University of Zurich[UZH]

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

Increasing Privacy in Smart Contracts: Exploring Encryption Mechanisms

Raphael Imfeld
Zurich, Switzerland
Student ID: 18-702-696

Supervisor: Eder J. Scheid, Christian Killer
Date of Submission: April 1, 2022

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

Nach der Einführung des Konzepts der Smart Contracts (SC) im Jahre 1994, dauerte es ein weiteres Jahrzehnt, bis ein Anwendungsszenario gefunden werden konnte. Der Fokus auf Transaktionen der Blockchain-Technologie stellte sich als eine geeignete Grundlage zur Implementation von automatisierten, selbst-ausführenden Contracts heraus. Populäre Blockchains, wie Ethereum, wiesen eine enge Integration von SC mit den Kernfunktionen des Systems durch eine eigens dafür entwickelte Programmiersprache Solidity auf. Da herkömmliche, physische Verträge, spezifische Sicherheitsmerkmale bezüglich Geheimhaltung aufweisen, werden ähnliche Eigenschaften von ihren digitalen Äquivalenten erwartet. Die Herausforderung zur Einbindung dieser Eigenschaften akzentuiert sich durch das Merkmal einer Blockchain "trustless" zu sein, weshalb kein Kommunikationskanal zwischen den Vertragsparteien existiert. Um dies zu lösen, werden kryptographische Systeme genutzt, damit die Sicherheit durch Verschlüsselung von on-chain-Daten gewährleistet wird. Nur authorisierten Vertragsparteien ist es dadurch möglich Datenänderungen und Entschlüsselungen vorzunehmen. Alternativ kann durch eine off-chain-Variante der Ort zur Speicherung und Prozessierung für eine beliebige Anzahl von Daten auf eine Drittpartei ausgelagert werden. Diese verschiedenen Verschlüsselungsmethoden wurden in der vorliegenden Arbeit durch eine Implementation eines einfachen Transaktionsszenarios mit verschiedenen Datentypen untersucht, was Limitationen der Verwendung von on- und off-chain-Systemen aufzeigte. Im Anschluss wurde die Performance unter Anwendung verschiedener Verschlüsselungsmethoden bezüglich des benötigten Speicherplatzes sowie des Gas-Verbrauchs und der Laufzeit gemessen. Abschliessend wurde ein Vergleich zu den einzelnen Methoden und ihren Schwierigkeiten bezüglich eines on-chain-Systems erstellt, um so Handlungsfelder für weitere Forschungen zu eruieren.

Die Auswertung zeigte eine positive Korrelation zwischen den Verschlüsselungsmethoden und den drei genannten Messungsparameter. Bei Nutzung von unverschlüsselten Werten wurde der geringste Speicherbedarf, der Verbrauch an Gas und die kürzeste Laufzeit gemessen, wohingegen die homomorphische Verschlüsselungsmethode am anderen Ende der Skala lag.

After the introduction of the concept of Smart Contracts (SC) in 1994, it took another decade until a use case was found. The blockchain's focus on transactions appeared to be a perfect ground to implement the concept of automated, self-executing contracts. Popular blockchains such as Ethereum tied the integration of SC closely to their core functionalities, using the programming language Solidity specifically introduced for this purpose. Since physical contracts know distinct security properties due to privacy requirements, the digital equivalents are expected to fulfill the same. However, the transfer of

such properties are challenging as some blockchains are trustless systems and therefore no channel of communication between the two contracting parties is expected. In order to resolve this challenge, cryptographic mechanisms were introduced to ensure privacy by either encrypting the values on-chain and allow them to be read and manipulated by authorized contracting parties or using an off-chain approach, which outsources the storage and manipulation of sensitive data to a Trusted Third Party. Different encryption approaches were explored by implementing a simple transaction scenario using a SC with different types of data, showing limitations of each approach when using a on- or off-chain solution. Furthermore, performance of the encryption approaches were investigated in order to determine aspects, such as the contract size, the Gas used during the process and runtime. Finally, a comparison of all approaches was done, showing the difficulties of on-chain approaches for the chosen scenario and proposing some adjustments for further research to simplify the implementation.

The evaluation showed a positive correlation between the complexity of the encryption mechanism and the three parameters mentioned, since the unencrypted approach used the least amount of memory or Gas and was the fastest, while the homomorphic approach was located at the other end of the scale.

Acknowledgments

I would like to give my warmest thanks to the supervisors of this thesis, Eder J. Scheid and Christian Killer. In particular Eder J. Scheid carried me through the stages of designing and writing this thesis with his guidance. The fruitful discussions in the meetings motivated me along the way until the final presentation.

Additionally, I want to express my deepest thanks to Yelena Rubli and Ramon Solo de Zaldivar, for the unconditional and invaluable support in challenging times throughout my studies.

Finally, I would also extend my gratitude to Prof. Dr. Burkhard Stiller and the Communication Systems Group (CSG) at the University of Zurich for giving me the opportunity to write this thesis at their research group.

Contents

Chapter 1

Introduction

With the evolution of the blockchain, more applications from the industry started to make use of such a novel technology [6]. Transactions between different companies are performed using contracts or protocols, which creates the framework of possible actions between law and the private sector. Since transactions could be (partly) conditional, the prerequisites have to be checked prior to the contract execution to guarantee a valid outcome of the agreed contractual parts. However, continuously verifying if these conditions are met, is a time and cost-intensive task and prone to errors [39].

In this sense, blockchain-based Smart Contracts (SC) were proposed to solve the problem of performing such checks of the defined rights and obligations with an underlying digital protocol and relying on algorithms defined by the involved parties, which can exchange messages amongst each other [5]. As the contractual procedure is formalized programmatically in an implemented SC, the human interaction is kept to a minimum [39]. Using blockchain (*e.g.*, Ethereum [29]), the data of SCs can be stored, replicated, and updated by transactions sent by the involved parties. Further, due to the distributed aspect of blockchain without the need of a Trusted Third Party (TTP), reduction of execution times and required workforce (*e.g.*, employees) is achieved [43, 18].

Information on public blockchains is accessible by any interested party; thus, the same property applies to the content in SCs. Since the contracts might contain confidential information, the content of the contract or even its existence, exposing the relationship between the contractual parties must be kept secret. For example, one can assume an acquisition transaction between two companies, where competing institutions must not be able to have any possibility to gain insights regarding the acquisition conditions (*e.g.*, price). Therefore, the content of contracts should be only visible to the involved parties [43]. Encrypting SCs on the blockchain allows maintaining crucial aspects of conventional contracts by hiding sensitive information and enhancing trust, while still benefiting from an automated execution. In the early stages of before mentioned acquisition, even the fact of transactions being carried out between the involved parties could denote a breach of privacy since the competition will have an early indication of what will happen.

Fortunately, several encryption mechanisms can be used to preserve privacy of SCs. Zero-Knowledge Proofs (ZKP) allow storing ciphertext on-chain, while still letting the miners

verify the data, without having to expose any plain-text. Being a cryptographic primitive, ZKPs can be implemented in different architectures, such as a new SC language, which facilitates creation of privacy-preserving SCs, storing data encrypted as ciphertext on-chain together with a proof, such that the content of the SC can be validated [37]. Other approaches use Trusted Execution Environments (TEEs) to perform calculations off-chain and proofing such by introducing new authorities, which confirm the correct procedure [10]. Instead of performing calculations off-chain, homomorphic encryption allows executing them on-chain, since functions are able to compute ciphertext without the need of decrypting it into plain-text [36].

Since Ethereum allows SCs to be implemented in a Turing-complete language (*i.e.*, Solidity), it is one of the most referenced blockchains for creating a proof-of-concept in current research about encrypting SC data. However, there are solutions being developed which are system-independent [21] or requiring new properties. Therefore, implementing their own types of blockchain platforms, *e.g.*, [8, 31].

