4       "multiple_tx" : "False",
5       "blockchain" : 2
6     },
7     "data"   : [
8       {
9         "type"   : "transaction_hash",
10        "fields"   : {
11          "value" : "0xb2057d719cc25b46886d3e274b385e  ...""
12        }
13      }
14    ]
15  }
16
17  # RESPONSE SUCCESS
18  {
19    "status_code" : 200,
20    "data"   : [
21      {
22        "type"   : "transaction_hashes",
23        "fields"   : {
24          "value" : [
25            "0xb2057d719cc25b46886d3e274b385e ..."
26          ]
27        }
28      }
29    ]
30  }
```

Finally, the structure of a response in case of an error is displayed in Listing 4.4. Note that the format is equal for all three API functions. To get an overview over the possible error messages that may occur for each of the individual functions, consult Table 4.1.

Listing 4.4: JSON for error response

```
1  # RESPONSE ERROR
2  {
3    "status_code" : 404,
4    "error_message" : "error message text"
5  }
```

The list of status codes and error messages may easily be extended in the future. For more on how generic error handling is done, refer to Section 4.2.3.

4.1.2 Architectural Standardization

Whilst previously the focus was on how interactions with a BC interoperability API such as Bifröst might be standardized, there is another form of standardization embedded

Table 4.1: Status Code and meaning per API function

Status	Operation	Error Message
404	store	The specified Blockchain has not been found.
404	retrieve, migrate	The specified Transaction does not exist. Verify your Transaction Hash.
413	store, migrate	Cannot perform operation due to transaction size limit! Consider enabling multiple transactions.
500	all	Oops, Something went wrong! If this behaviour persists, please contact Bifröst support staff.

within the architecture of Bifröst. Namely the standardization of how adapters for specific BCs are constructed and thus, by extension, the standardization of the high-level process for each of the three exposed API methods (`store`, `retrieve` and `migrate`).

To understand how this standardization is built into the architecture of Bifröst, a closer look at how the adapters are structured is required. Note at this point, that what is discussed here relies on insights that can be gained from inspecting the code-base of the Bifröst project. The code is publicly available at [47]. Upon inspection of the code, it can be observed that the adapters make use of a principle called inheritance.

Inheritance, a concept which is part of object-oriented software engineering, describes the fact that classes, such as the adapters used in Bifröst, can be put into a hierarchical relationship of "superclass" and "subclass" [38]. For the purpose of understanding how Bifröst standardizes its adapters, it is sufficient to understand that such a subclass inherits the structure of the superclass, meaning it by default comes with the features (properties and operations), which are defined on the superclass, already built in [38].

Within the subclass, additional features may be added on top of the ones inherited from the superclass and inherited features may be redefined, thus overriding the behaviour of that feature as inherited from the superclass with their own behaviour, with the superclass being agnostic to the overrides [38]. Such inheritance hierarchies can be seen as a hierarchy of increasing specialization, with the uppermost superclass being the most abstract and generic class in the hierarchy and each subclass being more specialized than its direct superclass [38].

Figure 4.1 illustrates this within the context of Bifröst. Note that all the classes displayed within the figure have been simplified and do not display their entire list of features, as this is not meant to be an exhaustive discussion of all the classes' features involved in the inheritance hierarchy, but rather a simplified example to elaborate how the mechanics of inheritance come into play with regard to the API function `store` and adapter standardization, with the same principles applying analogously to `retrieve` and `migrate`. Those interested in inspecting all the classes' features may refer to the Bifröst codebase [47].

Within Figure 4.1, a number of classes are displayed. The generic `Adapter` class in terms of inheritance is the superclass in the adapter class hierarchy, whilst the BC-specific adapters

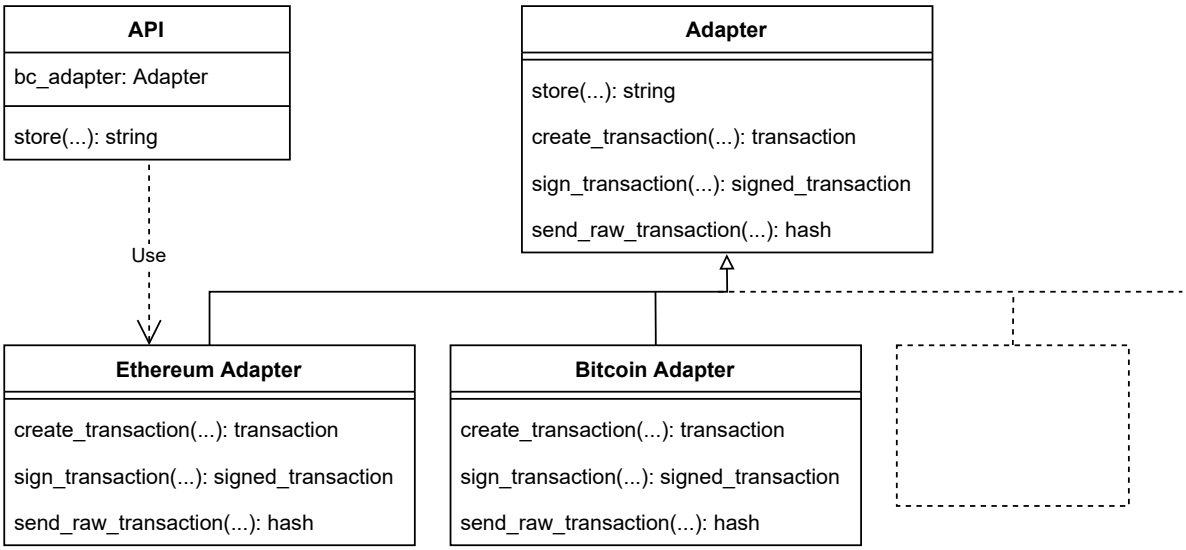


Figure 4.1: Bifröst adapter architecture, based on [48] and code insights

such as the Ethereum Adapter (`EthAdapter`)and Bitcoin Adapter (`BTCAdapter`) are its subclasses. In the Adapter class, a number of operations are defined, namely `store`, which defines the generic, high-level structure of the process of storing data on a BC, as well as `create_transaction`, `sign_transaction` and `send_raw_transaction`, which are called from within the Adapter's `store` method and thus can be seen as sub-processes of the store process.

In the subclasses, these three operations are listed again, indicating that they are redefined, thus overriding their respective implementation in the superclass. It is important to note here that the `store` operation is not redefined in the subclasses, meaning the implementation of the operation as it is present in the superclass remains unchanged in the subclasses.

As indicated previously, the `store` operation represents the high-level process of storing data on a given BC, whilst the operations `create_transaction`, `sign_transaction` and `send_raw_transaction` are sub-processes within the high-level store process. By enabling the redefinition of the sub-processes in every subclasses, each of which is responsible for a different BC, whilst simultaneously maintaining the high-level process, inheritance provides a mechanism to achieve standardization of the adapter architecture.

BC-specific aspects *e.g.,* how a transaction has to be constructed, are resolved at the subclass level for each adapter and thus for each BC individually. Since these BC-specific aspects are resolved in the subclasses, the superclass is left agnostic to anything concerning individual BCs, leaving the high-level process of storing data on an arbitrary BC generic and standardized. The same logic applies to the process of retrieving data from an arbitrary BC, as well as migrating data from and to an arbitrary BC.

This is where the API class in Figure 4.1 comes into play (note that technically speaking the API is not actually a class, but for the purpose of simplicity it is treated as such here). The `store` operation listed here is the one exposed to users of the Bifröst API.

Therein the `store` operation of an adapter is called, with which adapter is to be used being determined based on the user input the API receives. Since the `store` operation the API calls is agnostic to anything BC-specific as previously established, the `store` operation of the API is agnostic to that as well. Again, the same logic applies to the API operations `retrieve` and `migrate`, which are not listed here.

As a result, all of the three methods the API exposes to its users are generic, meaning they are always the same, no matter which BC is specified by the user and thus the overall processes of `store`, `retrieve` and `migrate` are standardized.

The standardized architecture of the adapters comes with a number of important benefits. First of all, any changes to something BC-specific *e.g.,* changes to any of the libraries used to interact with the different BCs can be resolved completely within the adapter of whichever BC that is affected, meaning no changes to the API, the generic adapter or any of the other BC-specific adapters are required.

Second, changes which are not BC-specific can be resolved entirely within the generic parts of the architecture, the API and/or the generic adapter. As such new generic elements within the high-level processes, such as data encryption (see Section 4.2.1), can be added without having to revisit each BC-specific adapter and without increasing the amount of work which needs to be done when constructing a new BC-specific adapter. Essentially, the first two points indicate that Bifrösts architecture achieves good ***modularity***.

Finally, it results in what can be seen as three distinct levels of abstraction, with each of them having a set of responsibilities. In case of the API these are facilitating interaction with its users, initiating the needed high-level processes and bookkeeping of the transactions that have been made. The generic adapter meanwhile is responsible for structuring the high-level processes and for any generic operations not covered by the responsibilities of the API, whilst the BC-specific adapters are responsible for anything that has to do with the individual BCs. This distribution of responsibilities is valuable, as it allows quick identification of the level at which changes should to be made *e.g.,* when introducing a new feature.

4.1.3 Documentation

Given the architectural standardization discussed in the Section 4.1.2, the process of adding support for additional BCs by adding new adapters is clearly defined. This part of the thesis shall serve as a documentation on how a new adapter is to be created, which operations inherited from the generic adapter superclass are to be overriden and in which locations in the Bifröst codebase outside of the adapter classes changes have to be made.

Note at this point that the word “operation” is synonymous for “method” or whichever term is used to describe an operation defined within a class in a given programming language. In the case of Bifröst the implementation at hand is written in Python, so the term “method” shall be used throughout the rest of the thesis. Although Bifröst is implemented in Python and thus any files mentioned will be Python files, the general approach should translate to other programming languages as well. Any files mentioned are referenced by their file path relative to the root folder of the project.

Minimal Setup for Implementation

The first thing that needs to be done when getting ready to implement a new adapter is to create the respective adapter class, such that it is a subclass of the generic `Adapter` class found in `adapters/adapter.py`. Once that it is done, any new libraries that add support for the new BC as well as their respective dependencies need to be installed. The newly supported BC then will have to be entered in the enum called `Blockchain` within `blockchain.py`. In addition to that, within `db/config.py` the confirmation times for transactions made on the new BC need to be specified within the variable `CONFIRMATION_WAITING_TIMES` and the size limits for the transactions will have to be registered in the variable `TRANSACTION_SIZE_LIMITS`.

Furthermore, the credentials with which the API will authorize transactions on the newly supported BC need to be created and specified in `db/config.py` within the `CREDENTIALS` variable. How theses credentials are obtained will differ from BC to BC and is best looked up in the technical documentations or tutorials of any given BC. Note that if the BC you are adding support for uses an UTXO-model for its funds handling, you will need to create a seed transaction which simulates such an unspent transaction output, so that any transactions made for that BC has an UTXO it can use as input. Such seed transactions can be specified within the `TRANSACTIONS` variable in `db/config.py`.

Finally, in the file `api.py`, the variable `Adapter` will have to be extended with an entry where the name is the enum entry created previously and the assigned value is the newly created adapter class. With that the setup is ready such that implementation of the new adapter class can begin.

Note that if the terminal interface is to be used to interact with the API, then one further modification is required. In the file `cli.py` in the functions `caseStore` and `caseMigrate` one of the objects defined in the questions array is responsible for presenting the user with a list of BCs to chose for their respective operation. The possible choices are specified as objects within the `choices` array. In both cases, the variable `bc_choices` is used, where all the different BCs are defined in an array. To have the new BC be available as a choice it consequently needs to be added there. Once that is done the terminal interface is set up to support the new adapter.

Implementing the new Adapter

The actual implementation of the new adapter then starts by assigning values to a number of properties within the new adapter class.

Table 4.2 lists properties which have been defined within the generic adapter. Since the new adapter class inherits from the generic adapter, it has the option to override these properties by assigning values to them. Inspection of the code shows that the properties are abstract, meaning they have not actually been implemented in the superclass, hence they need to have a value assigned if they are to be used anywhere in the new adapter class.

Table 4.2: Bifröst adapter class properties structure. Based on code insights

	Generic Adapter	BC-Specific Adapter	(Potential) Additional Attributes†
Properties	`chain`	`chain`	`web3`
	`credentials`	`credentials`*	`ENDPOINT_URI`
	`address`	`address`*	
	`key`	`key`*	
	`client`	`client`*	

* Optional override, † Examples from Ethereum adapter

Further inspection shows that not necessarily all of these properties have to be used in each adapter and that, more importantly, with the exception of `chain` they are never used explicitly anywhere in the superclass. The reason for that is that the other properties are set up to be used in those methods that have been categorized as "sub-processes" (*e.g.,* `create_transaction`) in Section 4.1.2 which the superclass only defines but never implements itself.

What this means is that it is left to the implementation of the BC-specific adapters to decide whether to use the properties or not. As such the only property that has to have a value assigned to it in every BC-specific adapter is the property `chain`. The value assigned to it is the enum entry created earlier on in `blockchain.py`. As for `credentials`, `address` and `key`, their respective values are meant to be those that have been entered previously in the `CREDENTIALS` variable in `db/config.py`, with `credentials` holding the entire entry and `address` and `key` being assigned their respective counterparts from that entry, such that they can be accessed more conveniently. Finally, `client` is meant to be used when access to an external client is required for some action, such as *e.g.,* signing a transaction in the Ethereum adapter.

Should there be the need for additional properties, then these should be added in the new adapter class and not in the generic adapter class. For an example of such additional properties, refer to the Ethereum adapter examples given in Table 4.2, where there are two such properties, `web3` and `ENDPOINT_URI`.

Moving on to the methods, Table 4.3 shows for each of the three high-level processes (store, retrieve and migrate) which methods are defined at the level of the generic adapter, which of those have to be overridden at the level of the BC-specific adapter and, at the example of the already implemented Ethereum adapter, what kind of methods may be newly introduced within a BC-specific adapter. Note that there are no methods associated with the process of migration, as migrate internally uses both the store and the retrieve process to achieve its goal.

As is shown in Table 4.3, unlike the properties, the methods are defined clearly with respect to which methods from the generic adapter have to be overridden. Each of the methods listed in column "BC-specific Adapter" has to be implemented in every BC-specific adapter, whilst each method not listed in that column should not be overridden in any BC-specific adapter.

The names of the methods are self-explanatory in terms of what the methods wants to

Table 4.3: Bifröst adapter class methods structure. Based on code insights

	Generic Adapter	BC-Specific Adapter	Helper Methods†
store	`store_wrapper`		`estimate_gas`
	`store`		
	`confirmation_check`		
	`create_transaction`	`create_transaction`	
	`sign_transaction`	`sign_transaction`	
	`send_raw_transaction`	`send_raw_transaction`	
retrieve	`retrieve_wrapper`		
	`retrieve`		
	`get_transaction`	`get_transaction`	
	`extract_data`	`extract_data`	
	`to_text`	`to_text`	

† Examples from Ethereum adapter

achieve. How the method is implemented depends entirely on the BC for which the adapter is being created, as the different BCs, as well as the interfaces provided by potential libraries to facilitate interaction with the respective BCs, may vary. Here it is necessary to work with the technical documentations of the BC in question and of any libraries that are being used.

Analogous to the properties, here too additional methods may be defined, should they be of help when implementing the methods specified in column 2 of Table 4.3. Such additional methods should not be added to the generic adapter, but rather implemented directly within the BC-specific adapter. For an example of such a method, refer once again to the Ethereum adapter, where the method `estimate_gas` has been added as an additional method.

4.2 New Features Design

This section serves as a high-level outline of the designs produced and choices made with respect to the new features. The focus is put primarily on information flow and how the introduced changes affect the initial Bifröst prototype with regard to the three key requirements for Bifröst (***flexibility***, ***modularity*** and ***ease of use***) put forth in [48].

To quickly recapitulate, flexibility means users can store arbitrary data on the supported BCs, given proper formatting, modularity relates to the architecture of Bifröst and means enabling a simple way to add support for additional BCs [48] and ease of use refers to the Bifröst API exposing simple functions to users and abstracting away the complexity of interacting with the supported BCs.

The main goal pursued during the design-phase was to introduce the changes necessitated by the new features in such a way that any negative effects on these key requirements are kept to a minimum.

4.2.1 Encryption Feature

Introduction and Potential Impact

This feature gives users of Bifröst the ability to have the data they want to store on a given BC to be encrypted. To this end, they are given the opportunity to provide a password of their choice alongside the data, which is then used by the API to encrypt the data with the specified password. The same password then has to be provided in the retrieval process, when the users want their data to be decrypted before it is returned to them.

Looking at the three key requirements, ***modularity*** is the only one of them that is of some concern, as depending on where the encryption takes place within Bifröst, the individual BC-specific adapters, each responsible for interacting with a given BC may be affected. If this were to be the case, then the act of adding support for a new BC would now also have to concern itself with encryption, in addition to what already needed to be done in the initial version of Bifröst, resulting in poorer modularity. Furthermore, each already existing adapter would have to be modified. This is to be avoided.

There is no effect on ***flexibility***, since the API expects to receive data in string format and stores the data in string format on the specified BC [48]. Encryption simply speaking turns plain-text, whose information content is open to be read by anyone, into cipher-text where the information content is not derivable [59]. In the context of Bifröst, both plain-text and cipher-text can be represented in string format, hence the kind of data that can be stored is not affected.

As for ***ease of use***, the only change is that the exposed API functions `store` and `retrieve` now allow for an additional parameter, the password. This parameter is optional, meaning no input for the password parameter has to be supplied. Hence the effect on ease of use is minimally impacted by the introduction of this feature.

Choice Of Encryption Scheme

There are two distinct encryption schemes that were considered for the data encryption feature, *i.e.,* symmetric and asymmetric encryption. In symmetric encryption the key used to decrypt a piece of data is the same key that was initially used to encrypt it, or it is at least easily derivable from it [51]. This means that in order to establish secure communication using symmetric encryption, that key must be shared between the parties in such a way that is not leaked to outsiders and thus gets compromised [51].

In asymmetric encryption schemes this is not the case, as the key used for decryption is not the same as the key used for encryption [51]. Instead, each participant in such an encryption scheme holds a pair of keys, one of which is secret and shall be called “secret key”, with the other key being publicly available hence being called “public key” [51]. A participants public key is then used to encrypt a message sent to them and only that participant is able to decrypt the message, since the participants never share their secret keys and thus the participant is the only one who knows the secret key needed to decrypt the message again [51].

This scenario is depicted in Figure 4.2 on the left side, where Alice sends an encrypted message to Bob using an asymmetric encryption scheme. The crucial difference between symmetric and asymmetric encryption then is that in asymmetric encryption there is never the need to exchange a secret key, whilst symmetric encryption entirely relies on such an exchange [51]. This advantage comes at a cost however, with symmetric encryption algorithms being significantly faster than their asymmetric counterparts [81].

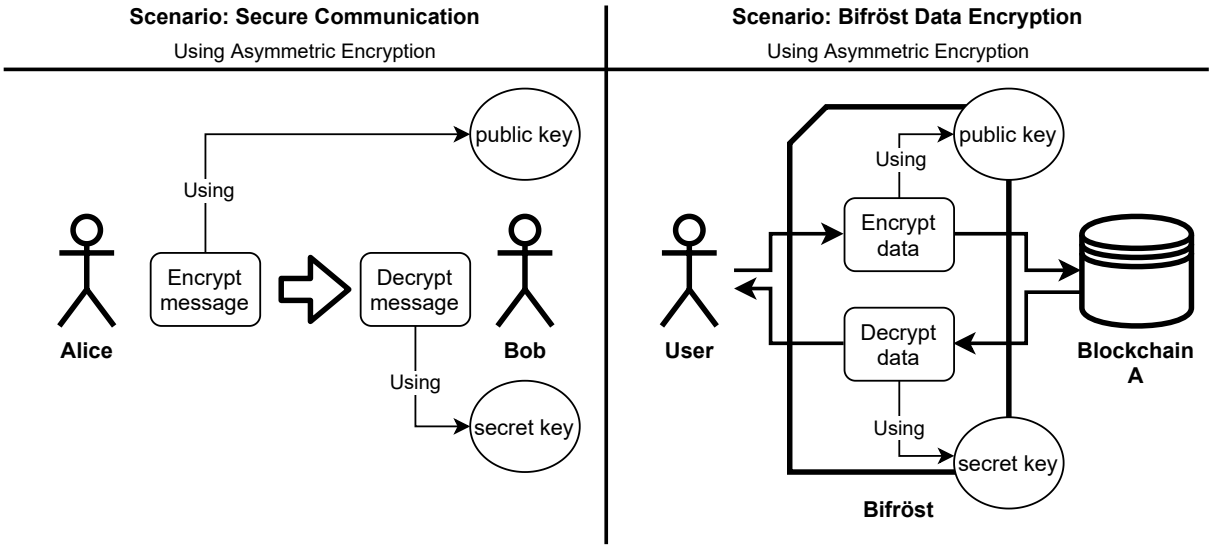


Figure 4.2: Comparison of encryption scenarios. Left side based on [51]

Looking at the scenario of Bifröst, as depicted in Figure 4.2 on the right-hand side, one can identify that there is only one entity, Bifröst, involved in encrypting and decrypting data, meaning the scenario of secure communication with two or more parties as outlined above and shown schematically in Figure 4.2 on the left, does not apply. Since only Bifröst partakes in the encryption and decryption of data, it would use its own public key to encrypt data and its own secret key to decrypt it again. Because only Bifröst uses its own public key it removes the necessity to make the public key public entirely.

Thus, Bifröst not having to share any key information with an other entity means that key-exchanges are not a factor that needs to be considered. As such the advantage asymmetric encryption has in scenarios where key-exchanges are necessary does not apply to the Bifröst scenario. Choosing an asymmetric encryption scheme for Bifrösts data encryption feature then would mean to not benefit from the advantages asymmetric encryption has to offer, whilst suffering the drawbacks of its comparatively poor performance. As such, the encryption scheme of choice for Bifröst is symmetric encryption.

Design

There are three distinct locations where encryption and decryption can take place, at the level of the API, at the level of the generic adapter or at the level of the BC-specific adapters. Doing so at the level of the API would dilute its tasks, which are allowing user interaction with Bifröst and keeping track of any transactions that have been made.

Doing so at the level of the BC-specific adapter is what has been identified at the start of Section 4.2.1 as leading to poor modularity, as it would affect each individual adapter separately. This leaves the generic adapter, which happens to be a good fit.

The reason the generic adapter is a good fit is that since the procedure and mechanisms for encryption and decryption are identical no matter what BC the data ultimately has to be stored on, there is no need to ever override any of the methods used to perform encryption and decryption within the subclasses, meaning the BC-specific adapters. As a result, the BC-specific adapters remain completely agnostic to the existence of the encryption feature, resulting in modularity not being affected at all by the introduction of this feature. For more details on the class hierarchy of the adapters and how the BC-specific adapters relate to the generic adapter, refer to Section 4.1.2.

Whilst there is no effect on the BC-specific adapters, there is an effect on the API. The reason for that being that if a piece of data for a given transaction is encrypted, information about that encryption, namely the "salt" and the "verification key", has to be stored for each transaction in the database, otherwise it cannot be decrypted again. Since the API is responsible for tracking the transactions that were made, it therefore needs to be changed, such that it can make the entry for the encryption information in the database. Alongside that the transactions table in the database needs to be expanded to accommodate the additional information.

Cryptographic Discussion of the Encryption Scheme

Understanding why salt is used in cryptography, requires a high-level introduction to how passwords are stored. Commonly, passwords are not stored in plain-text, but rather the cryptographic hash of a password is stored instead, such that access to whatever medium the passwords are stored on does not immediately compromise the passwords [21]. However, simply hashing the passwords with a cryptographic hash function does not render them secure against all forms of attacks though [21].

In particular a "birthday attack" or its more sophisticated version, the "dictionary attack", can be used to deduce the original password based on its stored hash [21]. These attacks consist of an attacker comparing a precomputed table of inputs and their respective pre-computed output, given a cryptographic hash function, against the present collection of password hashes, with any matches between the precomputed outputs and the stored password hashes resulting in those passwords being compromised [21].