1.1 Description of Work

Each of the considered works showcases a different approach of using one or a combination of the before mentioned encryption schemes. This work analyzes different approaches (*e.g.*, symmetric encryption and homomorphic encryption), by comparing and applying them on a simple use-case of a transaction between two parties. Then, it assesses the applicability of different solutions, investigating the simplicity to implement the proposed solution. Additionally, the usage of resources is compared in-between the different solutions, in order to show the performance of each approach on the same hardware. Since some approaches use overlapping techniques, certain properties are discussed in order to identify drivers for specific findings.

1.2 Thesis Outline

In Chapter 2, the theoretical background of this thesis is discussed, explaining the used technologies as well as the encryption mechanisms. Chapter 3 summarizes state-of-the-art research on privacy-preserving SCs, comparing and discussing the findings of a selection of works. The design of scenario and different approaches is explained in Chapter 4. Further, in this chapter the implementation of the scenario for each approach is compared. The results are presented, evaluated, and discussed in Chapter 5, while the summary and conclusions are presented in Chapter 6.

Chapter 2

Background

In the course of this chapter, the theoretical background for the thesis is provided. Section 2.1 introduces the blockchain, emphasizing on the most known applications of the technology such as Bitcoin and Ethereum. The concept of Smart Contracts is described in Section 2.2.

2.1 Blockchain

A public ledger tracking all occurred events in historical order is the core of the blockchain [38]. Such transactions are collected together with metadata in so-called *blocks*. They are created by *miners*, which are peers in the network that act as a participant by validating and executing transactions. Each one of these receives and stores a copy of the blockchain with the Genesis-block at the beginning of the chain, containing the very first transactions in the network as seen in Figure 2.1.

Following the concept of a linked list, the blocks are chained together via a pointer connecting a previous element to the next. A single block consists of the hash value of the previous block, a timestamp and a nonce (number only used once).

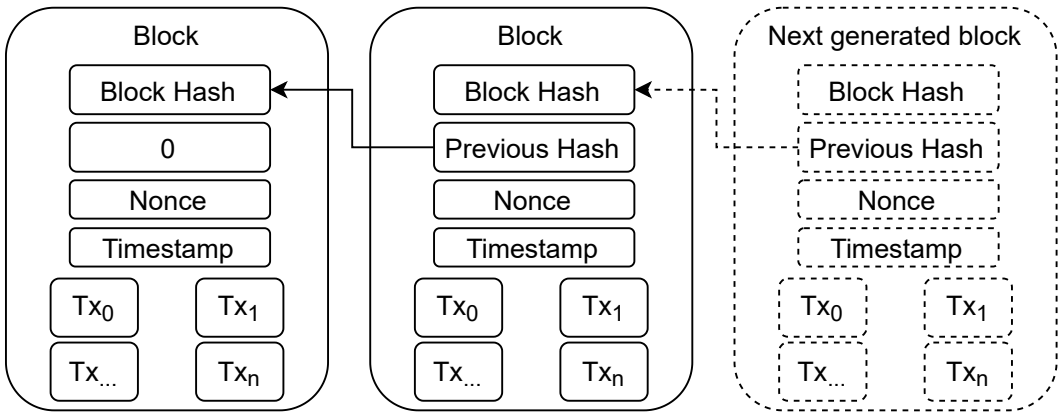

Figure 2.1: Principle of a blockchain

Altering data on the chain can by definition only be done by achieving a consensus within the network through announcing changes publicly. However, due to the decentralized system there needs to be a well-defined process to determine a common order of accepting incoming transactions in order to avoid double-bookings [30]. To resolve the challenge of using a distributed timestamp server and whose block will be used next, a Proof-of-Work (PoW) principle has been implemented [30]. Each miner tries to correctly hash the randomly generated nonce of each block, whereas finding the specific hash determines the mathematical difficulty. Incentivized by receiving a reward for having the right to generate the next block, the solving of this mathematical problem is done in the fastest possible time by the network of miners. Immutability is given by using the contained data as input for the hash function, which will only return the correct hash value, if the input remains the same.

These attributes of the system form the basis of trust in the system in contrary of commonly known centralized systems, which afford trust between the transaction partner and a possible third-party intermediary (*i.e.*, a bank) [38]. Even though the first blockchain proposal (*i.e.*, Bitcoin) implements specific functions such as PoW, other mechanisms might be applied to provide this trust. For example, Proof-of-Stake (PoS) consensus relies on the willingness of each node to put coins at stake in order to secure the blockchain. Therefore, computational power is not the main driver anymore for a node to be chosen to validate a new transaction. Such approaches resolve the problem of the criticized power consumption, which is needed to run a PoW-based blockchain [35].

2.1.1 Types of Blockchain

Despite all these security measures, alternative implementations have evolved in the recent years such as consortium or fully private blockchains. To change from a fully centralized to a distributed, decentralized system might not be tempting for companies working with sensitive data. The two mentioned alternatives mitigate the risk of data security and visibility.

Consortium blockchains change the above explained process for achieving consensus of the *public blockchain*. Instead of leaving the control to agree on changes to all nodes, only a set of miners is chosen to validate new blocks. Additionally, data access can be restricted to the before mentioned selection or still be public. Even partial access to the chain can be granted through limiting number of queries or proofs. The degree of decentralization is decreased from full to partial [29].

Fully private blockchains restrict the permissions to write to the blockchain to only one party of the network in a common setting as commonly known in a centralized system. This implementation could be needed for internal applications such as database management for which a public data access is not necessary but proofs are of general interest [29].

Both alternatives can increase trust in the system since less parties are allowed to alter data. Additionally, it is still possible to have sovereignty over the data, due to the more restrictive right to write to the blockchain. This prevents the system from being taken over with a 51%-attack, in which a majority of miners overthrow the consensus mechanism.

With less nodes the calculation power of the nodes can be steered to achieve a lower transaction cost, due to higher calculation power and less nodes to verify new blocks. In contrast, trust in the maintenance of the system as well as data integrity is based on the counterparty only. In such a (semi-) centralized system, network effects are low, since the fees are set solely by intermediaries, because they are needed to ensure the correct servicing of the transaction. If both, the currency and the actual transferable commodity, is on the same blockchain, these fees can be cut by drastically removing the need of having an intermediary.

Not only the participating actors are part of creating trust in a system, but also the developers, since they have the knowledge to change it and know fundamental technical details. The application logic is still implemented by them, however, the processes run autonomously and the blocks are created everywhere based on the data, which is continuously created without a developer being able to interfere. Such intentional cutting of power protects user from the creators of the system, since it is simply impossible to alter data without showing it to everyone, even if it was a developer [9].

2.1.2 Bitcoin and Ethereum

The bitcoin whitepaper was published in 2008 titled "Bitcoin: A Peer-to-Peer Eletronic Cash System" by an alias of Satoshi Nakamoto [30]. Its goal was to create a decentralized digital cash system without the need of having intermediaries or other central institutions, regulating the flow of the currency. Through the increasing difficulty of the PoW algorithm, a new block is issued every 10 minutes on average, controlling the order of transactions, which is crucial in a decentralized system. By increasing the credibility of the system the popularity of Bitcoin grew, starting in 2009 and at times reaching a market value of $35 billion US dollars, based on the exchange rate [2]. Since the beginning the system is fully transparent, providing all underlying mathematical principles, source code and how the consensus process is working. Another important part of the system are wallets. They are used similar to bank accounts, storing the amount of currency while providing one or more addresses, such that it is possible to transfer from and to the account [30].

As with cash in a classic wallet, it is possible to store other digital currencies in the same wallet, *e.g.*, Ethereum the second-most capitalized currency after Bitcoin [12]. However, the intended purpose differs strongly to the one of Bitcoin, which is creating a decentralized, transparent payment system by using a digital currency. Ethereum, in contrast, is a platform for creating contracts and applications in a P2P network, acknowledged by all participants of the network similar to the mechanism for bitcoins. The currency of the system is called *ether*, used as a reward for the miners and developers for building and running distributed applications [29].

2.1.3 On- and Off-chain Systems

Maintaining such a system comes with a cost of validating transaction, checking if there is consensus about the validation and subsequently updating the blockchain on every node.

This process can create overhead since distributing the same data to all nodes is not the most efficient way [15]. Such actions lead to a limit of transactions being carried out per second, with Bitcoin having a lower limit compared to Ethereum. Furthermore, rewarding the miners translates to transaction costs, which will be paid as fees by the transaction parties. To mitigate these inefficiencies, approaches of moving the data and computation off the blockchain were found. Even though this sounds at first easy to execute, it breaks the properties being used to establish trust in the system.

To move data off the chain it is required to still fulfill the prerequisites of data immutability due to the public distribution when being put on-chain, preserving privacy even if data is off-chain and the ability to verify off-chain processed private data on the chain, without having to expose the data. Following these guidelines, a layer of flexibility gets implemented, due to the new gained possibility of deciding how data being used in the transaction is stored and how changes will be computed [17].

Hence, on-chain defines the designed process for altering data on the blockchain via the chosen degree of decentralization, whereas off-chain approaches are used to enhance these processes for certain use cases, leaving the users a higher degree of flexibility to save costs and time [15].

2.2 Smart Contracts (SC)

These improvements apply also for other use cases of the blockchain, in particular for such used to transfer any tradable goods on the Internet without using an intermediary. Smart Contracts (SC) allow an automated execution, based on the terms the contract parties agreed on without relying on a central authority to check if the conditions are met since this check is already implemented in the code of the SC [29]. Once signed off, even the creator is not able to alter the content or to prevent execution. Since the state transitions in the life cycle of a contract need to be validated on the blockchain, terms of the contract are revealed, which prevent feasibility for confidential contracts. Moreover, timing could be critical, which is not given by the system regarding the creation time of a new block, adding up with transaction fees, which increase the overall spending for a contract based on validation steps [11]. Overcoming these issues and being able to use SCs in traditional businesses could contribute to cut costs by decreasing the need of manual control and therefore human resources. Such needs are shifted to the area of software developers who set up the contracts for a certain party [39].

Setting up a contract differs from the used blockchain technology. When implementing one on Ethereum, the contract acts comparable to a usual account with an own balance and the ability to send and receive transactions through the blockchain. As already described in the previous paragraph, this account is not belonging to a specific user, they are just a simple program and run due to the contained code. Transactions can trigger certain parts of the code (functions), which are pre-defined. After invoking a function, the outcome is defined by the rules of the SC and can not be deleted since an execution is irreversible. Despite automating the checks of conditions, it is not possible to let SCs to send HTTP requests in order to check for third-party events, since it breaks the consensus

process of the blockchain [20]. A contract is able to access the own state, the transaction, which triggers functions contained in the contract and information about previous blocks. Staying in the context of the Ethereum Virtual Machine (EVM) does not break the before mentioned properties. The functions of the contract will produce the same outcome based on the state of the blockchain and the context of the transaction triggering the event [29].

2.2.1 Non Turing-Complete vs Turing-Complete

In the Computer Science area, programming languages and systems can be divided into two categories, *(i)* Non Turing-Complete and *(ii)* Turing Complete. If a computer language is able to implement any Turing machine or program, it can be said to be *Turing Complete*. If not, it is *Non Turing-Complete* [25]. Nevertheless, being *Turing-complete* does not imply that any program can be fully executed in a finite time. There is no possibility to assess if a program takes forever to run without waiting an infinite amount of time for an eventual successful termination of execution, which is called the *halting problem*.

If any program ran forever on the blockchain of Ethereum for example, the system would be unusable, since there is no concurrent execution. To work in the Ethereum environment Gas is needed, which is the unit to measure computational and storage resources to execute the desired action. This allows the EVM to halt an execution as soon as the provided amount of Gas is 'consumed'. Of course there is the possibility to provide more Gas until a certain limit, which can be raised on consensus, but eventually the "block Gas limit" will be reached and execution will be halted. Thus the EVM is not a full *Turing-complete* machine, since programs with insufficient Gas will not be executed until they reach an accepting state [3].

However, using Non Turing-complete SCs allow easier auditing, due to the lower complexity of code, since they do not support recursion or complex loops. This will also decrease the possibility of defects being implemented, since the code will be more straightforward to review. Executing simpler programs result in a better performance and prevent congestion, which is caused by Turing-complete SCs using a lot of storage [28].

2.2.2 Solidity

Despite these disadvantages, the most widely used language for SCs is Solidity a *Turing-complete* language. It was developed specifically for writing SCs, providing features to be used on a decentralized blockchain. Mostly linked with Ethereum, it evolved and is now platform independent, resulting in usage on other platforms as well. Since it is universally usable, the language comes with an own compiler called `solc`, which is used to convert the code to EVM bytecode. It also handles the Application Binary Interface (ABI) of Ethereum, encoding contract calls for the EVM by retrieving the relevant data from the transaction. This is how the correct handling of functions is translated to machine code by defining the acceptance of arguments as well as the correct output. Upon creation of

the ABI for a contract, a JSON array will be set. After deployment of the contract, every other application is able to access it by using the created JSON array [3].

This facilitates the interconnection with wallets for example, due to the information stored in the JSON, allowing correct invokes of functions of the contract, using the correct type of arguments. A coding example is seen in Figure 2.1 showing a simple SC with two functions allowing to deposit and withdraw a certain amount, which is defined in the transaction sent to the SC (`msg.value`). The function `require()` shows the above described access rights to data being stored on the blockchain, in this case retrieved through calling `balances[msg.sender]`.

```
1  pragma solidity >=0.4.22 <0.8.10;
2  contract EtherBank {
3      mapping(address => uint256) public balances;
4      function deposit() external payable{
5          require(balances[msg.sender] + msg.value >= balances[msg.sender
               ]);
6          balances[msg.sender] += msg.value;
7      }
8      function withdraw(uint256 amount) external{
9          require(amount <= balances[msg.sender]);
10         balances[msg.sender] -= amount;
11         msg.sender.transfer(amount);
12     }
13 }
```

Listing 2.1: Example of a SC written in Solidity [23]

Due to the before mentioned *Turing-completeness*, SC written in Solidity can be of any desired complexity. However, calculation and therefore Gas costs have to be considered as well, since each call of a function will increase such [42]. Consequently, if a quicker, hence less expensive, execution is crucial for the specific use case, the implementation needs to carefully use resources, in order to get to the targeted outcome in an acceptable time frame.

2.3 Encryption Mechanisms

This chapter emphasizes on mechanisms to encrypt data in order to secure data from being accessed or altered by an undesired third party. Current daily used encryption schemes are based on asymmetric and symmetric encryption, which is illustrated in Section 2.3.1. Proofing knowledge without having to reveal any details through so-called Zero Knowledge Proofing is explained in Section 2.3.2. Working on ciphertext without the need to decrypt it to plain text is part of Full Homomorphic Encryption in Section 2.3.3.

2.3.1 Asymmetric and Symmetric Encryption

Sharing data in the form of messages is tied to a secure channel, which sender and receiver could trust. In order to achieve this, encryption is used to make the data unreadable for

third parties. Even if the channel is compromised or left insecure intentionally, the message can not be used since it is encrypted. *Symmetric encryption* and *Asymmetric encryption* are two mechanisms to secure data following this principle.