To combat this, salt, a randomly generated value, is hashed alongside the password, thus randomizing the password [21]. Since a given salt is at most used for a small number of passwords, an attacker that managed to guess a given salt correctly would end up comparing a large precomputed table of hashes against only a small subset of the stored password hashes, essentially making this type of attack not worth the effort [21].

The reason then why the Bifröst API has to store the salt value used in the process of encrypting the data for any given transaction, is that in order to decrypt a piece of encrypted data again, the same key used to encrypt it has to be reconstructed again. Since that key in part is based on the salt value, the salt needs to be stored.

As for the aforementioned verification key, it essentially corresponds to the result of hashing password and salt. Storing salt and verification key then allows for the verification of a given password input [21]. The new password input is hashed alongside the salt which was used in the encryption process for a given transaction. Should the resulting hash equal the stored verification key, the password that was provided is the same as the one used in the initial encryption process [21].

Hence, storing salt and verification key for each transaction that holds encrypted data is essential to be able to verify, whether the password provided when the decryption of a given transactions data is requested matches the password used to initially encrypt the data.

However, the the goal ultimately is not just to verify passwords, and thus decide whether or not decryption should be attempted, but also to actually encrypt and decrypt data in a secure way. Since the verification key is stored in plaintext in the Bifröst database, it is not safe to use for encryption, as any attacker with access to the database then could use the unsecured verification keys to decrypt any piece of encrypted data.

Therefore a secondary key is derived in, which is based on the salt, password and verification key. Unlike the verification key, this secondary key is never stored and has to be reproduced each time it has to be used for either encryption or decryption. Any attacker with access to the Bifröst database has access to the salt and verification key used in the encryption process for a given piece of data. However, they do not have access to any of the passwords. Furthermore, the passwords can not be derived from the salt and verification key, as outlined earlier.

This means that the secondary key can only be reconstructed upon provision of the correct password, which can not be done by an attacker, unless the password was compromised in a way outside of the control of the Bifröst system. This then renders any data encrypted with such a secondary key secure.

The entire encryption scheme is rather complex if only elaborated in words without accompanying illustrations. Hence in the following section provides a graphical rundown of the encryption scheme.

Illustration of the Encryption Scheme

Throughout this section, the encryption scheme will be explained via the use of Figure 4.3. Furthermore, numbers in brackets, *e.g.,* (3), unless stated otherwise, are used to indicate certain aspects / sub-processes within that Figure. The explanation is based on a scenario where a user first requests to have some data encrypted and stored on a BC and later on retrieves that data again, after having it decrypted by the API.

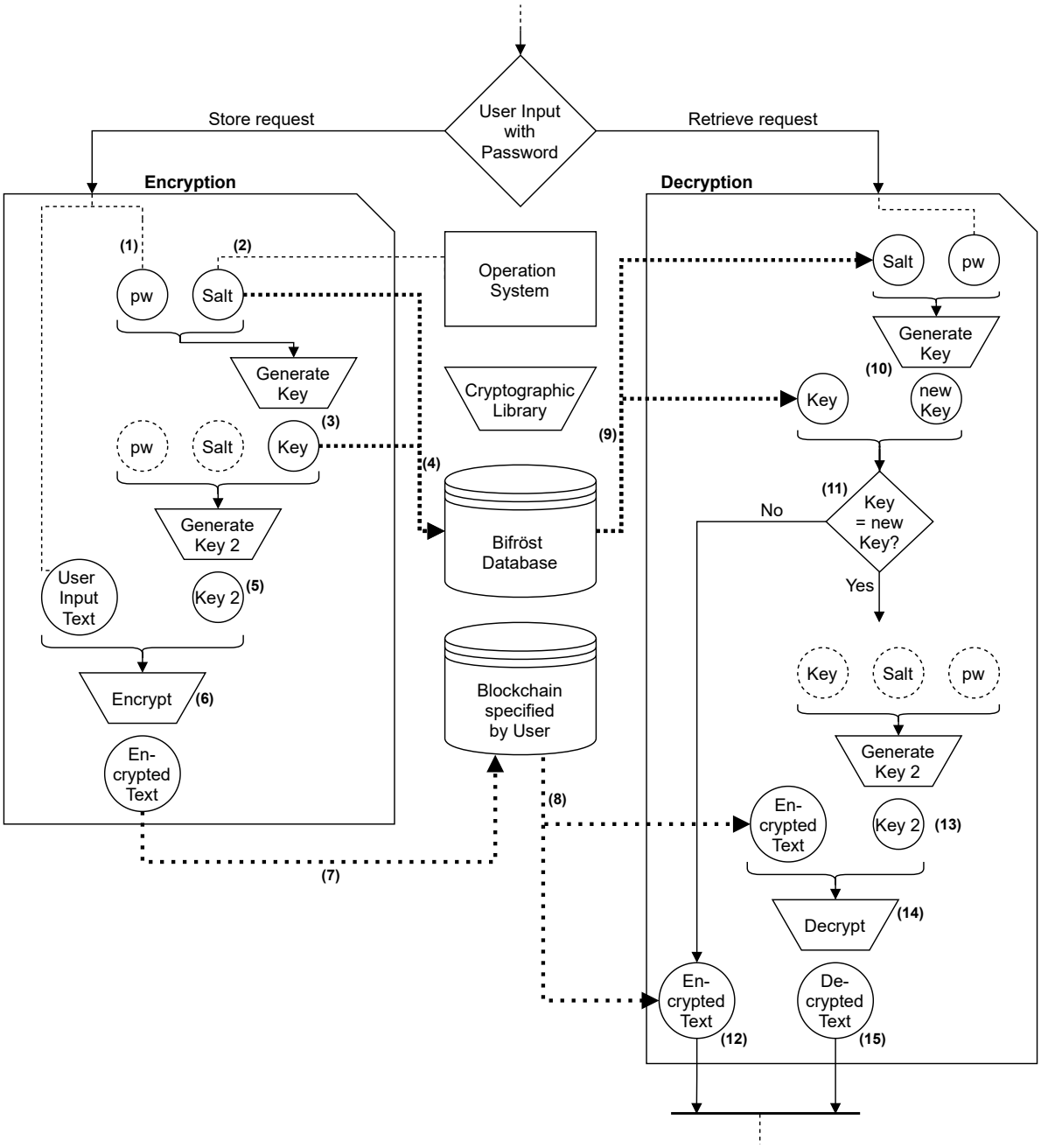


Figure 4.3: Bifröst encryption scheme

Upon providing the API with all the necessary data for a store procedure, the encryption process works as follows: First, the password is read from the user input (1). After that, the salt is generated randomly (2). Together, password and salt are used to derive the first key, which is the "verification key" (3). Both salt and verification key are stored in the Bifröst database (4).

A secondary key (5) is then derived from password, salt and verification key, making it only reproducible when all three of these components are available. This secondary key is then used to perform the actual encryption of the data (6). Once encrypted, the data is then stored with a transaction on whatever BC the user specified (7).

To briefly summarize the state after encryption; the API stored salt and verification key for this particular transaction. Since the secondary key however also requires the password to be reconstructed, the data is secure, even if the Bifröst database should be compromosied, as the password is never stored.

To retrieve the data again and have it decrypted, the user must provide the API with the transaction hash he received, indicating the specific transaction on which the data was stored, as well as the same password used to initially encrypt the data. Given those inputs, Bifröst uses the transaction hash to identify the transaction and receive the encrypted data from it (8). Furthermore, knowing the transaction hash, the salt and verification key associated with that transaction are retrieved from the database (9).

Given the retrieved salt and the password, the same procedure as in (3) is used to reconstruct the verification key (10). The resulting key is then compared against the retrieved verification key (11). Should the keys not be equal, then verification fails, meaning an incorrect password was provided. In this case, the data is returned still encrypted to the user (12). Should the keys match however, then the correct password was provided. In this case the verification key, salt and correct password are present, allowing for the reconstruction of the secondary key (13). This key equals the secondary key used during encryption (5). Given that, the data can now be decrypted (14) and returned to the user in its decrypted form (15).

4.2.2 String Splitting Feature

Introduction and Potential Impact

The role of the string splitting feature is to ensure that a piece of data that is to be stored on any given BC does not supersede the maximal amount of bytes that is permitted to be stored with a single transaction. See Table 4.4 for a list of BCs currently supported by Bifröst with their respective transaction size limits. The size limit listed there corresponds to the number of bytes of "arbitrary data" that can be attached to a transaction on the given BC and not *e.g.,* the maximal size of a smart contract.

Table 4.4: Transaction Size Limit for Supported Blockchains. Adapted from [48]

Blockchain	Transaction size limit (in Bytes)
Bitcoin	80
Ethereum	46'000
Stellar	28
EOS	256
IOTA	1'300
Hyperledger	20
Multichain	80

Note furthermore that some of these limits are not necessarily clearly defined, such as in the case of Ethereum, whose transaction size limit is only limited indirectly by the block

size limit, which in turn is limited by the block gas limit [46]. That block gas limit is not necessarily static and may change over time, as miners have the capability of raising or lowering the gas limit for each block by approximately 0.1% [61].

Given the fact that such size limits exist, it must be ensured that users of Bifröst have a way of navigating the issue of potentially superseding the imposed size limit resulting in the transaction attempting to store their data being rejected. As such, Bifröst must be capable of recognizing data that is too large for a given BC, have a way of splitting that data up into multiple pieces that then are storable using multiple transactions, as well as a mechanism to reconstruct the complete data when retrieving it again, which furthermore requires a mechanism to track transactions that hold parts of a given piece of data. Finally, as multiple transactions lead to increases in factors such as *e.g.,* transaction fees, the user must be given control over whether his/her data should be allowed to be split or just rejected should it supersede the given size limit.

Looking at the key requirements of Bifröst (***flexibility***, ***modularity*** and ***ease of use***), the potential impacts the introduction of the string splitting feature may have, are very similar to those of the encryption feature discussed in Section 4.2.1:

- Flexibility is not impacted, since whether the string representing the data is split or not does not alter the kind of data that is ultimately storable on a given BC.
- Ease of use is minimally impacted, as the user only has to deal with an additional parameter in the `store` and `migrate` functions of the API. The fact that splitting may have to be done in the author's view does not affect ease of use, as it is due to limitations of the underlying technology and not due to implementation choices. It is also not possible to resolve the issue silently behind the scenes, as the user needs to be aware of the effects splitting can have, such as *e.g.,* increased transaction fees.
- Modularity is the main concern, as depending where splitting is built into the existing system, changes to the individual BC-specific adapters may be needed, which would result in reduced modularity. Once again, this is to be avoided.

Design

Similar to encryption, the mechanics used for splitting and reconstruction of the data are ultimately independent of any specific BC. The only aspect that changes based on the BC the user specified is the size limit. Given a way to access the needed size limit outside of the BC-specific adapter, the same wrapper method as was used to introduce the encryption feature could be used to also introduce splitting and reconstruction.

Note at this point, that encryption would have to be performed first, as it does have an effect on data size. Otherwise, the splitting would have parts of the data at the maximal permissible size, but subsequent encryption would push the size over the limit once again. For more on how encryption affects data size see Section 5.2.

Since users of Bifröst explicitly have to specify the BC on which they want their data to be stored, that information is present from the very beginning of the store process and

can then be used to look up the size limit for the specified BC before the process reaches the BC-specific adapter. Hence the scenario outlined above where the wrapper method introduced with the encryption feature is used to also perform splitting and reconstruction is feasible and is the approach pursued during the implementation phase.

Splitting and Reconstruction Mechanics

The way splitting of over-sized data is done is simple. The data Bifröst receives is always formatted as a string. A string can be seen as a sequence of characters ([42] states Python itself does technically not have a built-in character type, but instead a string of length 1. For the sake of simplicity, the term character will still be used to refer to strings of length 1 in this context), with each character having a size of 1 byte, meaning the number of bytes used to represent a given string is the number of characters contained within that string [57].

With that splitting is simple, as it only requires comparing the number of characters in the string specified by the user with the given BCs size limit. Should the string exceed the limit, then the limit marks the splitting point, meaning that, *e.g.,* in the case of Stellar with a data size limit of 28 bytes, a string of length 100 would have to split into four parts, three of length 28 and one of length 16. This is done using a *ceiling* function to round up the result.

Before discussing the reconstruction of split data, it must be established how Bifröst tracks which transactions contain data that belongs together. Not only is it important to know which transaction's data together form a larger piece of data, but also the order in which the data needs to be concatenated to reconstruct the data. To do so the fact that splitting and reconstruction happens in the wrapper methods can be used, because the wrappers themselves do not actually perform the actions of creating and retrieving transactions, that is still done by the original `store` and `retrieve` methods called from within the wrappers.

This means that when splitting data for storage, the `store_wrapper` method has both access to the transaction hashes of all the transactions that were performed during that particular store process, as well as the order of the transactions, given by the order in which the hashes are returned by `store`.

With that information present, when storing the information for a given transaction in Bifröst's database, a new column called `next` is used to specify the transaction hash of the transaction that was made after it with the next part of the data. In case of the last transaction in a given store process, the field will contain the transaction hash of the first transaction made in the same process, thus creating a loop of pointers between the transactions involved in storing a given piece of data.

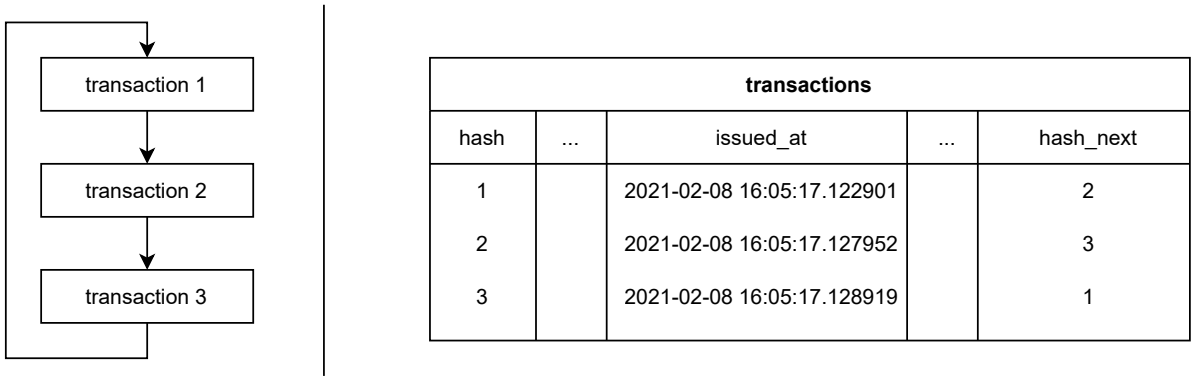


Figure 4.4: Bifröst Data Split Tracking

This loop of pointer enables Bifröst to track down all transactions used in a given store process based on any single transaction hash that belongs to that group of transactions. Due to that, both the `retrieve` and `migrate` have access to the entire group with the user only having to specify a single transaction hash, meaning no changes to those functions' signatures are required with regard to how the users delivers information to identify the data that needs to be treated.

The order of the transactions meanwhile can be established based on the entries of the `issued_at` column in the database's transaction table, which holds the timestamp of when the entry was created.

Figure 4.4 illustrates this on an example where data is split up into three parts. On the left side, the loop of pointers that is kept via entries in the database is shown. On the right, a simplified excerpt from the database's transaction table is displayed to illustrate how this loop of pointers and the timestamps is stored.

Given the data stored to track data splitting, reconstructing of the original, non-split data is simple. In the example shown in Figure 4.4, if the user calls the API's `retrieve` function and specifies the hash `2`, then first the transaction data for transaction 2 is retrieved. The `hash_next` entry indicates that there was another transaction involved, leading to the retrieval of transaction 3 and in the same vein to the retrieval of transaction 1. Once transaction 1 is retrieved, its `hash_next` entry is evaluated and since it is already present in the retrieved collection of transaction, Bifröst knows that all required transactions have been retrieved. Now the order can be established using the `issued_at` entries of the retrieved transactions.

Once that is done the strings can be concatenated and the data is completely reconstructed. In contrast to the store process, where encryption has to be dealt with before splitting, when retrieving data the reconstruction has to occur before decryption.

Finally, for the migration process, the only change is an additional parameter in the function signature, to enable / disable data splitting. This is due to `migrate` internally using a `retrieve` process, followed by a `store` process. Since `store` has been changed to receive an additional parameter, the value that parameter must also be present in `migrate`.

4.2.3 Error Handling Feature

Introduction and Potential Impact

Generic error handling in the case of Bifröst means introducing a means *(i)* to catch errors that may occur during the execution of the code, as well as *(ii)* doing so with a setup that is generic enough that it can easily be extended to catch additional errors.

The latter point refers to the fact that error handling, *i.e.,* the catching of errors can both target specific errors, as well as just any non-explicitly caught error that happens in general. Extending the error handling then does not refer to catching additional non-explicitly targeted errors, as there any error is caught anyway, but rather creating a setup where it is easy to add additional specifically targeted errors, which then can be treated separately.

It may be tempting to assume that such error handling is of little relevance for users and that it is mainly something the developers will benefit from, but that is not the case. It is important for any program to have well defined behaviours in case of something going wrong, even if it is just something simple like displaying an error message, such that the users at least receive confirmation that indeed something has gone wrong during the execution and are not left wondering whether *e.g.,* their connection is faulty or they simply experience long delays.

The aforementioned assumption is correct however with regards to how users interact with Bifröst, as the introduction of generic error handling does neither negatively affect ***flexibility***, meaning the kind of data the users can store, nor ***ease of use***, as interacting with the API has not become more complex. If anything, ease of use is impacted positively, as the behaviour in case of errors is both more robust, due to being clearly defined, as well more transparent towards the users.

Once again, the main concern is with ***modularity***, as errors may occur anywhere in the code, thus, overzealous application of error handling mechanics may unnecessarily overcomplicate interactions between API, generic adapter and BC-specific adapters, thus making the introduction of support for additional BCs more complex.

Thankfully, due to the fact that the API functions are essentially the coordinating entities for the entire process that happens within Bifröst for any of the three exposed actions (`store`, `retrieve` and `migrate`), this can be avoided quite easily.

High-level Introduction to Error Handling

Because the design relies heavily on the principles of how errors are handled in code, first a brief high-level outline of how those mechanics function is given. Note, that there may well be differences in how those mechanics work in languages other than Python, but the general approach is likely the same. The terminology will differ too, but since the implementation at hand is written in Python, its terminology is used here.

Fundamentally, error handling relies on the principle of "try-except", meaning some part of code is executed, essentially "tried out" and if errors occur within that section of the code, there is the opportunity to prescribe how the program is to proceed if "exceptional behaviour" occurs, *i.e.,* an error happens, even if it is just simply telling it to terminate in a controlled fashion.

Error handling can be targeted towards specific errors as well as towards a generic exception that simply will catch any error that occurs. Ideally, a combination of both is used, where first specific errors are handled and thus can be treated separately, and where afterwards a generic except will universally treat all the errors that do not fall into the first category.

Finally, there is the capability of writing code that even if it does not run into any of the numerous errors that are built into a given programming language can still "raise" an error by itself if certain conditions are met. This includes raising custom errors that a developer can define. Such manually raised errors, no matter whether they are built in errors or custom errors will also be caught by a generic except clause and can be treated separately with targeted except clauses.

Design

Due to the facts that *(i)* a try-except setup will allow the code to detect any errors that happen within the encapsulated code and that *(ii)* the API functions exposed by Bifröst each encapsulate the entire code related to their respective process, the addition of such a try-except scheme at the level of the API function is capable of comfortably handling any errors that happen during the execution of any of the three main processes (`store`, `retrieve` and `migrate`).

Furthermore, with such a setup, anywhere in the process errors may be raised and no matter whether they are raised in the generic adapter or in the BC-specific adapters, they will be caught at the same place in the code.

This means that modularity is ultimately not affected at all, as the only changes within the generic adapter or BC-specific adapters are the potential additions of code statements that programmatically raise errors, if certain conditions are met. Should such a condition apply to all supported BCs, it is checked for and raised in the generic adapter. If that is not the case it can simply be raised within the BC-specific adapter in question without affecting any of the other BC-speficic adapters.

One thing that needs to be pointed out is that the `migrate` API function does not actually need a try-except scheme, as it simply concatenates a `retrieve` with a `store` process, meaning if both of those implement a try-except scheme, error handling is automatically covered in `migrate` as well.

4.2.4 Redundancy Feature

Introduction and Potential Impact

The goal of the redundancy feature is simple; give users of Bifröst the ability to decide for each transaction made through the API whether it should not only be made on the specified BC, but also be submitted a second time on the PostreSQL database present within the Bifröst architecture.

This feature then would allow for the retrieval of transaction data that may no longer be obtainable from the BC it was initially stored on. One scenario this could be useful for is when the transaction was stored on a fork of the BC that was abandoned or rolled back.

As for how this feature should be implemented within the existing codebase of Bifröst, the philosophy is the same as with the other features. Implement it in such a way that it minimally impacts the three key requirements imposed on Bifröst (***flexibility***, ***modularity*** and ***ease of use***).

Starting with ease of use, the potential impact at its worst is the introduction of one additional parameter that is to be supplied to the `store` function exposed by the API, as the user only needs to indicate that not only one specific BC adapter but also a second adapter, the PostgreSQL adapter, is to be used. Since this parameter can be assigned a default value, it can even be made optional. As such, the impact on ease of use is minimal.

Flexibility is not affected. The reasoning here being that since the PostgreSQL adapter is used to facilitate the redundancy, and it is also available as an adapter during the standard store process, where it functions with all the types of data supported by other adapters, the type of data that can be stored with and without redundancy enabled does not change.

Modularity on the other hand, could potentially be an issue, depending on where within the Bifröst architecture the redundancy is taken care of. Should redundancy *e.g.,* be implemented at the level of the BC-specific adapters, then each adapter would have to have the ability to initiate a store process for another adapter, namely the PostgreSQL adapter.

The initiation of such a process has thus far however been reserved to the API functions, whilst the adapters simply execute whatever commands they receive from hose API functions. Allowing such a decision to be delegated to the BC-specific adapters means that any changes to how redundancy is initiated would have to be made for each existing adapter on an individual level as well. This would lead to poorer modularity.

Design

As established in the previous section, modularity is the main concern when implementing this feature. Thankfully, there is a simple way to design redundancy in such a way that it only affects the API function `store`, such that modularity is not affected at all.

This can be achieved with the use of recursion. In the context of Bifrösts redundancy feature this means that if the `store` function of the API is called with redundancy enabled, then the `store` function calls itself a second time, this time with the BC of choice being hard-coded to the PostgreSQL, such that that particular adapter is used for the second store process. This then results in the data being stored not only on the initially specified BC but also in the PostgreSQL database, which is exactly the goal the redundancy feature. To prevent infinite loops, the redundancy parameter is set such that redundancy is disabled for the recursive call.

4.3 Implementation

Whilst Section 4.2 serves the discussion of design decisions made on how the given features are to be implemented schematically, it does not comment about any aspects related to the code produced during the actual implementation of the features. As such it can be seen as generic in the sense that it does not take into consideration the properties of any given programming language with which a Bifröst implementation might be produced.

In contrast, this section focuses on the code written during the production of this thesis; thus, is intrinsically tied to the mechanics of the Python programming language. That is not to say that potential implementations of Bifröst in other programming languages are necessarily structured differently and that no parallels may be drawn, but rather that this section does not attempt to make such considerations in the first place.

4.3.1 General Refactoring

Before the new features were implemented, some minor refactoring took place with regard to where entries into the `transactions` table of the database are made. So far, the `store` method of the generic adapter had the task of making that entry, as can be seen in Listing 4.5 on line 8. Note that the `store` method has been heavily reduced to just the relevant lines of code.

Listing 4.5: Transactions Entry Location So Far

```
1  class Adapter(ABC):
2      # ...
3
4      @classmethod
5      def store(cls, text):
6          # ...
7
8          cls.add_transaction_to_database(transaction_hash)
9          return transaction_hash
10
11     # ...
12
13 class EthAdapter(Adapter):
14     # ...
15
```

```
16      @staticmethod
17      def add_transaction_to_database(transaction_hash):
18          database.add_transaction(transaction_hash, Blockchain.ETHEREUM)
19
20      # ...
```

This `add_transaction_to_database` method was then implemented by each BC-specific adapter separately, with the only difference between the implementations in the different adapters being the second argument, which specified which BC the transaction was made on (line 18).

This has the disadvantage of each BC-specific adapter having to provide an implementation of `add_transaction_to_database`, as well as mixing up the responsibilities of the API and the adapters. Recall Figure 2.5 from Section 2.3. The figure indicates that it should be the API which creates the entry, and not any of the adapters.

Hence, the implementation of Bifröst was changed, such that the API now creates those entries. Refer to Listing 4.6, line 6 to see the new implementation. The database's `add_transaction` method that was previously called from within the BC-specific adapter, is now called by the API's `store` function directly. Note that the number of parameters for the `add_transaction` method have changed in the meantime, hence the additional arguments in the refactored code, but it is still the same method.

Listing 4.6: Transactions Entry Location Refactored

```
1  def store(text, blockchain, pw=None, mlutiple_tx=False, redundancy=False):
2      # ...
3
4      for i in range(len(transaction_hashes)):
5          # ...
6          database.add_transaction(transaction_hashes[i], blockchain, salt,
               key, next_transaction)
7
8      # ...
9      return transaction_hashes
```

This change brings the implementation in line with what is shown in the architecture design depicted in Figure 2.5. Furthermore, the responsibilities of the API and the adapters are more clearly divided. The adapters are responsible for interacting with the BCs, hence what happens with the information gathered during such a store process (salt and hashes) is not relevant to the adapters, once those interactions are over. The API on the other hand is responsible for facilitating interactions with users and coordinating the required actions based on user input. In the author's view, tracking information about transactions by creating database entries is a coordination task and hence the responsibility of the API.

4.3.2 Encryption Feature

There are two distinct parts of the code that are discussed regarding the implementation of the data encryption feature. First, the implementation of the encryption and decryption

methods as well as the choice of cryptographic library used to do so is explored. Second, the way the aforementioned encryption and decryption methods are introduced into the existing adapters is explained for each of the three processes available through the API (store, retrieve and migrate).

Encryption and Decryption Methods

The cryptographic library chosen for this feature is called "cryptography" [71]. A number of reason led to this library being chosen; it is actively maintained, with the latest release at the time of writing this only being a number of days old, it provides a solid documentation [69] and it is used for a large number of encryption related Python tutorials, thus plenty of resources are available. On top of that, the stated goal of the developers is to make the cryptography the de facto standard library for Python [71].

The method of the generic adapter class used to perform encryption is shown in Listing 4.7. Note that compared to the code present in the Bifröst code-base, most of the comments have been removed in the code shown below. Note furthermore that the parameters used with regard to anything cryptography related, such as the generation of the salt or the Key Derivation Functions (KDFs), are those that have been used in examples from the library's documentation (which can be found in [69]). The general code structure is based on an example in the documentation that shows how passwords are to be used with this library [70].

Listing 4.7: Encryption Method