Symmetric encryption is based on using an algorithm, which is applied while encrypting in order to transform plain, readable text to ciphertext. This is done using a secret key, which is seen in the Figure 2.2. The output of the encryption seems random to a non-involved third party, which is not in possession of the secret key. When receiving the message in ciphertext, the addressee will use the same secret key to decrypt the message, such that it is readable again.

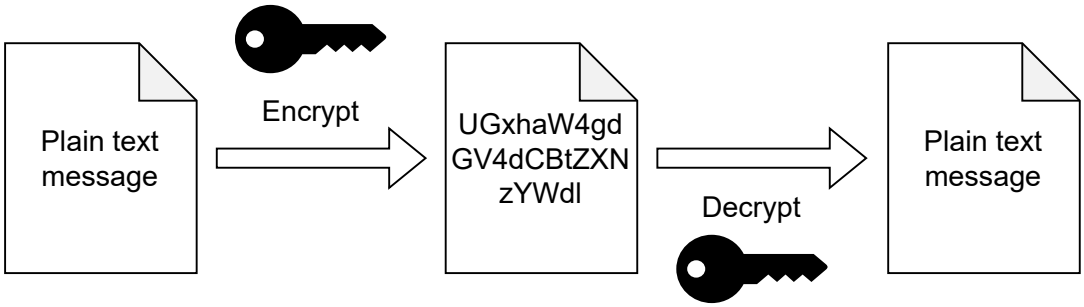

Figure 2.2: Sharing data using symmetric encryption

Using this encryption mechanism, the two parties have to exchange the secret keys, since the encrypted data can only be decrypted by knowing the initial algorithm of encryption [41].

Asymmetric encryption is based on using two different keys: A *public* and *private* key. Each party in a system owns a private and a public key, which is shared to everyone. By using it to encrypt data, a sender ensures that only the owner of the respective private key can decrypt the ciphertext. In Figure 2.3 a simple example of asymmetric encryption is shown, with Bob sending a message to Alice by first encrypting it with her *public key* (dark gray). Only Alice is now able to decrypt the ciphertext message, as she is holder of the associated *private key* [41].

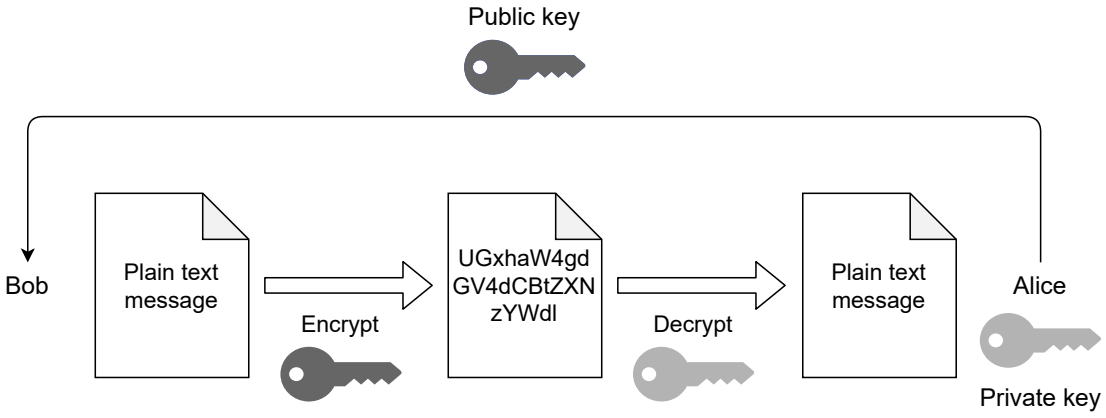

Figure 2.3: Key exchange of asymmetric encryption

Using this encryption approach only works for small data, since the process itself is slow. To mitigate this bottleneck, it is not the underlying data being encrypted but a symmetric key, which is explained at the start of this subsection. This enhances security of symmetric encryption, due to the added security layer of exchanging the symmetric key. Without this it would be possible to *eavesdrop* in order to acquire the shared secret, which is the only item needed to encrypt and decrypt the transmitted data.

2.3.2 Zero Knowledge Proofing

Encrypting data in order to secure it, relies on keeping secret the algorithms used for encryption and decryption. As soon as an algorithm gets leaked, the data can be read by third parties. Such a leak can be avoided by never having to expose the data, not even as ciphertext. However, this needs to be done by maintaining the quality of expression as it would have if the data was exposed encrypted.

Zero Knowledge Proofs (ZKP) are cryptographic protocols for which the verifier is not able to get any knowledge without the actual person to proof, apart from the knowledge that is gained by the task's affordances [4]. Additionally, the proof needs to be legitimate, which is achieved by letting the prover perform multiple iterations. With a significant number of desired outcomes the likeliness of random luck decreases, which undermines trust of the proof and therefore the prover. However, the protocol should not only protect the verifier, but also the prover. Given the prover knows the secret, it is not possible under any circumstances that the verifier gets to know it, even if the protocol is broken. This property also protects the prover from being mocked by the verifier. Since there is no possible exchange of knowledge between the two parties, only the prover will end up to know the secret after proving it.

In Figure 2.4 the Ali Baba's cave problem is shown, which is an example of a ZKP fulfilling all properties. The verifier on top randomly tells the prover on which route it is expected to return. Also the prover arbitrarily chose a route to start. Neither of the two did see each other, which fulfills the property of not sharing knowledge to each other. By doing several iterations the verifier can check if the prover correctly knows the code of the door due to the above described decrease of probability of cheating. Despite these security measures, there is a possibility of a "man-in-the-middle attack" as a third person could catch the traffic coming from the prover. The verifier then could not know that the imposter is only mocking to be the prover [4].

ZKP can cost less computational requirements when compared to public key protocols. This is due to less calculations being used as it has to be done for example, for Rivest-Shamir-Adleman (RSA) [33]. As iterations can be lightweight because of the possibility of splitting the process into light transactions, this approach performs better than others with one heavy transaction.

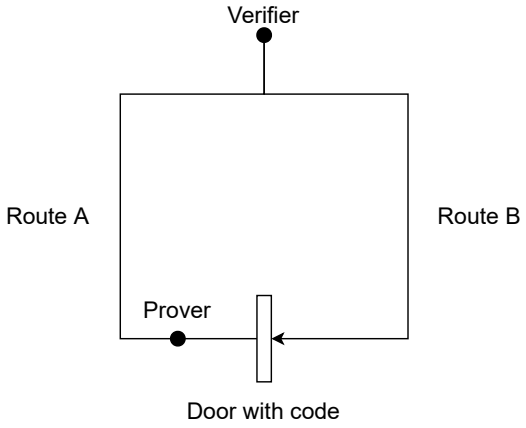

Figure 2.4: Ali Baba's cave

2.3.3 Fully Homomorphic Encryption (FHE)

Instead of not exposing data at all, not being reliant on decrypting data in order to be able to perform some actions, is another solution of achieving a higher security level. This is the target of *Fully Homomorphic Encryption* (FHE). Functions are capable of taking ciphertext as an input, perform the desired actions on the still encrypted data and eventually output ciphertext [32]. Therefore it is not necessary to exchange keys at all, which is one of the main security issues (*e.g.*, "man-in-the-middle attack" or "eavesdropping") in previously reviewed approaches. Figure 2.5 shows how data is shared between the client and a cloud service. The client is the only party in possession of the key to encrypt and decrypt, while the cloud service is able to perform mutations on the encrypted data received.