```
1  class Adapter(ABC):
2      # ...
3
4      @staticmethod
5      def encrypt(text, pw):
6          # generate salt, build kdf, generate keys
7          pw = pw.encode()
8          salt = os.urandom(16)
9          kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),
10                           length=32,
11                           salt=salt,
12                           iterations=100000,
13                           backend=default_backend())
14         kdf_2 = PBKDF2HMAC(algorithm=hashes.SHA256(),
15                            length=32, salt=salt,
16                            iterations=100000,
17                            backend=default_backend())
18         key_1 = base64.urlsafe_b64encode(kdf.derive(pw))
19         key_2 = base64.urlsafe_b64encode(kdf_2.derive(pw + key_1))
20         f = Fernet(key_2)
21         text_encrypted = f.encrypt(text.encode())
22         # return data, text_encrypted as string, salt and key_1 in byte-
               format
23         return text_encrypted.decode(ENCODING), salt, key_1
24
25     # ...
```

The encryption function takes two parameters, `text`, the data that is to be encrypted and `pw`, the password with which a key for encryption is to be built.

The task of the encryption method presented in Listing 4.7 is divided into three subtasks, *(i)* salt generation, *(ii)* KDF building, and *(iii)* key generation.

First, a salt needs to be generated. This is accomplished on line 8. Second, a KDF needs to be built, with which a key can be derived from the password and the generated salt. Note that the salt is built into the KDF, hence it needs to be passed as a parameter when constructing the KDF and is not used as a parameter on line 18 and 19 when the keys are being generated. Note furthermore that such a KDF can only be used for a single key-derivation, hence two KDFs.

Normally, a so called "deep copy" of the first KDF object would be created to create a new separate KDF object [43]. This however is not supported by the KDF object and will result in an error. Thus, a second KDF object has to be created from scratch, as shown in lines 14 - 17. Though speculation on the author's part, it is likely that this is due to the library not implementing a `__deepcopy__()` method itself for the KDF object, as that is required to add deep copy support to classes according to [43].

Third, two keys need to be generated, `key_1`, which will end up being stored and used for password verification, and `key_2`, which is used for encryption and decryption. Lines 18 and 19 show how the two keys are generated.

Finally, a *fernet* object `f` is built. This object then can be used to perform cryptographic operations such encrypting and decrypting data. It is then used to encrypt the data received in the `text` parameter of the encrypt method.

Once all that is done, the `encrypt` method returns three objects; the encrypted data in string format, using the 'utf-8' encoding, as well as the salt and `key_1`, the "verification key" discussed in Section 4.2.1 in byte format. The reason why the last two objects are passed in byte format is that the cryptography library requires the use of the byte format and will not work with strings. This is also the reason why the password is converted to byte format in line 7 and why the data is converted to byte format before encryption in line 21.

Moving on to the decryption method used in the generic adapter class, its implementation is shown in Listing 4.8.

Listing 4.8: Decryption Method

```
1  class Adapter(ABC):
2      # ...
3
4      @staticmethod
5      def decrypt(text, pw, salt, key_1):
6          # given received pw and stored salt & key, build key_2 to decrypt
7          pw = pw.encode()
8          kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),
9                           length=32,
10                          salt=salt,
11                          iterations=100000,
```

```
12                              backend=default_backend())
13          key_2 = base64.urlsafe_b64encode(kdf.derive(pw + key_1))
14          f = Fernet(key_2)
15          text_decrypted = f.decrypt(text.encode())
16          return text_decrypted.decode(ENCODING)
17
18      # ...
```

As already established when discussing the encryption method, the library requires all the data to be in byte-format, hence the conversions on lines 7 and 15, as well as the conversion back into a string format for the decrypted text on line 16.

The `decrypt` method takes four arguments, `text`, the data retrieved from the transaction specified by the user, `pw` the password provided by the user, as well as `salt` and `key_1` which are retrieved from the database, where they have been stored when initially performing the encryption. Note that the method assumes the password to be the correct one. Password verification is not its responsibility and is performed before this method is called.

The decryption method has two tasks. First, based on password, salt and `key_1`, `key_2`, the key which is used for encryption and decryption, is to be reconstructed (line 13). This also requires the reconstruction of the KDF (line 8 - 12). Second, the fernet object `f` needs to be built (line 14) and used to decrypt the encrypted data as seen on line 15. Once done, the decrypted data is returned in string format.

Integration into the Adapters

The encryption and decryption method of the generic adapter are integrated into the already existing processes for storing, retrieving and migrating data via the use of two wrapper methods, one for the `store` and one for the `retrieve` method of the generic adapter class. These wrappers are then called from the respective API function instead of the original `store` and `retrieve` method.

This enables the integration of new features via the use of the wrappers, whilst leaving the original methods unchanged. Whilst the processes of storing and retrieving data each require a wrapper method, the migrate process does not, as it internally consists of a combination of a retrieve and store process.

The code for the `store_wrapper` method is shown in Listing 4.9. Note that most functionality pertaining to other features such as *e.g.,* string splitting have been removed from the wrapper to allow closer focus on the data encryption feature.

Listing 4.9: `store_wrapper` Method

```
1  class Adapter(ABC):
2      # ...
3
4      @classmethod
5      def store_wrapper(cls, text, pw=None, multiple_tx=False):
6          salt = None
```

```
7           key = None
8           if pw:
9               text, salt, key = cls.encrypt(text, pw)
10
11          # ...
12          # size-checking, splitting if needed, collecting transaction_hashes
13          # ...
14          for i in range(nr_of_transactions):
15              # call original store method for each required tx separately
16              subtext = text[i*limit:(i+1)*limit]
17              transaction_hashes[i] = cls.store(subtext)
18
19          return transaction_hashes, salt, key
20
21      @classmethod
22      def store(cls, text):
23          # ...
24
25      # ...
```

There are two things that need to be highlighted with respect to the implementation of the `store_wrapper` method. The first thing to note is that the the data is encrypted, provided the user specified a password, before any other operations are done. This ensures that all subsequent operations already use the encrypted text and don't have to worry about *e.g.,* splitting the text and then having to use multiple calls to the encryption method.

The second aspect worth highlighting is that the wrapper allows for an additional parameter `pw` when compared to the original `store` method. This means that the original method is agnostic to encryption when it is called on line 17 from within its wrapper method.

The `retrieve_wrapper` method is similar in its approach. Its code can be seen in Listing 4.10. In addition, the helper method `verify_password` is also displayed, as it is used in the wrapper to verify passwords.

Listing 4.10: `retrieve_wrapper` Method

```
1   class Adapter(ABC):
2       # ...
3
4       @classmethod
5       def retrieve_wrapper(cls, transaction_hashes, pw=None, salt=None, key=
            None):
6           text = ""
7           for i in range(len(transaction_hashes)):
8               text += cls.retrieve(transaction_hashes[i])
9           if pw and cls.verify_password(pw, salt, key):
10              text = cls.decrypt(text, pw, salt, key)
11          return text
12
13      @classmethod
14      def retrieve(cls, transaction_hash):
15          # ...
```

```
16
17      @staticmethod
18      def verify_password(pw, salt, key):
19          if salt is None or key is None:
20              return False
21          kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),
22                           length=32,
23                           salt=salt,
24                           iterations=100000,
25                           backend=default_backend())
26          key_attempt = base64.urlsafe_b64encode(kdf.derive(pw.encode()))
27          return key == key_attempt
28      # ...
```

Notice first that as with the methods used for storing data, the wrapper method allows for additional parameters, whilst the original method is left agnostic to the new features. The wrapper uses the original `retrieve` method on line 8 when retrieving the text as it is stored on a given BC.

Should the text have been encrypted, the wrapper will have received the necessary values for `pw`, `salt` and `key` when called, and thus can make use of the `decrypt` method discussed previously.

Note that before performing the decryption, any password is verified using the method `verify_password`. That method builds a key given the password specified by the user during the retrieval process, analogous to the `encrypt` and `decrypt` methods discussed earlier.

The resulting key is then compared against the verification key (`key`). Should they be equal, then the password is the same as the one that was used when initially encrypting the data and is considered verified. Should the password fail the verification, then decryption is not attempted but instead the encrypted text is returned, as the value assigned to the `text` variable remains unmodified by line 8.

Finally, a look at how the encryption feature is integrated into the migration process is required. It might be tempting to assume that encryption and decryption is not relevant for the migration process. After all, migration first uses the retrieval process and then the store process to create a copy of a given piece of data on another BC.

However, if the data is retrieved from its new location was encrypted it still has to be able to be decrypted. This means that the encryption information stored for each transaction, namely the salt used for key generation and the "verification key" used for password verification need to be available.

To achieve that, one could simply use the normal procedures for retrieving and storing data. In the case of encrypted data, this would then require the user initiating the migration process to provide the password, as otherwise during the retrieval the data would not be decrypted and not re-encrypted again during the subsequent storing process. This approach then would entail allowing for an additional parameter, `pw` to be provided when making a call to the APIs migrate method.

There are two drawbacks with this approach. First, it lessens the ease of use of Bifröst as the user is required to provide more information in the form of a password, which he should not have to provide. Migrating data in the context of Bifröst, whether encrypted or not, is understood by the user as simply copying it in whatever form it is stored to a new location. There is no clear reason why this should require a password from user's perspective. Given the transaction hash required to specify which data to move, anyone could perform that migration outside of the API, so requiring a password when doing so through the API does also not provide any security benefits.

Second, the data ends up in its encrypted form at the new location, making both the decryption and subsequent encryption that would occur with this approach redundant and operations that unnecessarily increase the computational complexity of the migration process.

Thus a different approach is taken. No password is provided by the user, meaning the data does not get decrypted and encrypted again during the migration procedure, eliminating unnecessary cryptographic operations. If the data is in fact encrypted, the required encryption information is read from the database for the data which is retrieved during the migration procedure and added after the fact to the new transaction entry in the database that was created as per normal procedure when the data was stored on the new BC.

Listing 4.11 shows some code snippets that illustrate this approach. Note that the functions shown here are not methods belonging to any of the adapter classes. Rather these are excerpts from the three API functions exposed to users. They can be found in the file `api.py` in the Bifröst code-base [47].

Listing 4.11: Encryption Compatible Approach To Migration

```
1  def migrate(transaction_hash, blockchain, multiple_tx=False):
2      value = retrieve(transaction_hash)
3      new_hashes = store(value, blockchain, multiple_tx=multiple_tx)
4      _, salt, key, _, _ = database.find_transaction_information(
           transaction_hash)
5      for i in range(len(new_hashes)):
6          database.add_encryption_info(new_hashes[i], salt, key)
7      return new_hashes
8
9  def store(text, blockchain, pw=None, multiple_tx=False, redundancy=False):
10     # ...
11     transaction_hashes, salt, key = adapter.store_wrapper(text, pw,
           multiple_tx=multiple_tx)
12     salt = salt if salt is not None else ""
13     key = key if key is not None else ""
14     for i in range(len(transaction_hashes)):
15         # ...
16         database.add_transaction(transaction_hashes[i], blockchain, salt,
               key, next_transaction)
17
18     # ...
19     return transaction_hashes
20
21 def retrieve(transaction_hash, pw=None):
22     # ...
23     if pw:
```

```
24          text = adapter.retrieve_wrapper(transaction_hashes, pw, salt, key)
25      else:
26          text = adapter.retrieve_wrapper(transaction_hashes)
27      # ...
28      return text
```

The API functions for the store and retrieve procedures are defined on lines 9 and 21 respectively. In both of the functions most of the code has been removed, but the calls to the respective wrapper methods of a given adapter can still be seen.

`Migrate` itself calls the API functions `retrieve` and `store` on lines 2 and 3 respectively, which then results in those functions making calls to their respective wrappers. Since the function for the migration procedure does not accept a password parameter, `retrieve` ends up being called without a password (line 26), thus not decrypting data, even if it is encrypted. The data is then stored as it was received from the `retrieve` call, again without a password (line 11), thus no encryption operations are performed.

As no encryption was performed, the values for `salt` (line 12) and `key` (line 13) are set to the empty string, which in Bifröst's database results in them having no value assigned to them, once the entry for the new transaction is made (line 16).

If the migrated data was encrypted, then the values for `salt` and `key` need thus to be added after the fact, which happens on line 4, where they are retrieved from the information stored about the initial transaction and on line 6, where the entry for the new transaction is enriched with the necessary `salt` and `key` values.

4.3.3 String Splitting Feature

In this part of the thesis, insight is provided into how string splitting is integrated into the store process, how string reconstruction is implemented in the retrieve process and finally how the migrate process makes use of both splitting and reconstruction.

Splitting During Store Process

Listing 4.12 shows an excerpt of the generic adapter class. More precisely, two helper methods (`get_text_size` and `determine_nr_of_tx`) are shown which are used in the store process which is performed by the `store` and `store_wrapper` methods.

Listing 4.12: String Splitting Implementation in Adapter

```
1  class Adapter(ABC):
2      # ...
3
4      @classmethod
5      def get_text_size(cls, text):
6          return len(text.encode('utf-8'))
7
8      @classmethod
```

```
 9      def determine_nr_of_tx(cls, text, limit):
10          size = cls.get_text_size(text)
11          return math.ceil(size/limit)
12
13      @classmethod
14      def store_wrapper(cls, text, pw=None, multiple_tx=False):
15          # encryption takes place if needed
16          limit = TRANSACTION_SIZE_LIMITS[cls.chain.value]
17          nr_of_transactions = cls.determine_nr_of_tx(text, limit)
18          transaction_hashes = [""]*nr_of_transactions
19          # if multiple transactions disabled
20          if nr_of_transactions != 1 and not multiple_tx:
21              raise TransactionSizeError
22          for i in range(nr_of_transactions):
23              # call original store method for each required tx separately
24              subtext = text[i*limit:(i+1)*limit]
25              transaction_hashes[i] = cls.store(subtext)
26          return transaction_hashes, salt, key
27
28      @classmethod
29      def store(cls, text):
30          # perform transaction
31          return transaction_hash
32
33      # ...
```

The entry point into the process within the adapter is the wrapper method, which is called by the API's `store` function as shown in Listing 4.13. For now however let the focus be on what happens in the adapter class.

Lines 16 to 17 show how the size limit is retrieved and passed alongside the text the helper method `determine_nr_of_tx`, whose task is to determine how many transactions are needed for a given BC limit and the size of the supplied text. The size of the supplied text is calculated at line 10, where the other helper method, `get_text_size` is used to essentially just count the number of characters contained within the given text (credit for the code of the method `get_text_size` goes to [55]). With both sizes present, the number of needed transactions is calculated and returned on line 11.

With both the data size limit as well as the number of needed transactions known within the wrapper method, an array is constructed to hold the hashes that will be returned from the individually submitted transactions (line 18).

If the user did not enable data splitting, by overwriting the default value of `multiple_tx`, if multiple transactions are needed, the process is aborted and an error is raised (line 21), displaying to the user that the data supersede the capacity of a single transaction.

If data splitting is enabled, or no slitting is needed, for each needed transaction, the subtext is defined, as shown in line 24. For this, string indexing is used with the BC-specific transaction size limit being the demarcation for which parts of the text are to be extracted for each transaction.

Once the a subtext is extracted, the original `store` method of the adapter is called, which creates the actual transaction and sends it to the specified BC (line 25). The returned

hash is stored into the array prepared on line 18. Since the text is split from front to back and the transactions are done sequentially, the order of the hashes stored in the `transaction_hashes` array is the same order in which the subtext can be concatenated to reconstruct the original text.

This array of hashes is returned to the caller (API `store` function), alongside information which was used during a potential encryption process (line 26).

Having now established, how the string splitting works in the adapter class, a look at how the presence of multiple transaction hashes and the respective tracking of which transactions together belong to a given piece of data are implemented is required. Listing 4.13 shows parts of the APIs `store` function found in `api.py`.

Listing 4.13: String Splitting Tracking in API

```
1  def store(text, blockchain, pw=None, multiple_tx=False, redundancy=False):
2      adapter = Adapter[blockchain]
3      transaction_hashes, salt, key = adapter.store_wrapper(text, pw,
           multiple_tx)
4
5      # ...
6
7      for i in range(len(transaction_hashes)):
8          next_transaction = transaction_hashes[i+1] if i < (len(
               transaction_hashes)-1) else transaction_hashes[0]
9          database.add_transaction(transaction_hashes[i], blockchain, salt,
               key, next_transaction)
10
11     # ...
12
13     return transaction_hashes
```

Of note is primarily what happens on lines 7 to 9. A loop is constructed, which for each of the hashes returned on line 3 from the wrapper method of the adapter performs two distinct actions.

First, on line 8, it is established what should be specified in the `next_hash` field of the database when creating the entry for a given transaction. Should the current iteration index of the loop be smaller than the total number of hashes contained in the array (-1 due to indexing starting at 0), then the current iteration index does not yet point to the last hash in the array. Thus, the value for `next_transaction` which will be stored in the database table's `next_hash` field is set to next hash in the array.

Should that condition evaluate to false instead, then the value is set to the first hash in the array. In case of the call on line 3 returning multiple hashes in the array, this would lead to the "loop of pointers" being closed. Why this is important is explained in detail in Section 4.3.3, but to quickly reiterate, it allows for the identification of all transactions involved in a given store process based on any single transaction hash.

Alternatively, if only one transaction was needed to store the data, then the transaction essentially points to itself, making it clear that no other transactions were involved. Having established the value to be used to indicate the next transaction, the entry for the transaction is made in Bifröst's database (line 9).

Reconstruction During Retrieve Process

Reconstruction of a split string is also an act that is shared between the API and the adapter class, with the task of the adapter being comparatively simple. Listing 4.14 shows the section of code from the generic adapter class where reconstruction happens on the level of the adapter.

Listing 4.14: String Reconstruction in Adapter

```
1  class Adapter(ABC):
2      # ...
3
4      @classmethod
5      def retrieve_wrapper(cls, transaction_hashes, pw=None, salt=None, key=
           None):
6          text = ""
7          for i in range(len(transaction_hashes)):
8              text += cls.retrieve(transaction_hashes[i])
9
10         # ... potential decryption
11         return text
12
13     @classmethod
14     def retrieve(cls, transaction_hash):
15         # retrieve data from transaction
16
17     # ...
```

Given an array of transaction hashes, which the adapter can take for granted to be *(i)* a group of transactions that belong together in forming a string from their respective substrings and *(ii)* to be provided in the correct order, the `retrieve_wrapper` method runs a loop (line 7) where for each of the hashes the original `retrieve` method is called (line 8), which in turn retrieves the string stored on the specified transaction.

Since the hashes are provided in the correct order to the adapter, reconstruction simply is a matter of concatenating the retrieved substrings in the order in which they are retrieved, as can be seen on line 8. Once done, the now reconstructed string may potentially have to be decrypted and is then returned to the API function `retrieve` from where the wrapper method was originally called.

Having established how the substrings are used to reconstruct the complete string within the adapter, it must now be shown how the group of transaction hashes is *(i)* identified based on user input and *(ii)* how those transaction hashes are put into the correct order, such that the adapter can blindly concatenate the retrieved substrings.

Listing 4.15 shows parts of the code of the `retrieve` API function. The user input used to identify the transaction(s) needed to retrieve a string is a single transaction hash, as can be seen in the function signature (line 1).

Listing 4.15: String Reconstruction in API

```
1  def retrieve(transaction_hash, pw=None):
```

```
2       blockchain, salt, key, time_stamp, next_transaction = database.
            find_transaction_information(transaction_hash)
3       adapter = Adapter[blockchain]
4
5       # ... handle potential encryption
6
7       # retrieve all needed hashes
8       transaction_hashes = [transaction_hash]
9       time_stamps = [time_stamp]
10      earliest_time_stamp = time_stamp
11      while next_transaction != transaction_hashes[0]:
12          transaction_hashes.append(next_transaction)
13          _, _, _, next_stamp, next_transaction = database.
                find_transaction_information(next_transaction)
14          time_stamps.append(next_stamp)
15          if next_stamp < earliest_time_stamp:
16              earliest_time_stamp = next_stamp
17
18      # reorganization of retrieved hashes according to time_stamps
19      while time_stamps[0] != earliest_time_stamp:
20          transaction_hashes.append(transaction_hashes.pop(0))
21          time_stamps.append(time_stamps.pop(0))
22
23      # call wrapper with needed parameters
24      if pw:
25          text = adapter.retrieve_wrapper(transaction_hashes, pw, salt, key)
26      else:
27          text = adapter.retrieve_wrapper(transaction_hashes)
28
29      # ...
30
31      return text
```

The function then first retrieves the data associated with the transaction associated with the hash specified the user from the database (line 2). After that a number of variables are set up: `transaction_hashes`, initialized with the hash specified by the user, will be used to store the hashes of all transactions involved when originally storing the data. `time_stamps` will hold the time stamps for each of the transactions and `earliest_time_stamp` will be used to establish which of the transactions happened first, such that the proper order can be reproduced.

Now, the "loop of pointers", constructed previously when creating the `next_hash` entries in the database, is unravelled. Recall, that each transaction entry in the database carries a pointer to the next transaction, with the last transaction closing the loop by pointing to the first one.

Since it is not clear, where in that loop the transaction the user specified with `transaction_hash` lies, the hashes retrieved in line 13 are in the correct order relative to each other, but not necessarily in the absolute correct order, as the user may have specified *e.g.,* the second to last transaction out of a group of *e.g.,* 5 transactions.

To solve that problem, the time stamps retrieved on line 13 are stored on line 14, whilst the `earliest_time_stamp` is continuously updated on line 16. Given the time stamps, with

the respective index of a given time stamp being equal to the index of the corresponding transaction hash in `transaction_hashes`, the transactions are reorganized in the while-loop on line 19.

Simply put, if the first time stamp entry in the `time_stamps` array does not correspond to the earliest stamp out of the group of retrieved time stamps, then the elements at the first index of both the `transaction_hashes` and `time_stamps` arrays are moved to the back of the array, leading to a new set of first elements. This rotation happens until the time stamps align according to the condition in line 19, which in turn guarantees that the hashes in `transaction_hashes` are in the correct order in an absolute sense.

Once that order is established, the `retrieve_wrapper` method of the adapter is called, initiating the process discussed previously, leading to the substrings being concatenated, potentially decrypted and returned to the APIs `retrieve` function.

String Splitting and Reconstruction During Migration

The API's `migrate` function is not actively involved in either splitting nor reconstruction, as it internally uses the API functions `retrieve` to retrieve data from its origin and `store` to store it in the new location. Seeing as splitting is taken care of during the store process and reconstruction is taken care of when retrieving data, migrate does not have to perform any logic itself to directly facilitate splitting, it simply must forward whether or not splitting is allowed to the `store` function via the parameter `multiple_tx`, as seen on line 3.

Listing 4.16: String Splitting During Migration

```
1 def migrate(transaction_hash, blockchain, multiple_tx=False):
2     value = retrieve(transaction_hash)
3     new_hashes = store(value, blockchain, multiple_tx=multiple_tx)
4     # handle potential encryption -> store salt & key in DB for the new
          transaction as well
5     _, salt, key, _, _ = database.find_transaction_information(
          transaction_hash)
6     for i in range(len(new_hashes)):
7         database.add_encryption_info(new_hashes[i], salt, key)
8     return new_hashes
```

Furthermore, since the transaction size limits for different BCs are different, meaning that *e.g.,* the data may have been sufficiently small to be stored using one transaction on the original BC, but requires multiple transactions on the new BC, `migrate` must take care to now add encryption information (`salt` and `key`) to all of the potentially numerous transactions created on the BC the data is migrated to. As such, when receiving the array of new hashes from `store` (shown in Listing 4.16 on line 3), a loop must now be used to modify the database entires (lines 6 and 7).

4.3.4 Error Handling Feature

First, Listing 4.17 shows an example of how custom errors have been implemented in Bifröst. A separate file (`custom_errors/custom_errors.py`) has been created to house them separately from the rest of the code.

Listing 4.17: Custom Error Implementation

```
1 error_messages = {
2     CustomErrors.BC_NOT_FOUND_ERROR: "\tBlockchainNotFoundError: The
          specified Blockchain has not been found."
3 }
4
5 # ...
6
7 class BlockchainNotFoundError(Exception):
8     def __init__(self, msg=error_messages[CustomErrors.BC_NOT_FOUND_ERROR],
          *args, **kwargs):
9         super().__init__(msg, *args, **kwargs)
```

Lines 7 to 9 show how such a custom error is defined, by inheriting form the built-in `Exception` type. The code is adapted from [58]. The constructor is set up such that it uses a default error message, which is passed into the `msg` parameter. If no argument is given when the error is actually raised within the code, the default message is used, if an argument is given, *e.g.,* the string "`Error 123`" then that is used as error message instead.

The default messages for the different custom errors are defined within the `error_messages` object on line 1, such that the constructor on line 8 is kept more tidy. Note the use of an enum (`CustomErrors`) to where each new custom error has to be defined initially.

Listing 4.18 then shows how the error handling setup looks like at the example of the `store` API function, and how custom errors can be raised within the code.

Listing 4.18: Error Handling Scheme

```
1  def store(text, blockchain, pw=None, multiple_tx=False, redundancy=False):
2      try:
3          # ... perform store process
4      except BlockchainNotFoundError as error:
5          sys.tracebacklimit = 0
6          logging.error(error)
7          exit()
8      except Exception as error:
9          # treat any non-specified errors here
10         sys.tracebacklimit = 1000  # default
11         logging.exception(error, exc_info=True)
12         exit()
13
14 def __identify_blockchain(blockchain):
15     try:
16         adapter = Adapter[blockchain]
17         return adapter
18     except:
19         raise BlockchainNotFoundError
```

Line 4 shows how the error Handling for a specific error is set up. Note that the stack-trace is essentially removed on line 5 to keep the display for the error message simple. Furthermore, the `logging.error` option is chosen to display the error as it, in contrast to the `logging.exception` option only outputs the string of the error message.

For each error that is known to potentially occur, such an `except` block may be written to target it specifically. Here, since Bifröst in its current iteration does not actually return a HTTP-response to its users, the error message is simply logged and the program terminated (line 7).

To ensure that errors that are not targeted specifically such as the `BlockchainNotFoundError` on line 4, a generic `except` clause is defined on line 8 that will catch them. There, the stack-trace has not been shortened, since it is not clear which error occurred, hence the full output of the trace is desirable. To facilitate this, `logging.exception` is used here.

Finally, `__identify_blockchain` on line 14 is a helper-function called within the try block of the `store` function (line 2). If anything within that try block raises an error it will either be caught by an error-specific `except` clause or by the generic one. In the case of the helper-function, it uses a `try-except` scheme itself to detect invalid blockchain identifiers and ultimately raise the error (line 19), although errors may also be raised programmatically by other means, *e.g.,* by checking for conditions in an `if` statement.

4.3.5 Redundancy Feature

The code showing how redundancy is implemented is shown in Listing 4.19. As discussed in the design section about the redundancy feature (Section 4.2.4), the feature is implemented solely within the `store` function of the API. Note that only those parts of the `store` functions are shown here that pertain to the redundancy feature.

Listing 4.19: Redundancy Implementation

```
1  def store(text, blockchain, pw=None, multiple_tx=False, redundancy=False):
2      adapter = Adapter[blockchain]
3      transaction_hashes, salt, key = adapter.store_wrapper(text, pw)
4
5      # ...
6
7      if redundancy:
8          transaction_hashes.append(store(text, Blockchain.POSTGRES, pw=pw,
               multiple_tx=multiple_tx, redundancy=False)[0])
9
10     # ...
11
12     return transaction_hashes
```

On line 1, the signature of the `store` function can be seen. The default value for redundancy is set to `False`, meaning redundancy has to be explicitly enabled by users. Lines 7 and 8 show what happens if redundancy is enabled. The `store` function calls itself

a second time after having run through the normal store procedure (line 3) for the BC specified by the user.

The `blockchain` parameter is hard-coded to hold the enum value that specifies the PostgreSQL adapter, whilst the `redundancy` parameter is set to `False` to prevent infinite loops. The `text`, `pw` and `multiple_tx` parameters meanwhile are left as they were when the function was initially called by the user, such that the data and encryption password are the same for the redundancy transaction as for the initial transaction on the specified BC.

This results in an entirely new store process being initiated and the data within `text` being stored as a backup of sorts in Bifrösts database. Once that process has run its course, it returns the resulting transaction hash to the original store process, where it is integrated into the already present array of transaction hashes that resulted from storing the data on the specified BC. The array holding these transaction hashes then is returned to the user.

Chapter 5

Evaluation

5.1 Evaluation Setup

The performance and size evaluations of the encryption feature described in this chapter were done on a MacBook Pro (2017) with a QuadCore Intel Core i7 CPU at 2.8 GHz and 16 GB of RAM. The extended version of Bifröst [47], which includes all the features discussed in Chapter 3, was used to perform the operations.

Unless stated otherwise, randomly generated passwords of string length 10 were used, with data being collected and aggregated over 30 runs for each of the evaluations. All Evaluations were performed on a randomly generated strings of Byte sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1014, 2048 and 4096.

Note at this point, that any Byte sizes mentioned refer to the Byte size of only the sequence of characters and not of the respective Python string objects. The Python string objects store a number of additional information and thus consuming more memory [55].

For the performance evaluations, a system warm-up was done, which was achieved by essentially running each evaluation twice in a directly subsequent fashion, with the first pass being the warm-up and the second pass being the actual evaluation which led to the data presented in this chapter.

5.2 Encryption Size Overhead

The encryption method chosen to implement the encryption feature leads to an increase of the data size for the encrypted strings. Figure 5.1 shows the size of the strings before and after encryption.

Notably, the increase in size after encryption gets more significant in absolute terms the larger the initial string gets, whilst the smaller strings experience a more drastic relative increase in size.

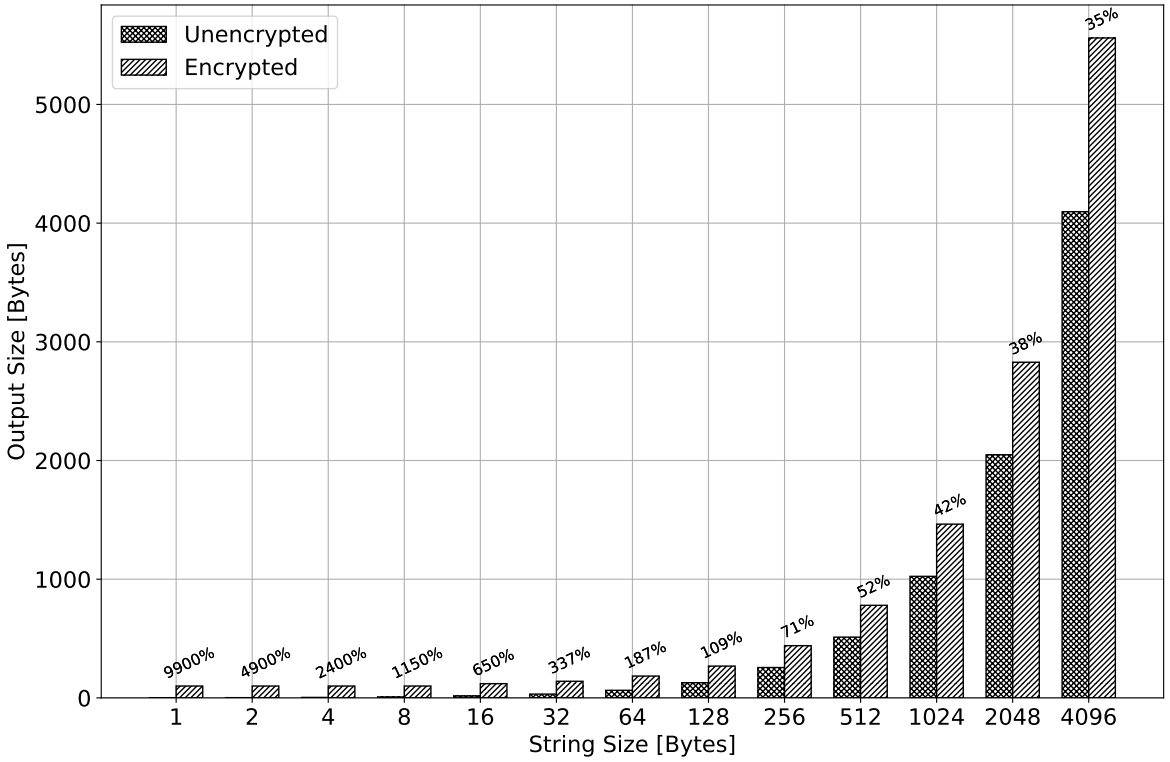


Figure 5.1: Bifröst Encryption Size Overhead

This increase in size is undesirable, as all of the supported BCs, with the exception of Ethereum and perhaps IOTA impose prohibitively small transaction size limits, as shown in Table 4.4 in Section 4.2.2. Under those restrictions all but a handful of BCs require multiple transactions to be able to store a single encrypted character. Thus, in particular the size-increase for the small string sizes is problematic, as for larger strings the use of multiple transactions is required for most BCs anyway.

There is a potential way to mitigate this issue, by using so called "format-preserving encryption" and it can quite easily be integrated into the existing encryption process. Section 5.2.1 gives an overview over what format-preserving encryption is, as well as the reasoning for why it was not selected for implementation.

5.2.1 Format-Preserving Encryption

As the name might suggest, format-preserving encryption produces ciphertext that maintains the format of the plaintext [7]. [7] give the example of encrypting a 16 digit credit card number which results in a 16 digit ciphertext. This would lend itself nicely to the scenario of Bifröst, where it is critical to maintain string sizes as small as possible due to the transaction size limits.

There exists a Python library, "pyffx" [19], which provides an implementation of the format-preserving, Feistel-based encryption mechanism discussed in [7]. It can be inte-

grated without many changes into the existing encryption feature of the extended Bifröst [47]. Listing 5.1 shows how that integration can be achieved.

Listing 5.1: Incorporating pyffx