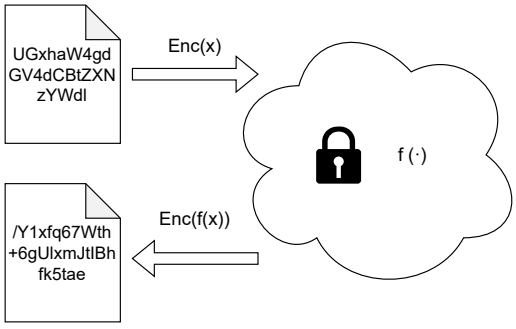

Figure 2.5: Example of data flow between a client and a cloud service using FHE

Due to the complexity of performing calculations on encrypted data, the costs increase drastically [40]. This narrows the use case, due to the fact that for some applications it is crucial to be secure but also responsive. Additionally, it poses other difficulties since programming languages are depending on data being readable, which is not the case when using FHE. If-statements or loops are therefore not as trivial to use as they are with decrypted data. This is addressed by creating standards through providing APIs, which are tackling the manipulation of encrypted data [1].

As seen in the Figure 2.5, FHE is applied to a client-server system. However, with today's P2P approaches like blockchain, there are systems with another network topographic to be covered. In such systems there is not a single counterparty, but there are other nodes being present in the network.

Chapter 3

Related Work

Current research is being conducted by using the described techniques. In Section 3.1 papers are presented, which are showing possible use cases and approaches for solutions by implementing various encryption mechanisms. These papers then are compared and discussed, concluding the chapter in Section 3.2.

3.1 Approaches

3.1.1 FHE

[36] uses an on-chain encryption by dividing accounts into public and private with the respective functions. The details of public accounts, such as the *amount* are publicly visible as well as the corresponding operations. Hence, they are handled as they would be by Ethereum. In contrast, private accounts provide private functions, which are signed with a signature key-pair issued on account-creation. Using this approach, SCs can be created allowing the code to be run on encrypted values. To prove to a miner that the ciphertext can be trusted and satisfy certain conditions, ZKPs are used. After checking the plausibility of these proofs, the miner will then perform the homomorphic actions on the ciphertext and output the result to the blockchain.

Zether relies on a similar approach by using ZKPs to ensure that the ciphertext still is verifiable [8]. A Zether Smart Contract (ZSC) uses Zether tokens (ZTH), which ensures the encryption of the amount. ZTH can be changed back to ETH by uncovering the balance, while providing a proof that the ciphertext is indeed encrypting the exposed balance. However, the design chosen is not fully homomorphic encrypted, regarding the implementation of an anonymous transfer using a ZKP. Since Zether is intended to also cover privacy of transactions along with SC, it is thus needed to allow such an interoperability. This is done by locking accounts to SCs, which restricts the allowed transactions for the corresponding account to be conducted by the SC itself for the time until the account gets unlocked through the SC.

3.1.2 ZKP

Instead of performing calculations with data on-chain and therefore being able to work on encrypted data, [37] uses the approach to apply functions on data by performing them off-chain. This allows the owner of the data to ensure that plain text only exists on its own device, without the need of publishing it on the blockchain. To ensure that altering of private data has been done correctly off-chain, the owner will provide a ZKP, namely a Non-Interactive Zero Knowledge Proof (NIZK). By using such scheme, it is possible to prove selected properties of private data, avoiding any exposure of it without the need of several iteration rounds to be carried out for completing the proof. The *zkay* language then uses these proofs to transform a SC into a fully public *zkay*-contract preserving private data. Implementing this language, Steffen et al. showed a possibility of automatically compiling such high-level privacy statements to low-level primitives in order to be implemented with SCs.

Hawk uses the same technique of utilizing ZKPs to ensure that encrypted data can be proven doubtlessly while being stored on the blockchain [21]. However, in this work, on-chain privacy and contractual security are separated, since different approaches to ensure the security are chosen. *Contractual security* defines the protection of two contract parties, as a blockchain is a trustless system, which also applies to the counterparty of a contract. Such protection includes not only the need for confidentiality, but also for a trustworthy execution and compliance with the agreed terms. In order to ensure this, a *minimally trusted manager* is implemented, which is able to see the user's inputs but is trusted not to reveal any private data from the participating parties. It is possible that the manager colludes with the involved parties or does not follow the protocol properly, but it is not possible to alter the execution of the contract. If the protocol gets terminated earlier than intended, the involved parties receive a compensation and the manager gets penalized. The manager's public key is used to encrypt the inputs prior to submitting them, whereas the ZKP is provided along the ciphertext in order to prove the correctness.

[34] claims that not every SC needs to be verified, which also applies to the underlying code. Regarding a permissioned blockchain, it is possible that contracts do not necessarily have to be verified since there are already existing restrictions concerning the participants. As these are already known, the different properties (content, inputs and outputs) of a contract can be trusted, even though they might be encrypted. In this work proofs are not used only for proving the correctness of the outcome, but also for other properties such as resource consumption. Since blockchains are a trustless system, this predicate applies to code as well. As such, it poses a possible threat if executed on another system. To mitigate this risk, an approach is chosen, which allows equipping code with proofs in order to be able to verify if such code is a possible threat by cross-checking the conclusions of the proofs with a provided security policy. Additionally, the concept of using ZKP to be a "proof of proofs" is discussed to resolve the problem of leakage during the verification of the code. However, such additional privacy specifications lead to more computation costs and are therefore still in need of optimization and overhauling.

ZEXE implements a system, allowing the users to perform off-chain calculations, while still keeping them publicly-verifiable [7]. Information of the underlying transaction as well as the off-chain calculation is not exposed. Additionally, the time used for the validation

is independent of the computational costs which aroused off-chain. All transactions being uniform in terms of validation time lead to the need of Ethereum's Gas becoming obsolete in such a system. ZEXE uses Decentralized Private Computation (DPC) in order to achieve the privacy guarantees not depending on a single chosen application. Since this system aims for function privacy, a classic usage of ZKP is not applicable, as the proof to show the correct evaluation of the function reveals the actual function. To prevent this from happening, ZKP are used in a recursive way to be the proof of proofs. However, since the most outer proof is the ZKP (NIZK in this system), the "inner" proofs do not have to be ZKP, leading to a less costly verification.

3.1.3 Others

Multi-Party Computation (MPC) is an existing cryptographic protocol to do computations with multiple inputs from different parties without revealing any given information to the other participants. In order to send the result to the blockchain and preserving privacy, ZKP were used to be sent together with the result. Due to the lack of practicality and versatility of this solution, the principle of Multi-party transactions (MPT) based on TEEs is introduced [31]. The consistent sequence of state transitions is stored on the blockchain, while the contract's state, inputs and results are privately shown to the owners only. Developers prepare the SC together with a privacy invariant, which the provided engine of the system will then use to generate the desired privacy policy and generating a contract. This contract is compatible with the TEE-Blockchain Architecture. Additionally, a transaction class is created, which allows the participant of the SC to interact with the verifier contract such as an interface. The verifier contract contains the results which can be verified and the update of the states, whereas the private or service contract is keeping the computational logic. Binding the addresses of these two allows to divide the logic as well as the outcome data and therefore also control access rights.

Ekiden introduces an architecture of combining TEEs and blockchains, where the computation is separated from the consensus mechanism [10]. It allows the computational part to be done off-chain in TEEs, which then can be trusted for a proper execution by putting an attest of this on-chain. There are two different types of nodes on the underlying blockchain, one used for consensus and one for computation. Allowing such a division, consensus nodes are not required to run on external hardware, which lets both type of nodes scale in a different manner. Due to the before mentioned attests of off-chain computations, the blockchain is required to be able to verify remote attestation. SCs are kept encrypted on-chain, with digital, symmetric and asymmetric encryption scheme. Thus, there is a *key management committee* to which the Contract TEE has access to in order to retrieve keys, such that the contracts can be en- and decrypted when being executed. Accordingly, the state transition will be encrypted and put on the blockchain. In order to prevent a security breach due to possible key leakage, the keys are so-called *hot-keys* which are derived from a long-term, less-exposed main secret.