```
1  class Adapter(ABC):
2      # ...
3
4      @staticmethod
5      def encrypt(text, pw):
6          # generate key_1 as in current scheme
7          pw = pw.encode()
8          salt = os.urandom(16)
9          kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),
10                          length=32,
11                          salt=salt,
12                          iterations=100000,
13                          backend=default_backend())
14         key_1 = base64.urlsafe_b64encode(kdf.derive(pw))
15         # use key_1 as password for pyffx scheme
16         alphabet_props = ascii_letters + whitespace + digits
17         e = pyffx.String((key_1+pw), alphabet=alphabet_props, length=len(
               text))
18         encrypted = e.encrypt(text)
19         return encrypted, salt, key_1
20
21     # ...
```

The encryption scheme would remain unchanged up to and including the generation of `key_1`, which is the "verification key" used for password verification. Once `key_1` is generated, instead of deriving `key_2` and building a *fernet* object, a *pyffx* object is generated based on the concatenation of `key_1` and the password, a given alphabet and the length of the plaintext (line 17). This object then is used to encrypt the plaintext string in `text`.

Such a scheme maintains the security by requiring `salt`, `pw` and `key_1` as input for the pyffx object and thus an adversary that manages to get access to Bifröst's database, hence has access to the stored `salt` and `key_1`, still requires the password which is never stored. This is in line with the way encryption is currently implemented in Bifröst, thus would not require any changes to what information is stored and how it is stored and mechanics such as password verification can be modified to remain functional.

The definition of the alphabet here includes all ASCII letters, possible whitespaces as well as digits. This does not have to be the case, but it does at a minimum have to include all characters used in the plaintext string within `text`.

There are two reasons why the decision was made against the addition of format-preserving encryption to the extended Bifröst. First, format-preserving encryption does leak some information about the plaintext through its cyphertext, such as the length of the plaintext (since it is maintained in the cyphertext), as well as the alphabet (meaning the set of characters used) of the plaintext, if the most minimal alphabet is used when building the pyffx object on line 17. The most minimal alphabet being the exact alphabet of the plaintext.

This does relate to the concept of "semantic security" which roughly speaking states that a piece of cyphertext is semantically secure if an adversary can not learn any information from it [56]. There is a caveat with the definition of security as it assumes the length of the plaintext to be public knowledge [56]. Hence the fact that the pyffx encryption scheme as shown above leaks the length of the plaintext is not seen as a violation of semantic security.

This leaves the issue that an adversary might derive the alphabet of the plaintext from the ciphertext and hence could conclude that *e.g.,* a purely numerical cyphertext of length 16 is likely an encrypted credit card number. In the author's view this can be mitigated by simply using a generic alphabet as is done in Listing 5.1 on line 17. However, the author has not been able to procure any sources for this and lacks himself the theoretical background knowledge to make such a claim, hence this is to be seen as an educated guess.

The second problem with the pyffx approach is that the library itself *(i)* seems to no longer be kept up to date (latest release on May 12, 2019) and *(ii)* has seen very little adoption, based on its GitHub statistics [19]. This is in sharp contrast with the "cryptography" library used for the current encryption scheme, which, at the time of writing this, has had its latest update on February 16, 2021 and has seen much wider adoption [69].

It is that second issue that really prevents the adoption of the encryption scheme shown in Listing 5.1, as in the author's opinion the issue of semantic security can be circumvented with the choice of a generic-enough alphabet. Should there at some point be a Python library for format-preserving encryption that is kept up to date and sees wider adoption, then in the author's view a switch to a scheme as shown in Listing 5.1 is advisable.

5.3 Encryption Performance

This section serves the evaluation of how encryption affects the performance of storing a piece of data on a BC. For all evaluations in this section that include making transactions, Ganache [14] running in a Docker [18] container was used to simulate an Ethereum BC.

First, in Section 5.3.1 the impact of password choices on encryption performance are explored. Note that here purely the performance of the `encrypt` method provided by the generic adapter is measured, no transactions were made.

After that, Section 5.3.2 highlights the relative performance of storing data with and without encryption. For this the simulated Ethereum BC that was described previously was used to perform transactions on.

5.3.1 Performance Impact of Password Choice

Four different configurations were evaluated with respect to their performance, each on strings of the same set of predefined lengths as discussed in Section 5.1. The configurations were as follows:

1. A set of 10 randomly generated passwords of a fixed length were evaluated as to their average performance on a fixed set of strings (the same strings were used for all passwords and for all runs). The results are displayed in Figure 5.2a

2. A set of 10 randomly generated passwords of random lengths between 4 and 20 characters were evaluated as to their average performance on a fixed set of strings. The results are displayed in Figure 5.2b

3. A single password of fixed length was evaluated as to its average performance on a fixed set of strings. The results are displayed in Figure 5.2c

4. A single password of fixed length was evaluated as to its average performance on a set of strings of default predefined lengths that was generated on a per-run basis. The results are displayed in Figure 5.2d

All Figures of the individual configurations were plotted on the same Y-axis range to allow for direct comparison. Figure 5.3 meanwhile display all of the four configurations side by side, with outliers being shown, hence the somewhat drastic change on the Y-axis range.

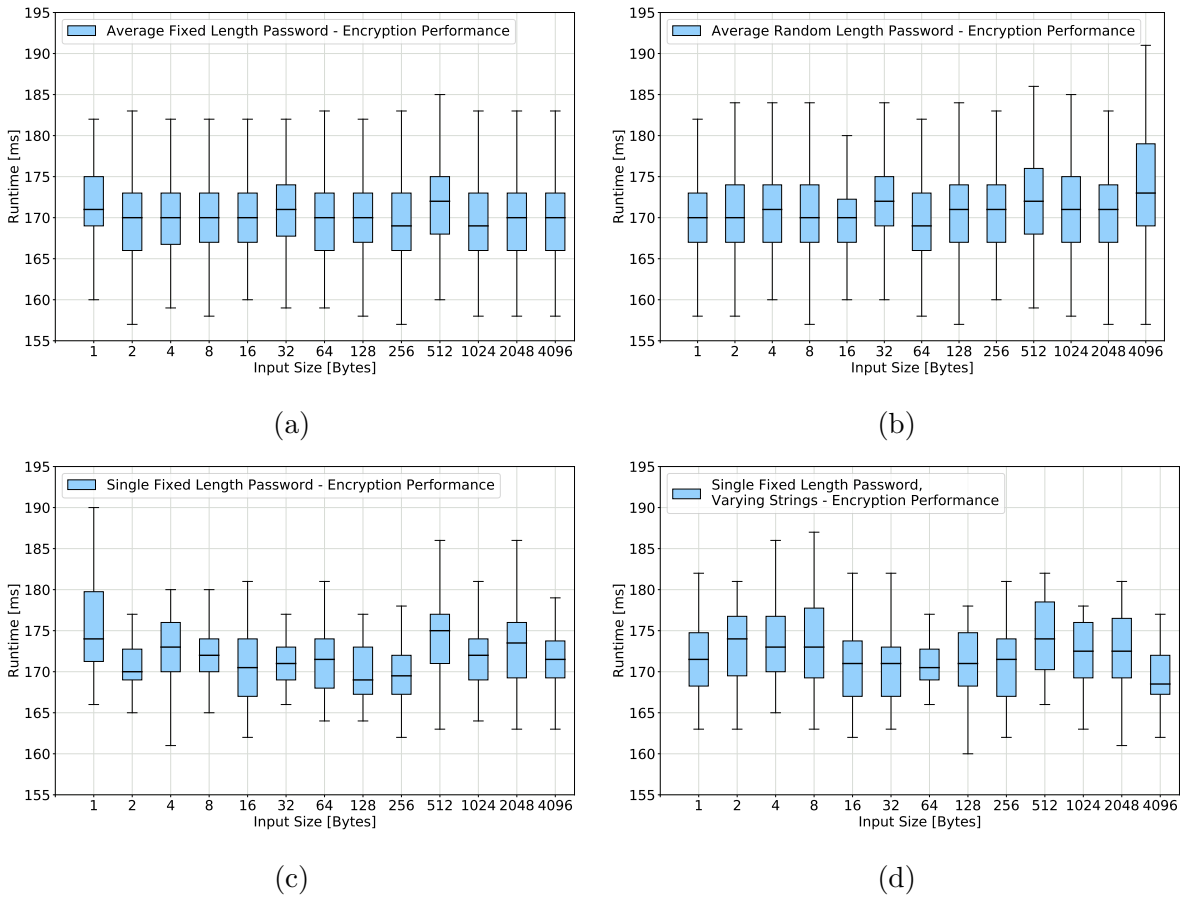


Figure 5.2: Comparison of the Performance Impacts of different password configurations, with configuration (a) being randomly chosen passwords of fixed lengths, (b) being randomly chosen passwords of random length, (c) a single fixed length password and (d) also a single fixed length password but with new strings to be encrypted for each run.

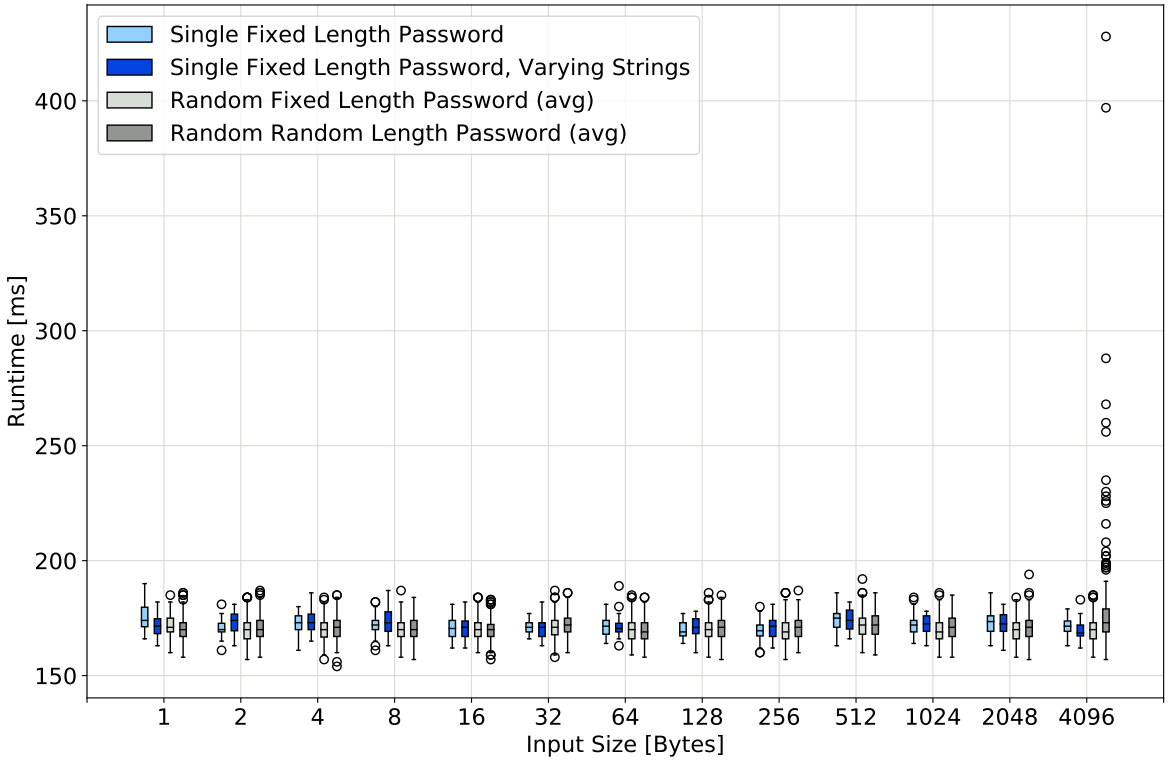


Figure 5.3: Bifröst Encryption Performance: Comparison of Password Configurations

A number of observations can be made when comparing the different configurations amongst each other.

- The choice of the underlying strings does not seem to markedly impact performance, even as they increase in size (comparing configurations 3. and 4.). This can also be seen in Figure 5.4 in Section 5.3.2 when comparing the performance of store processes with and without encryption.
- Disregarding outliers, the length of the passwords does not seem to have a significant impact on encryption performance (comparing configurations 1. and 2.).
- Not disregarding outliers, varying the size of the passwords does lead to a marked increase in the amount of outliers on the top end, leading to potentially significantly slower encryption times (comparing configurations 1. and 2.), see Figure 5.3.
- Disregarding outliers, the choice of password across all four configurations leads to remarkably consistent encryption performance.

This all means that encryption performance is essentially constant, regardless of the password configuration. Therefore, for the comparison of store processes with and without encryption there is no need to account for different password configurations.

5.3.2 Encryption Performance

In this section, the performance of storing data on a simulated Ethereum BC with and without encryption is compared. As Section 5.3.1 has shown that the choice of password is essentially inconsequential, the default evaluation setup described in Section 5.1 is used.

Figure 5.4 shows the comparative performance of storing data via Bifröst with and without encryption. Figures 5.5a and 5.5b meanwhile show the performance of store processes only with respectively without encryption.

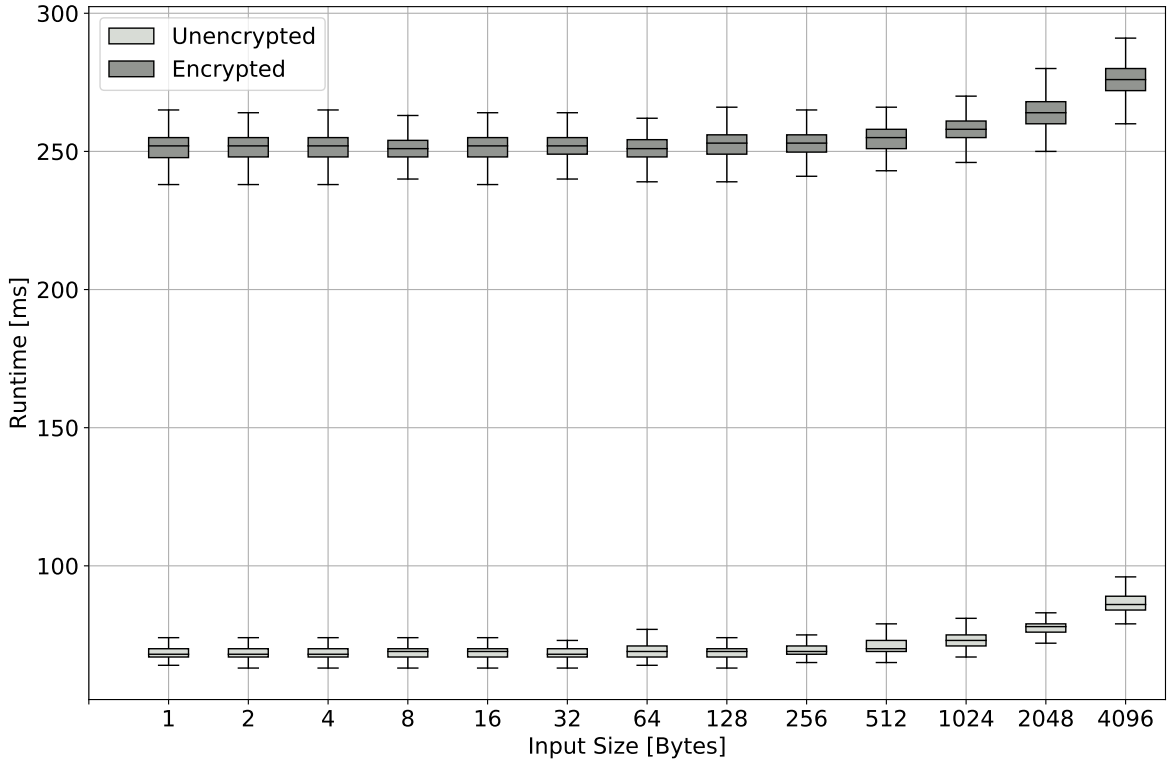
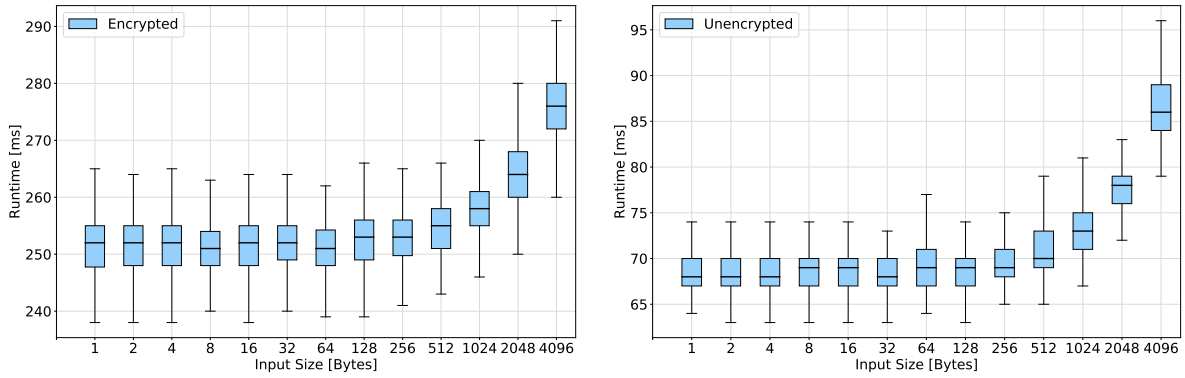


Figure 5.4: Bifröst Encryption Performance Comparison

The take-away of the performance analysis is that the cost for performing encryption is more or less constant, regardless of the size of the string that is encrypted. The increase in performance cost with larger string sizes is largely due to the an increase in performance cost of the `store` action itself, rather than due to an increase in performance cost of encrypting the data.

This is shown by the relatively consistent absolute distances between data points for the process with and without encryption in Figure 5.4 across all sizes of the input strings.



(a) (b)

Figure 5.5: Performance of the Bifröst `store` process with encryption in (a) and without encryption in (b).

5.4 String Splitting Evaluation

The evaluation of the string splitting feature is done by showing a how information about a split piece of data is stored in Bifröst's database during a `store` action, how that data is used during a `retrieve` action and how migrating a split data to a BC which has a different transaction size limit works.

The example scenario is as follows:

1. A user wants to store a string of size 65 on a BC with a transaction size limit of 28 Bytes. The user allows Bifröst to perform multiple transactions by enabling string splitting. Due to the transaction size limit a total of 3 transactions will be required to perform the store operation.
2. The same user wants to retrieve the stored data some time later, which requires that Bifröst reconstruct the entire string based on a single transaction hash as input.
3. Later, the user wishes for the data to also be stored on a different BC, triggering a migration process. This BC has a transaction size limit of 40 Bytes. With this size limit, the data will have to be split differently.

This scenario leads to a number of critical situations *(i)* splitting the string for a given size limit, *(ii)* reconstructing the entire string based on partial information and *(iii)* performing a migration onto a BC with a different transaction size limit.

For the sake of simplicity, in all three sub-scenarios a docker container [18] running Ganache [14] was used to simulate an Ethereum BC. The different transaction size limits of 28 respectively 40 Bytes do not reflect any limits imposed by Ethereum, but are superimposed by settings within Bifröst to artificially create different limits that will actually be exceeded with the example input.

The following subsections (5.4.1, 5.4.2, 5.4.3) are each dedicated to one of the three sub-scenarios enumerated above.

5.4.1 Evaluation of Splitting Feature for `store`

Let the string the user wants to store be "Lorem ipsum dolor sit amet, consectetur adipiscing elit volutpat.". No encryption is requested by the user and the target BC has a transaction size limit of 28 Bytes.

After a successful `store` operation, the three new entries have been added to Bifröst's database. Table 5.1 shows those new entries.

Table 5.1: Bifröst database entries after `store` process.

hash	blockchain_id	issued_at	salt	key	hash_next
0x79619f0c63228bb 4bb8bcc93266d0078 588785342bbff03e0 783e079bde34157	1	2021-03-21 14:37:47.198485			0xaeef172d4f4f2d1 10d8ca4fca56b858b 6f508eeeaab07618e b33df5be006b30a
0xaeef172d4f4f2d1 10d8ca4fca56b858b 6f508eeeaab07618e b33df5be006b30a	1	2021-03-21 14:37:47.205058			0xec222ff24808219 ade21f42faa2508a5 5345373dad9fcad4c f1ffa68e439f57d
0xec222ff24808219 ade21f42faa2508a5 5345373dad9fcad4c f1ffa68e439f57d	1	2021-03-21 14:37:47.205903			0x79619f0c63228bb 4bb8bcc93266d0078 588785342bbff03e0 783e079bde34157

The columns `salt` and `key` are empty, as no encryption was performed. Note how the `hash_next` entry for a given entry equals the `hash` of the subsequent transaction, with the last one's `hash_next` equaling the `hash` of the first transaction, resulting in a closed loop of "pointers".

The timestamps in `issued_at` relate to the point in time in which the transaction hash was returned to Bifröst's Ethereum adapter, hence also indicate the order in which they were performed. The small gaps between the timestamps is due to the simulated Ethereum BC running locally, hence processing transactions quickly.

5.4.2 Evaluation of Splitting Feature for `retrieve`

Retrieving the data can be initiated with a `retrieve` request by the user, in which the only input needed to identify the set of transactions, into which the data was split, is any single one of the transaction hashes produced in Section 5.4.1.

Assuming the user supplies *e.g.,* the second hash as argument, Bifröst will use the closed loop of pointers implemented via the `hash` and `hash_next` entries in the database to identify all the involved transactions and retrieve the data stored therein.

Table 5.2: Unordered retrieved data, based on second transaction hash as input.

hash	issued_at	data
0xaeef172d4f4f2d110d8ca4fc a56b858b6f508eeeaab07618eb 33df5be006b30a	2021-03-21 14:37:47.205058	consectetur adipiscing elit
0xec222ff24808219ade21f42f aa2508a55345373dad9fcad4cf 1ffa68e439f57d	2021-03-21 14:37:47.205903	volutpat.
0x79619f0c63228bb4bb8bcc93 266d0078588785342bbff03e07 83e079bde34157	2021-03-21 14:37:47.198485	Lorem ipsum dolor sit amet,

Using the data stored in the database which has been shown in Table 5.1, the retrieved data takes the form shown in Table 5.2. Notice that it is not yet in its correct order, but with the help of the timestamps, the order can easily be reinstated to produce the initially stored text of "Lorem ipsum dolor sit amet, consectetur adipiscing elit volutpat.".

5.4.3 Evaluation of Splitting Feature for `migrate`

The `migrate` action internally consists of a `retrieve` process followed by a `store` process. Thus it can be initiated by the user by supplying any of the three transaction hashes. The way the data is returned has been discussed in Section 5.4.2. After ordering the data is in its initial form.

The subsequent store action then happens again on the simulated Ethereum BC but with a different superimposed transaction size limit of 40 Bytes. Given the 65 Byte length of the initial text, only two transactions are required this time around. After a successful `migrate` action the database thus holds two new entries. They can be seen in Table 5.3.

Table 5.3: Bifröst database entries after `migrate` process.

hash	blockchain_id	issued_at	salt	key	hash_next
0x36841aa03bc8c6b badc4d1f4b30aa9cd 303fcf2c67233b6c0 635695084cd0328	1	2021-03-21 15:38:41.447688			0x3901c881c6df818 de1ad68043b14bac0 29957e15b7541d6e9 cbe58ac9cb59b2a
0x3901c881c6df818 de1ad68043b14bac0 29957e15b7541d6e9 cbe58ac9cb59b2a	1	2021-03-21 15:38:41.454191			0x36841aa03bc8c6b badc4d1f4b30aa9cd 303fcf2c67233b6c0 635695084cd0328

Should a `retrieve` action be requested for the newly migrated data, the data splits would read as “Lorem ipsum dolor sit amet, consectetur ” and “adipiscing elit volutpat.”, both adhering to the new size limit of 40 Bytes.

5.5 Secure Private Key Management

Regarding the topic of the private keys, the research conducted during this thesis has found a number of ways how such private keys might be securely stored. Approaches range from key protection servers [10] where a part of the private key is stored on a separate server where it is protected by a password, over biometric solutions [3] to smart-card based security using physically unclonable functions [65].

What all of these approaches have in common is that they rely on the active provision of a secret or unfalsifiable property (such as in the case of *e.g.,* biometrics based approaches). For the sake of simplicity, the term “secret” shall cover both secrets and unfalsifiable properties such as fingerprints for the rest of this discussion.

The issue with these approaches is that they rely on *(i)* the secret being kept separate from the secured key, meaning should an adversary get access to the secured key they would still need to somehow procure the secret to access the private key and *(ii)* the secret being provided actively.

This poses a problem in the case of Bifröst in its current form, where Bifröst itself would have to actively provide the secret to unlock its secured private keys, hence would have to store both the secret and the secured private keys, thus rendering any such approach to secure private key storage obsolete as an adversary with access to Bifröst has immediate access to the secret as well.

Attempting to secure the secret itself requires the application of yet another secure key storage approach where that secret also has to be kept on Bifröst, thus achieving no progress in making the system more secure.

Furthermore, Bifröst cannot rely on secrets being provided actively by other parties, which would be the users of Bifröst, as a single set of credentials is used to make transactions on behalf of all users, hence the secret would have to be universally known for all users, making it inherently non-secret.

Multisignature schemes [9], where a transaction needs to be signed by multiple parties, provide additional security by requiring a valid signature from either all or a pre-defined number of all signatories to allow for transactions being made from a given address. This is also not applicable to Bifröst’s current private key setup, as signatories would need to decide in an automated fashion whether or not to sign a given requested transaction.

However, there is no real way to evaluate whether or not a given transaction that is requested through Bifröst is legitimate, as users simply send their requests and have Bifröst use its own addresses to perform the transactions from, meaning that ultimately all requested transaction using any of Bifröst’s addresses are legitimate.

Thus an adversary with access to Bifröst automatically has access to all of Bifröst's addresses (Bifröst does have to store these locally, else it cannot facilitate any transactions), and thus will be able to get valid signatures from the required signatories.

There may potentially be a multisignature setup where the signatories decide whether or not to sign a transaction based on the type of transaction at hand. Should it be a transaction that simply stores data, it is automatically signed, should it be a transaction that aims to transfer funds to a different address, then it is automatically rejected.

This would work, as Bifröst is not intended for facilitating anything but interoperability in terms of arbitrary data storage. However, it still would not prevent an adversary from completely draining the funds on Bifröst's addresses by spamming data-storage transactions without being accountable for it.

This is due to the fact that even if Bifröst does some sort of accounting for performed transactions, since the addresses are known to an adversary with access to Bifröst, he could circumvent Bifröst entirely by sending requests to the signatories directly. Security measures such as Cross-Origin Ressource Sharing (CORS) policies on the side of the signatories could then be used to restrict where requests are accepted from [39].

The feasibility of such a setup, which *(i)* is compatible with all supported BCs, *(ii)* properly blocks requests from undesired locations and *(iii)* does not accidentally, through its complexity, introduce additional security risks, has not been verified in this thesis due to time-restrictions.

Existing Python libraries such as "pymusig" [72] or "multi-party-schnorr" [68] which both implement the "Schnorr signature scheme", or "multisig-hmac" [66] which does not declare the underlying signature scheme, do in the author's view not show the necessary degree of maturity or upkeep to be used for such security critical aspect of Bifröst, but may potentially serve as inspiration for a bespoke multisignature solution, specifically designed for Bifröst, should this be the avenue pursued in future work.

A different solution would be the introduction of user-specific credentials, meaning Bifröst would require users to register with a password and for each user creates a set of credentials for each supported BC in an automated fashion. The respective private keys would then be encrypted with the user's password, which he then would have to supply for each transaction such that the private keys can be used.

This would solve both the issue the active provision of the secret, as well as keeping the secret separate from the secured private keys as discussed earlier in this Section. However, it would require an extensive rework of Bifröst's architecture in terms of credential handling, as well as the addition for user registration, user account handling (*e.g.,* compliance with General Data Protection Regulation (GDPR) [62]) and automated credentials generation, which put this solution outside of what is feasible within the timeframe of the thesis.

5.6 Discussion

To summarize, the extension of Bifröst has produced benefits in a number of areas, ranging from flexibility to security, whilst maintaining the original functionality and keeping the impact on the original architecture as minimal as possible.

Flexibility has been enhanced with the introduction of the string splitting feature, as users now have the capability of storing data split over multiple transactions, hence no longer being restricted by the respective BC's transaction data size limit. This allows for a potentially larger set of use-cases for which Bifröst is now usable.