[22] emphasizes on distinguishing between performing heavy calculations, such as for sensitive data of a SC and public functions with less computational costs. This division is done after the original SC is developed but not yet deployed, with supplementing functions,

in case any dispute has to be resolved. The on-chain contract then is ready to be deployed by any participant, whereas the off-chain part has to be signed by all participants. It is mandatory that each participant holds a copy of the off-chain contract with the signatures in order to be able to interact with the on-chain contract. After executing computation of the off-chain contracts, the results can be submitted and trigger the state change. Since this can be done by any honest participant, a challenge period is implemented, such that any objections can be resolved. In case of no objections, the state will be changed accordingly as expected. No miner is needed to perform any calculation, since this has been done off-chain without leaking any information of the data. A possible dishonest participant will be resolved in the challenge period, for which there is an extra function implemented in the on-chain contract. Any honest participant will be able to submit the initially received signed copy, which then will be validated by the function and eventually creates a verified on-chain contract.

Voting systems are a use case of SCs. In [13] a possible design of such a system is shown. The SC is kept on the blockchain, with every participant (voter) has to be verified first, in order to check if the user is allowed to vote. Doing such a verification on-chain causes the privacy and anonymity to be at risk, thus, it is required to divide the voter's ID and the encrypted vote. To achieve this, voter verification is done through a central authority (Election Commission), which then distributes random tokens only to eligible voters. Such a procedure allows the voter database to be kept private, since the on-chain SC is publicly available. A hash-database of the issued tokens lets the SC validate the provided token of a voter simply by checking the existence of the hash in the database. A user will also receive an encryption key in order to encrypt the vote, which then will be provided to the SC along with the token. Since the decryption keys are released public after a certain time-frame, an asymmetric key scheme is used. The decryption keys are stored off-chain and distributed by so-called *wardens*, which then will provide the decryption keys at a certain point in time to the SC.

3.2 Comparison and Discussion

Table 3.1 summarizes the approaches described in the paragraphs above. It shows that most of approaches implement ZKP, since this mechanism allows to combine on- and off-chain calculations. Interoperability with existing blockchains is given, since the procedure is kept the same with miners verifying the blocks, in addition of the ability to validate the ZKPs. However, computational costs get higher due to the need of performing several iterations in order to proof the verifier that the correct result was not only caused by chance. This is addressed by using NIZK in [37], [34], [7], which allows reducing the number of rounds to be done.

Generating ZKPs is done off-chain, due to privacy-preserving requirements. As the computation of ciphertext and proofs might lead to a possible leak of private data, it is done off-chain in the investigated related work. The subject of the proofs differs, as in [37] and [21], they are mainly used to be able to transform a classic SC to a more privacy-preserving contract or to simply be able to prove that a provided ciphertext stored on a

SC can be trusted. In [34], the focus lies on verifiability of code, since parties interacting with the SC could risk to face harmful code.

Even though FHE is used in [36], ZKPs are needed, such that verification can be done in a trustless system such as blockchain. Since it is possible to work on ciphertext directly without the need of decryption, computation can be done on-chain, due to the encrypted data in the SC. Miners are responsible for performing the calculation, which reduces the workload on end users, leaving scalability to the blockchain and the respective computational power.

Zether is adding support for anonymity on SCs, which is causing incompatibility with Ethereum, as the computational costs cannot be reflected in Gas due to the block Gas limit [8]. However, the property of a system using homomorphic encryption and providing anonymity is being mentioned in [36] to be targeted in future work; therefore, the amount of computational costs is still subject of further research. Additionally, Zether links the privacy-preserving property to system-owned tokens and SCs which limits the functionality, whereas smartFHE targets a more flexible approach, by not relying on a particular implementation of ZKP and FHE schemes.

Using third parties to preserve the privacy of SCs on the blockchain is another approach being used by [10], [31], [22], and [13]. This breaks with the decentralized aspect, since the third party can be seen as an authority, which has to be trusted in a trustless system. Such a third party can be run in a TEE, as in [31] or [10]. Despite using the same idea, the level of integration into the blockchain ecosystem differs highly. CLOAK requires the developer to already implement a SC with the CLOAK engine, which then defines the further procedure and possible interactions. Ekiden even splits up the blocks into two different types and implements a semi-trusted manager. The architecture of a blockchain system is highly affected by such solutions and trust of the users in the chosen design is a requirement. The *architectural costs* of these systems are higher, but computational costs are lower, due to the ability of using interconnections of a blockchain with the TEEs and therefore not needing to perform heavy cryptographic calculations.

Possible interoperability of the studied encryption schemes is mentioned in [37] and [36], such as ZKP, FHE and TEEs, which shows that these encryption schemes and architectures can be rather complementary than contrasting each other. However, when combining them it is required to assess if possible overhead is being created. Even though privacy can be enhanced, also the computational costs need to be discussed. Zether shows how anonymity can be achieved with current technologies, but it is of limited compatibility with Ethereum for example. Additionally, the need of a tailored blockchain solution could lead to scalability problems, since the approach is not proven with a running system and a throughput seen as for Bitcoin and Ethereum.

In Solidity, the implementation of SC is straightforward due to the lack privacy requirements. Hence, adding such requirements leads to an increase of complexity, which can be observed in an increase of computational costs. Off-chain solutions with TEEs are simple, since the privacy aspect is resolved by adjusting the structure of the procedure rather than the actual hiding of data. Compared to complex cryptographic systems, as for ZKP by using NIZK and Proof of Proofs or for homomorphic encryption, the understanding of

users is less required. However, there are approaches, *e.g.*, [37], which hide the complexity by providing a compiler of the SCs without creating a lot of overhead, which can be derived from the computational cost required to create the actual proofs needed to hide the private data.

Table 3.1: Summary of Related Work

Work	Encryption	Use Case	Blockchain	On-/Off-chain
[36]	FHE and ZKP	Generic encryption of SC through being able to switch between *private* and *public* mode	Ethereum	On-chain
[37]	ZKP	Language for implementing SC which can be transformed to Solidity contracts	Ethereum	Off-chain
[10]	TEE	Separating consensus from execution by using Trusted Execution Environments for SC	TEE-allowing BC	Both
[21]	ZKP	Framework which introduces a minimally trusted manager in order to have a trusted off-chain party	Any	Off-chain
[22]	Any	Enforcing SC to be off-chain for heavy and private functions	Ethereum	Both
[8]	FHE	Targets transactions to be made confidential by making it possible to secure arbitrary SC	Ethereum-like	On-chain
[34]	ZKP	Using proofs or certificates to ensure correct coding of the contract and additionally providing an outsourcing protocol in order to allow offline usage	Ethereum, Cothority, Hyperledger Fabric	Both
[13]	ASYM	Authorization is done off-chain in order to obtain a token, which is a prerequisite to put the encrypted data to the SC. The encryption key is sent to the user which encrypts the content for the SC	Ethereum	Off-chain
[31]	TEE	A framework taking the approach of using Multi-party computation with ZKPs and introducing a TEE-based solution allowing anonymous contract state, inputs and return values	TEE-allowing BC	On-chain
[7]	ZKP	Allows execution of offline computations, which can be verified publicly through ZKP by using computations on a ledger (decentralized private computation)	Any	Off-chain

Chapter 4

Design and Implementation

In this chapter the selected scenario is explained in Section 4.1, which is used to test the implementation. Each encryption mechanism is implemented in a different way, which will be defined in Section 4.2. The source code for the implementation is provided in Section 4.3, organized into the different encryption mechanisms.

4.1 Application Scenario

The implementation is based on a generic scenario of a buyer and seller, using a SC to complete a simple buying transaction between the two parties, shown in detail in Figure 4.1. The architecture is originated in three interconnected layers: *Users*, *Smart Contract* and *Blockchain*.

The **seller** is a contracting party on the user level, trading goods or services upon receiving the agreed price. As an owner of the transaction's offered subject, the contract formalizes a needed procedure in order to eliminate risks for the contracting parties. By doing so, the seller will ensure, that the exchange of the good or service is only done, if the required return of funds is confirmed.

The **buyer** on the user-level is the counterparty in the contract. In order to obtain the good or service, the buyer is obliged to provide the funds being stated in the contract. These terms guarantee the desired transaction good or service to be delivered by the seller, as soon as the buyer fulfilled the contractual requirements.

The contract established between the two parties contains the required information, such as the identification of the buyer and the seller, in order to set a legal binding connection. The amount to be spent by the buyer and the description of the good, which will be transferred as soon as the provided amount has been validated with the contract terms, is also formalized. Concluding the user level, the parties agree on this contract, which is then programmatically represented through a SC, as it is shown in the Smart Contract layer of Figure 4.1. This layer is subdivided, since there are two approaches, which can be used to set up the SC.

An **on-Chain** solution is used to store the data encrypted in the SC, without the need of a third party. Only the contracting parties are able to decrypt the ciphertext, which requires the miners to validate the block even though it contains encrypted values. Data in the SC can be arbitrarily encrypted, *e.g.*, if not necessary, the contracting parties can leave selected data unencrypted to improve the calculation costs.

Using an **off-Chain** approach, data does not need to be stored in the SC. A third party either stores the values or is used to perform calculations on them. The chosen third party has to be accepted by all contracting parties and needs to provide sufficient proof of correct manipulations in order to verify the correctness by the involved parties but also by the miners, to ensure interoperability with the blockchain.

A simulated **blockchain** running on a local machine will be used to apply the scenario. Any blockchain supporting SCs can be considered, even though this work considers only one. Once the required parties agreed on the content of the contract, it gets transformed into a SC and is deployed to the blockchain. Transaction handling will then be done by the SC, following the implemented procedure automatically.

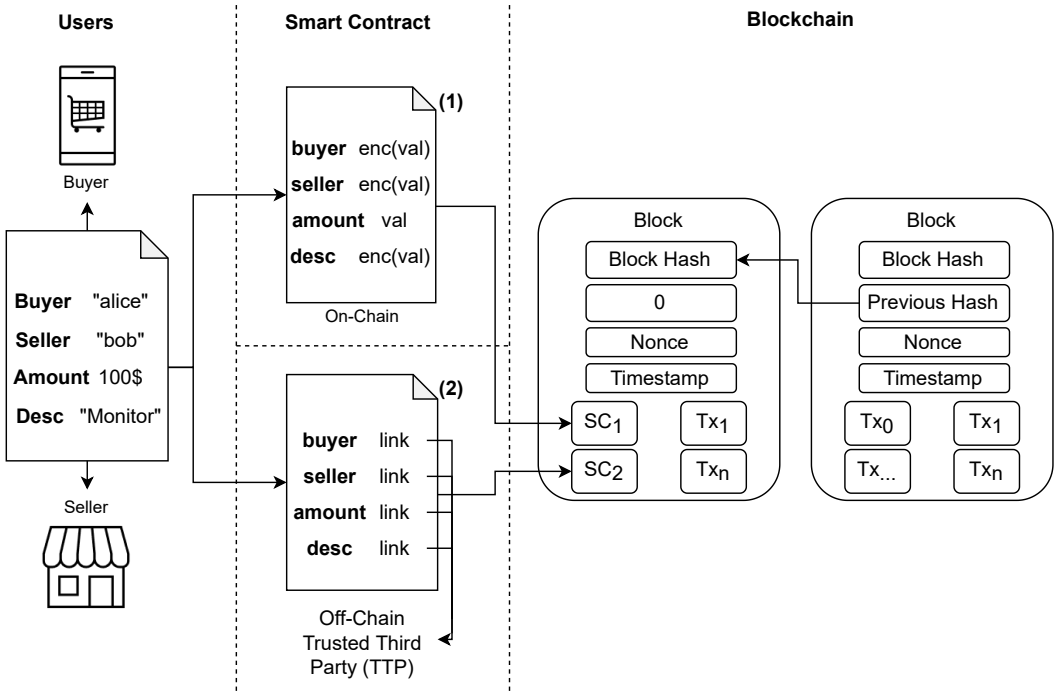

Figure 4.1: Scenario Considered in this Thesis

4.2 Encryption Approaches

This section describes the design of the selected encryption approaches. First, an application of the scenario by using symmetric encryption is shown in Section 4.2.1. Section 4.2.2 explains the approach using asymmetric encryption. In Section 4.2.3 the same scenario will be used to preserve privacy for a SC, using FHE.

4.2.1 Symmetric Encryption

Storing encrypted values in a SC only allows the holders of the corresponding keys to encrypt this data when using asymmetric and/or symmetric encryption. Since permissionless blockchains do not require knowing the verifying party (*miners*), the keys are only shared between the contracting parties. Hence, miners will not be able to perform any other mutations of this data but replacing it, due to the lack of the key allowing them to decrypt.

In the chosen scenario all fields of the SC are encrypted, which causes them to be immutable when using the implemented SC-functions and symmetric encryption, since the secret key must not be publicly available in order to maintain its privacy. However, the `get`- and `set`-functions can be used to retrieve data and alter it off-chain before updating it on the SC. A `struct` is used as an abstraction representing the required data of the **buyer**, **seller** such as identifiers **name**, **address**. Since the description of the traded good **desc** and the **amount** are not related to the contracting parties but are a subject of the contract itself, they are stored separately.

The scenario assumes that the contractual parties agreed on a Key Manager and that they are using a secure channel to exchange them. It is not possible to store these keys in the SC, preserving the property of the system being trustless, as the keys are stored publicly on the blockchain, resulting in compromising the secret immediately after deployment. Additionally, sharing the keys leads to miners being able to decrypt sensitive data leading to vulnerability of the system through data leakage.

4.2.2 Asymmetric Encryption

Using a key-pair of public and private-keys to achieve an asymmetric encryption, allows the contracting parties' public keys to be stored on the blockchain. Hence, the struct `ContractingParty` is extended by an additional field **pubKey**. If any contracting party wants to set or adjust encrypted data, it is fetched by calling the corresponding function. Since the data is encrypted, it first has to be decrypted by using the fetcher's private key if the data was initially encrypted by the counterparty. This can be skipped if the last value was set by the fetcher. In order to store altered data in the SC, the counterparty's public key has to be retrieved by calling the corresponding function. The key then has to be rebuilt from a primitive to the encryption library's representation, such that a successful encryption is performed through using this key to create the correct ciphertext.

This implementation assumes that both parties know, which cryptography library is used to encrypt data. Otherwise the recreation of the public key from the retrieved primitive stored in the SC can not be done. Due to the data being shared encrypted without exception, miners are not able to verify the correctness of it. This can only be done by holders of the corresponding private keys, hence the contracting parties.

4.2.3 FHE

The asymmetric approach of having a key-pair with public and private keys is used for the homomorphic approach as well. Due to the goal of being able to perform mathematical calculations on ciphertext, these keys are only used to encrypt the **amount**. Therefore, `string` types such as **desc** or **name** still need to be encrypted by the same mechanism used for the asymmetric encryption. The struct `ContractingParty` is adjusted accordingly, now storing two different public keys, one for the RSA-keypair **pubKeyAsym** and one for the homomorphic encryption mechanism **pubKeyHE**. Since the used library for homomorphic encryption requires multiple values to ensure recreation of the library's representation of an encrypted number, the **amount** is stored as a `string` containing a JSON, which contains the list of values.