Standardization efforts have been undertaken to make interactions with Bifröst more structured. Although ultimately, there was no existing standard to directly rely on, the proposed format has been set up to not only facilitate interactions with Bifröst as it is now, but also with similar APIs and potential future versions of Bifröst.

For example, with the format's support for specifying the type and fields of the data that are sent with the requests, Bifröst could set up custom data types which it expects to be used in requests. Received requests could then have their data's type specification and fields compared against one or multiple data types that are allowed for a given type of request. This would allow for *(i)* easy validation of the completeness of the received data as well as for *(ii)* the distinction of potentially different use-cases based on the received data types. Ultimately, this standard allows any API to use customisable data types as inputs, whilst still remaining within the same standard.

In terms of security there are two aspects that Bifröst has to deal with, the security of the private keys, as well as the security of the user's data. Unfortunately there has not been any tangible progress regarding the former during this thesis, besides reaffirming the conclusions drawn by Timo Hegnauer in his Master thesis when implementing the Bifröst prototye, by providing a more thorough examination of available options. Both the introduction of a multisignature scheme or the use of user-specific credentials tied to user-accounts, require changes, that are not feasible within the available timeframe.

However, the security concerning user's data has been improved significantly through the introduction of the data encryption feature, which allows users to have their data encrypted with a password of their choice on a per-request basis. The performance impact of the encryption, though noticable, in the author's opinion is still within reason. Importantly, it is more or less constant regardless of both the length of the user's password as well as the length of the data that is to be encrypted. As such, the performance impact of encryption is expected be constant across all use-cases and hence, does not introduce an unwanted specialization of Bifröst towards certain scenarios.

The fact that encryption leads to increased data sizes must be mentioned here as a drawback, as it prevents even minuscule data to be stored with a single transaction for a large number of supported BCs, once encrypted. For this, a solution in the form of format-preserving encryption has been explored both in theory and implementation and is easily feasible but from the author's point of view not currently recommended due to the out-of-date library which implements it.

Until a more established library for format-preserving encryption is available, the size-impact of data encryption significantly hampers use-cases where very small amounts of private data are to be stored. Here, an informed choice of the BC on which the data is stored would be beneficial, since BCs such as *e.g.,* Ethereum have sufficiently large transaction data size limits as to not require splitting for even data in the lower two-digit Kilobyte range, whereas Stellar will require a minimum of four transaction for even the most minuscule amount of encrypted data. As such, Bifröst could include use-case specific BC recommendations in its documentation for when it sees its full release.

Finally, the robustness of Bifröst has been improved by introducing a form of generic error handling and via the introduction of the redundancy feature, which creates an opt-in backup of data which users store via Bifröst.

Chapter 6

Summary and Future Work

Ever since the introduction of the blockchain (BC) technology in late 2008, when the Bitcoin white paper was published under the pseudonym "Satoshi Nakamoto", both the number of BC implementations, as well as the scopes of those implementations keep growing. At the time of writing this summary, coinmarketcap [13] lists close to 9'000 cryptocurrencies, which nicely illustrates the growth the BC technology has experienced.

With both the number of BC implementations and the number of scopes and visions for what might be achieved with the BC technology increasing, the result ultimately is a large amount of fundamentally incompatible BC platforms that cannot natively exchange their respective currencies or the data they store [48]. Thus, the need for interoperability solutions is strong.

Bifröst is one such interoperability solution, which takes a novel approach to facilitating data exchanges between different BC platforms by implementing a notary scheme and presenting an API which acts as a layer of absctraction between its users and the complex implementations of the underlying BCs which it connects to [48]. Thus, the users are presented with an easy to use, flexible and modular interface that is common for all the supported BCs [48].

The prototype implementation had potential for improvements in a number of areas by adding features or updating the existing code. During the duration of this thesis, the flexibility of Bifröst was increased by introducing data splitting which allows user to store data that exceeds the given BCs transaction data size limit by dividing it up onto multiple transactions and implementing the necessary tracking mechanism to reassemble it once needed.

Furthermore, the security of user data was strengthened via the introduction of an encryption scheme, that enables users to have their data encrypted on a per-request basis. This is critical, due to the immutable nature of BCs, where any private information, once stored on a BC, can neither be changed nor removed. Having the option to encrypt the data before storing it, thus, opens up new use-cases for which Bifröst can be used.

Performance evaluations have shown that the impact of encryption data, whilst noticeable, remains constant, regardless of the length of the password chosen by the user and the size

of the data which needs to be encrypted. Whilst the way encryption impacts performance can be seen as positive, the way it affects data size must be seen as a drawback. Data size increases significantly, in particular for very small data, which on most BCs can no longer be stored with a single transaction, once encrypted.

An exploratory alternative implementation using so called "format-preserving encryption" has been tested for feasibility and has shown that that type of encryption can easily be incorporated into the existing encryption feature, whilst not affecting data size at all, hence solving this particular issue. However, the author cautions against the use of this encryption until a more mature library implementation is present.

Additionally, two features, namely generic error handling and redundancy were implemented to increase the robustness of Bifröst in case of errors and data loss, which can also be seen as an improvement to ease of use.

Two more research-intensive tasks have been performed alongside that, with the first exploring standardized interaction formats for interoperability APIs such as Bifröst. The research done has found numerous standardization efforts, but ultimately none were directly applicable to a notary scheme, thus the findings simply served as inspiration for the creation of a JSON scheme that can be used to facilitate interactions with Bifröst and that is generic enough to be applicable to similar interoperability solutions. The resulting JSON scheme introduces a generic way to transmit data of arbitrary types, thus allowing Bifröst to potentially further enhance its flexibility by allowing for the storage of data other than generic strings.

The second research topic was the investigation of secure private key management schemes which are applicable to Bifröst' s non-user specific credentials setup. Ultimately, that investigation ended up not providing any immediately actionable solutions, but rather resulted in two conceptual approaches, one of which may be applied to the current credentials setup, with the other one requiring a complete redesign of how private keys are managed.

THus, both approaches pose important options for future work with regard to the Bifröst's private key management. Furthermore, an actual incorporation of the JSON format into a future version of Bifröst, that is deployed as a server with its API open to receiving HTTP-requests, also offers opportunities for future work.

Bibliography

[1] Andreas M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, Inc., 2017.

[2] ARK Ecosystem SCIC. ARK Ecosystem Whitepaper, 2019. `https://ark.io/Whitepaper.pdf`, Last visit January 12, 2021.

[3] Mehmet Aydar, Salih Cemil Cetin, Serkan Ayvaz, and Betul Aygun. Private Key Encryption and Recovery in Blockchain. *arXiv preprint arXiv:1907.04156*, 2, 2019.

[4] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling Blockchain Innovations with Pegged Sidechains. 2014.

[5] Imran Bashir. *Mastering Blockchain*. Packt Publishing Ltd, 2017.

[6] Rafael Belchior, André Vasconcelos, Sérgio Guerreiro, and Miguel Correia. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. *arXiv preprint arXiv:2005.14282*, 2, 2020.

[7] Mihir Bellare, Phillip Rogaway, and Terence Spies. The FFX Mode of Operation for Format-Preserving Encryption. *NIST submission*, 20:19, 2010.

[8] Marianna Belotti, Nikola Božić, Guy Pujolle, and Stefano Secci. A Vademecum on Blockchain Technologies: When, Which, and How. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[9] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-Signatures for Smaller Blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.

[10] Ernie Brickell and Matthew D. Wood. Secure storage of private keys, September 27 2005. US Patent 6,950,523.

[11] Vitalik Buterin. Slasher: A Punitive Proof-of-stake Algorithm, January 2014. `https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/`, Last visit February 2, 2021.

[12] Vitalik Buterin. Chain Interoperability, September 2016. `https://allquantor.at/blockchainbib/pdf/vitalik2016chain.pdf`, Last visit November 2, 2020.

[13] CoinMarketCap. Cryptocurrency Prices, Charts and Market Capitalizations, 2009. https://coinmarketcap.com, Last visit March 29, 2021.

[14] ConsenSys Software Inc. Truffle Suite - Ganache, 2021. https://www.trufflesuite.com/ganache, Last visit March 16, 2021.

[15] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, and Nachiappan. Blockchain Technology: Beyond Bitcoin. *Applied Innovation Review*, 2(6-10):71, June 2016.

[16] Advait Deshpande, Katherine Stewart, Louise Lepetit, and Salil Gunashekar. Distributed Ledger Technologies/Blockchain: Challenges, Opportunities and the Prospects for Standards. *Overview Report The British Standards Institution (BSI)*, 40:40, 2017.

[17] Digital Asset Holdings, LLC. Daml Documentation, 2021. https://docs.daml.com/index.html, Last visit February 23, 2021.

[18] Docker. Docker Homepage, 2021. https://www.docker.com/, Last visit March 16, 2021.

[19] Johannes Dollinger. Python Library "pyffx", 2019. https://pypi.org/project/pyffx/, Last visit March 16, 2021.

[20] Ethereum. BTC Relay Documentation, 2016. https://btc-relay.readthedocs.io/en/latest/, Last visit January 12, 2021.

[21] Praveen Gauravaram. Security Analysis of Salt || Password Hashes. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 25–30. IEEE, November 2012.

[22] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16, Vienna, Austria, 2016. Association for Computing Machinery.

[23] Vincent Gramoli and Mark Staples. Blockchain Standard: Can We Reach Consensus? *IEEE Communications Standards Magazine*, 2(3):16–21, 2018.

[24] Halon Security AB. Halon Scripting Language, 2020. https://docs.halon.io/hsl/about.html, Last visit February 23, 2021.

[25] Thomas Hardjono, Alexander Lipton, and Alex Pentland. Toward an Interoperability Architecture for Blockchain Autonomous Systems. *IEEE Transactions on Engineering Management*, 67(4):1298–1309, 2019.

[26] Herdius GmbH. Herdius Next Generation Decentralized Blockchain Financial Infrastructure, 2017. https://herdius.com/whitepaper/Herdius_Whitepaper_1.1.pdf, Last visit January 12, 2021.

[27] David Hyland-Wood and Shahan Khatchadourian. A Future History of International Blockchain Standards. *The Journal of the British Blockchain Association*, 1(1):3724, 2018.

[28] Interledger Project. Interledger Architecture, 2021. `https://interledger.org/rfcs/0001-interledger-architecture/`, Last visit January 12, 2021.

[29] Interledger Project. Interledger Protocol V4 (ILPv4), 2021. `https://interledger.org/rfcs/0027-interledger-protocol-4/`, Last visit February 23, 2021.

[30] Hai Jin, Xiaohai Dai, and Jiang Xiao. Towards a Novel Architecture for Enabling Interoperability Amongst Multiple Blockchains. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1203–1211. IEEE, 2018.

[31] Will Kenton. Hyperledger Fabric, 2020. `https://www.investopedia.com/terms/h/hyperledger-fabric.asp`, Last visit February 2, 2021.

[32] KomodoPlatform.com. Komodo Whitepaper, 2018. `https://whitepaper.io/document/34/komodo-whitepaper`, Last visit March 9, 2021.

[33] KomodoPlatform.com. How To Become a Liquidity Provider on AtomicDEX, 2021. `https://developers.komodoplatform.com/basic-docs/atomicdex/atomicdex-tutorials/how-to-become-a-liquidity-provider.html#requirements`, Last visit January 11, 2021.

[34] Jae Kwon and Ethan Buchman. Cosmos Whitepaper, 2021. `https://cosmos.network/resources/whitepaper`, Last visit January 12, 2021.

[35] Sergio Demian Lerner. RSK Bitcoin Powered Smart Contracts White Paper Overview, 2019. `https://www.rsk.co/Whitepapers/RSK-White-Paper-Updated.pdf`, Last visit January 12, 2021.

[36] Noam Levenson. Why Ark Deserves Your Attention, 2017. `https://medium.com/hackernoon/why-ark-deserves-your-attention-c57acd51846a`, Last visit January 12, 2021.

[37] Iuon-Chang Lin and Tzu-Chun Liao. A Survey of Blockchain Security Issues and Challenges. *IJ Network Security*, 19(5):653–659, 2017.

[38] Bertrand Meyer. *Object-Oriented Software Construction*, volume 2. Prentice Hall PTR, 1997.

[39] Mozilla. Cross-Origin Resource Sharing (CORS), 2021. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`, Last visit March 30, 2021.

[40] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009. `https://bitcoin.org/bitcoin.pdf`, Last visit November 24, 2020.

[41] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[42] Python Software Foundation. Built-in Types: Text Sequence Type, 2020. `https://docs.python.org/3.7/library/stdtypes.html#text-sequence-type-str`, Last visit February 28, 2021.

[43] Python Software Foundation. Copy - Shallow and Deep Copy Operations, 2021. `https://docs.python.org/3/library/copy.html`, Last visit February 16, 2021.

[44] Kaihua Qin and Arthur Gervais. An Overview of Blockchain Scalability, Interoperability and Sustainability. *Hochschule Luzern Imperial College London Liquidity Network*, 2018.

[45] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 99–100, Linköping, Sweden, 2001. IEEE.

[46] Sara Rouhani and Ralph Deters. Performance Analysis of Ethereum Transactions in Private Blockchain. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 70–74. IEEE, 2017.

[47] Eder John Scheid. Bifrost Gitlab Repository, 2019. `https://gitlab.ifi.uzh.ch/scheid/bifrost`, Last visit January 12, 2021.

[48] Eder John Scheid, Timo Hegnauer, Bruno Rodrigues, and Burkhard Stiller. Bifröst: a Modular Blockchain Interoperability API. In *IEEE Conference on Local Computer Networks (LCN 2019)*, pages 332–339, Osnabrück, Germany, October 2019. IEEE.

[49] Eder John Scheid, Daniel Lakic, Bruno B. Rodrigues, and Burkhard Stiller. PleBeuS: a Policy-based Blockchain Selection Framework. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, Budapest, Hungary, 2020. IEEE.

[50] Stefan Schulte, Marten Sigwart, Philipp Frauenthaler, and Michael Borkowski. Towards Blockchain Interoperability. In *International Conference on Business Process Management*, pages 3–10. Springer, 2019.

[51] Gustavus J. Simmons. Symmetric and Asymmetric Encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, December 1979.

[52] Matthew Spoke and Nuco Engineering Team. Aion White Paper, 2017. `https://whitepaper.io/document/31/aion-whitepaper`, Last visit January 12, 2021.

[53] Manu Sporny and Dave Longley. The Web Ledger Protocol, 2019. `https://w3c.github.io/web-ledger/`, Last visit February 23, 2021.

[54] Manu Sporny and Dave Longley. The Web Ledger Protocol - Examples, 2019. `https://w3c.github.io/web-ledger/#storing`, Last visit February 23, 2021.

[55] Stack Exchange Inc. Accepted Answer to Stackoverflow: "Python : Get size of string in bytes", 2015. `https://stackoverflow.com/a/30686735`, Last visit March 15, 2021.

[56] Stack Exchange Inc. Cryptography Stack Exchange: "Why is 'semantically secure' important for cryptosystems?", 2015. https://crypto.stackexchange.com/questions/30130/why-is-semantically-secure-important-for-cryptosystems/30138#30138, Last visit March 16, 2021.

[57] Stack Exchange Inc. Python: Get Size of String in Bytes, 2015. https://stackoverflow.com/questions/30686701/python-get-size-of-string-in-bytes, Last visit February 28, 2021.

[58] Stack Exchange Inc. Default Message in Custom Exception - Python, 2019. https://stackoverflow.com/a/56967197, Last visit March 31, 2021.

[59] Kevin Stine and Quynh Dang. Encryption Basics. *Journal of AHIMA*, 82(5):44–46, 2011.

[60] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 2015.

[61] The Block. Ethereum Miners Are Increasing The Network's Gas Limit by 25%, 2020. https://www.theblockcrypto.com/linked/69053/ethereum-miners-vote-for-25-gas-limit-increase, Last visit February 28, 2021.

[62] The European Parliament and the Council of the European Union. General Data Protection Regulation, 2016. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679, Last visit March 30, 2021.

[63] Stefan Thomas and Evan Schwartz. Interledger White Paper, 2014. https://interledger.org/interledger.pdf, Last visit March 9, 2021.

[64] Treat, David and Giordano, Giuseppe and Schiatti, Luca and Borne-Pons, Hugo. Connecting Ecosystems: Blockchain Integration, 2018. https://www.accenture.com/_acnmedia/PDF-88/Accenture-20180514-Blockchain-Interoperability-POV.pdf, Last visit March 2, 2021.

[65] Pim Tuyls, Boris Škoric, and Tom Kevenaar. *Security with Noisy Data: on Private Biometrics, Secure Key Storage and Anti-Counterfeiting*. Springer Science & Business Media, 2007.

[66] User "AmalieDue" on GitHub.com. Python Library "multisig-hmac" Repository, 2020. https://github.com/AmalieDue/py-multisig-hmac, Last visit March 30, 2021.

[67] User "MintDice" on Medium.com. A Guide to Ark Cryptocurrency, 2019. https://medium.com/bitcoin-news-today-gambling-news/a-guide-to-ark-cryptocurrency-505dc8bdc7b6, Last visit January 12, 2021.

[68] User "namuyan" on GitHub.com. Multi Party Schnorr Signatures (python extension), 2020. https://github.com/namuyan/multi-party-schnorr, Last visit March 30, 2021.

[69] User "pyca" on GitHub.com. Cryptography Library Documentation, 2021. https://cryptography.io/en/latest/, Last visit February 16, 2021.

[70] User "pyca" on GitHub.com. Cryptography Library Documentation - Example for Encryption with Password, 2021. https://cryptography.io/en/latest/fernet.html#using-passwords-with-fernet, Last visit February 16, 2021.

[71] User "pyca" on GitHub.com. Python Cryptography Library, 2021. https://pypi.org/project/cryptography/, Last visit February 16, 2021.

[72] User "rage-proof" on PyPI.org. MuSig multisignatures for Python, 2020. https://pypi.org/project/pymusig/, Last visit March 30, 2021.

[73] Various Community Contributors. ERC | Ethereum Improvement Proposals, 2021. https://eips.ethereum.org/erc, Last visit February 23, 2021.

[74] Dejan Vujičić, Dijana Jagodić, and Siniša Ranđić. Blockchain Technology, Bitcoin, and Ethereum: A Brief Overview. In *2018 17th international symposium infoteh-jahorina (infoteh)*, pages 1–6, East Sarajevo, Bosnia and Herzegovina, 2018. IEEE.

[75] Wanchain Foundation LTD. Building Super Financial Markets for the New Digital Economy, 2017. https://wanchain.org/files/Wanchain-Whitepaper-EN-version.pdf, Last visit January 11, 2021.

[76] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. *IEEE Access*, 7:22328–22370, 2019.

[77] Web 3 Foundation. A Scalable, Interoperable & Secure Network Protocol for the Next Web, 2021. https://polkadot.network/technology/, Last visit January 12, 2021.

[78] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On Availability for Blockchain-Based Systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–73, Hong Kong, China, 2017. IEEE.

[79] Gavin Wood. Polkadot: Vision for a Heterogeneous Multi-Chain Framework, 2016. https://polkadot.network/PolkaDotPaper.pdf, Last visit March 10, 2021.

[80] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain Technology Overview. *arXiv preprint arXiv:1906.11078*, 1, 2019.

[81] Muneer Bani Yassein, Shadi Aljawarneh, Ethar Qawasmeh, Wail Mardini, and Yaser Khamayseh. Comprehensive Study of Symmetric Key and Asymmetric Key Encryption Algorithms. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–7, Antalya, Turkey, 2017. IEEE.

Abbreviations

API	Application Programming Interface
BC	Blockchain
CORS	Cross-Origin Resource Sharing
GDPR	General Data Protection Regulation
KDF	Key Derivation Function
P2P	Peer-to-Peer
PoW	Proof-of-Work
PoS	Proof-of-Stake
RPC	Remote Procedure Call
SC	Smart Contract
TX	Transaction

List of Figures

List of Tables

Appendix A

Installation Guidelines

The installation guidelines in their entirety as seen here can also be found in the `README.md` of the Bifröst GitLab project [47].

A.1 Setup

Python 3.7.1 (also tested with 3.8.3) was used for this project. It was tested on MacOS.

Previous versions of the project were also tested on Ubuntu 18, hence the Linux specific installation instructions. These have not been re-tested but should still be valid.

A.1.1 Install Docker

(Linux) Follow this or this manual and then do sudo apt-get install docker-compose

(Mac) Install from https://docs.docker.com/docker-for-mac/install/

A.1.2 Setup virtual environment (venv)

```
# Linux Only
# Install venv:
sudo apt-get install python3-venv

# all Platforms
# Create environment:
python3 -m venv myNewVenv
# Activate environment:
source myNewVenv/bin/activate
# The python version of the environment will be the one with which the
    environment is created.
```

Don't forget to add the name of your new environment in the `.gitignore` under the `# Environments` header to prevent installed packages from being tracked by git.

A.1.3 Install dependencies

Preparation

First, upgrade pip:

```
pip install --upgrade pip
# (Linux only)
sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
```

Upgrade pip on <3.6:

```
# (Mac only) Use this command if upgrading upgrading pip fails due to SSL
    cert error:
curl https://bootstrap.pypa.io/get-pip.py | python
```

Important Notice

Note that there are two files that carry information about the dependencies and their respective versions, `requirements.txt` and `requirements_unresolved.txt`.

The normal `requirements.txt` has specific version information for each dependency, as resolved by pip. The unresolved requirements file has no specific version set for packages that caused issues when performing the initial setup for the development.

Thus, use `requirements_unresolved.txt` in case you get incompatible sub-dependency versions in the next step with the normal `requirements.txt`. This will let pip resolve the versioning on its own.

Install/Export Dependencies

```
# Import/Install depenencies:
myNewVenv/bin/pip install -r requirements.txt
# Export:
myNewVenv/bin/pip freeze > requirements.txt
```

(In case of errors, read the Important Notice under the previous header.)

A.1.4 Database Setup

```
# (Mac Only) Install sqlite:
brew install sqlite3

# Then import and setup the DB:
import db.database
db.database.setup()
```

Calling the setup function of the database module will:

- drop `credentials` and `transactions` tables if they already exist
- create tables for storing credentials and transactions
- seed the `credentials` table with credentials
- seed the `transactions` table with input transactions
- Seed values are read from the config module.

A.1.5 Blockchain Setup

See descriptions in SETUP.md on Bifröst's GitLab [47] for instruction to setup the local nodes. This goes beyond the basic setup, hence is not shown here.

Appendix B

Contents of the CD

The data for this thesis has not been submitted in form of a CD, but rather as a .zip file in accordance with what was agreed upon during supervisor meetings. The content consists of the following items:

- `thesis.zip`, a .zip file containing the LaTeX source code for the thesis.
- `P-Kiechl-BA-Thesis-Final.pdf`, a .pdf export of the final version of the thesis.
- `bifrost-v2.zip`, a .zip file holding the entire source-code for the extended Bifröst implementation. Local-only files excluded from version control via .gitignore configurations are included here.
- `midterm.pptx`, the slides for the midterm presentation held on January 12, 2021

Notably, the slides of the final presentation are not included. As per agreement with the supervisor, these are to be supplied at a later date, once the presentation has been held. The presentation is scheduled for April 16, 2021.