4.3 Implementation

In this section, the implementation details of the approaches are presented, using the different encryption mechanisms to store encrypted data on the SC and retrieving them in order to validate that the chosen mechanisms indeed work when decrypting the ciphertexts after encryption and storing them on the blockchain. This is a deviation from the envisioned scenario in Figure 4.1. The adjusted implementation merges the two approaches **on-chain** and **off-chain**, by encrypting each field but storing them off-chain, assuming there exists a secure environment without making any further assumptions on its design. This preserves the privacy of the SC but prevents third parties to be able to verify its content due to inaccessibility of the decryption key. From this implementation small snippets are then derived for executing sample runs, which yield the data to compare the different approaches in Chapter 5.

4.3.1 Symmetric Encryption

In order to create the encryption key, the `Fernet` package is used, which can be seen in Listing 4.1. Its encryption is based on the Advanced Encryption Standard (AES), with a block size of 128 bits using the PKCS7 padding algorithm [24]. Since the test suite assumes a secret channel between the contracting parties, the key variable `secr_key` in the test suite is accessible to both parties without an exchange, as such a procedure is not subject of this thesis.

```
1 from cryptography.fernet import Fernet
2 ...
3 secr_key = Fernet.generate_key()
4 fernet = Fernet(secr_key)
5 ...
```

Listing 4.1: Key generation using symmetric encryption

Encrypting the values requires the data to be stored as bytes, since the encryption function only allows an input of this type. After encrypting the fields by calling the `encrypt()` function, the values are returned as a bytestring as well, requiring the corresponding field type in the SC to be `bytes`, which represents an array of type `bytes`. The account addresses stored in the struct `ContractingParties` are arbitrarily chosen from the address list, which can be fetched from the blockchain. Along with the encrypted name of the respective contracting party, the chosen addresses are sent to the SC by using the `setSeller()` and `setBuyer()` functions. After a full initialization of the SC with every field set up, the data is again fetched through the `get`-functions. Since the key must be stored off-chain, the surcharge can only be added after decrypting the **amount** retrieved. This new amount is sent to the SC, whereafter the test suite retrieves all data stored on the SC, comparing it to the initial plaintext value of the script to validate the equality.

4.3.2 Asymmetric Encryption

The key-pair is created by using the RSA (Rivest-Shamir-Adleman) public-key cryptosystem. As shown in Listing 4.2, the python package `Python-RSA` is used to generate the keys, as well as using the provided algorithms to encrypt and decrypt plaintext data. Since a key-pair of corresponding public and private keys are created, keyrings could be used. However, for this implementation there are only two key-pairs generated for the contracting parties **buyer** and **seller**, making a clear overview possible due to the amount of four keys to be handled, hence no key-ring is needed.

```
1 import rsa
2 ...
3 buyer_pub, buyer_priv = rsa.newkeys(512)
4 seller_pub, seller_priv = rsa.newkeys(512)
5 ...
```

Listing 4.2: Key generation using asymmetric encryption

The keys are created first in the test suite using the `rsa.newkeys()` function, which takes a keysize between 128 and 4096 bits as input. Since this thesis discusses the different approaches without targeting the highest security, a keysize of 512 bits was chosen in order to decrease the amount of time spent to generate the key. As for the symmetric encryption the scenario shown in Figure 4.1 has been altered to match a possible implementation for the asymmetric encryption. It differs from the approach chosen in Section 4.3.1, as it is possible to share a public key. This allows a third party *e.g.*, the contracting counterparty, to encrypt data such that only the holder of the corresponding private key can decrypt it. Therefore the public key can be transformed from the package-internal representation to a primitive `string` value by calling the `_save_pkcs1_pem()` function.

Due to the property that only the corresponding private key can decrypt ciphertext, the set up of the test suite is done accordingly. Hence, the **seller** is fetched first, such that the **buyer** is able to encrypt the **name** with the seller's **pubKey**. This is also reflected in the tests, which decrypt the fetched data from the blockchain using the corresponding private key.

4.3.3 FHE

Since this thesis uses the SC-language Solidity, the implementation of the homomorphic encryption approach focuses on the off-chain calculation with ciphertext, as Solidity can only load deployed libraries, thus libraries exposing their implementation. The standard implementation of this SC-language only allows the manipulation of primitives, *e.g.*, mathematical operations on integers. It is therefore not possible with the current version to implement a logic of a calculation using ciphertexts without exposing implementation details to the public, which causes a security-breach.

The `python-paillier` library from the `phe` package allows performing mathematical operations on ciphertext. This library is using the same approach of asymmetric encryption in order to transform plaintext to ciphertext. If no input is given, as shown in Listing 4.3, the function `generate_paillier_keypair()` creates a key with size 2048 bits.

```
1 import rsa
2 from phe import paillier
3 ...
4 fhe_sel_pubk, fhe_sel_privk = paillier.generate_paillier_keypair()
5 fhe_buy_pubk, fhe_buy_privk = paillier.generate_paillier_keypair()
6 asym_buy_pubk, asym_buy_privk = rsa.newkeys(512)
7 asym_sel_pubk, asym_sel_privk = rsa.newkeys(512)
8 ...
```

Listing 4.3: Key generation using homomorphic encryption

The key is casted to a `string` value, as it is done in Section 4.3.2, taking only the modulus of the public key. This allows to recreate the internal representation of the public key, after fetching the value from the chain through the `getSeller()` function as shown in Listing 4.4 on line 2. The following encryption of the `string` values is taken from the asymmetric implementation, since the `encrypt()` function's parameters can only be of type `int`. Therefore, the `string` values, such as **desc** and **name** are encrypted through the above introduced `rsa` cryptographic library.

Since the library-internal representation of an encrypted number requires a tuple of ciphertext and exponent, both of these values have to be stored on the blockchain. The library's documentation suggests this serializiation to be done in a JSON-format, which can be stored as a `string`-primitive [14], shown in Listing 4.4 on line 15-19.

```
...
# Transform the integer of the public key to the package's
    representation
chain_fhe_sel_pubk = paillier.PaillierPublicKey(n=int(self.seller_struct
    [3]))
bname = rsa.encrypt(bname,chain_asym_sel_pubk)
dgood = rsa.encrypt(dgood,chain_asym_sel_pubk)

contract.functions.setBuyer(
    web3.toChecksumAddress(web3.eth.accounts[2]),
    bname,
    rsa.PublicKey._save_pkcs1_pem(chain_asym_sel_pubk),
    str(self.fhe_buy_pubk.n)
    ).transact()
...
# Encrypt the amount using the HE-key and putting the required values to
    the JSON
enc_amt = chain_fhe_sel_pubk.encrypt(self.amount)
enc_amt_json = {}
enc_amt_json['values'] = [
    (str(enc_amt.ciphertext()), enc_amt.exponent)
    ]
enc_amt_json = json.dumps(enc_amt_json)
...
```

Listing 4.4: Serialization of internal representation

Chapter 5

Evaluation and Discussion

This chapter evaluates the findings of the measurements in Section 5.1, which are done based on the implementation shown in Chapter 4. The results of the conducted measurements are presented in Section 5.2. In Section 5.3 the evaluation of the data is discussed. Section 5.4 then shows the challenges encountered during the design and implementation phase of this thesis.

5.1 Evaluation

The evaluation of the test suite for the four off-chain approaches *unencrypted*, *symmetric*, *asymmetric* and *homomorphic encryption* was conducted on a 4-Core Intel(R) Core(TM) i7 CPU @ 1.70 GHz with 16 GB of RAM. Three metrics are compared: In Section 5.2.1 the contract-size, in Section 5.2.2 amount of Gas used and in Section 5.2.3 the runtime for fetching the **amount** and adding the surcharge amount before sending it to the SC.

5.1.1 Mutual setup

In order to assess the respective scenarios, a test suite is set up using the Python package `unittest`. The suite validates a certain procedure of setting up the contract with the required fields and store encrypted values on the SC. A local blockchain is set up with Truffle Suite, which provides a GUI for visualizing the current state of the blockchain as well as a transaction list. After setting up the blockchain by executing the command `truffle init`, 10 accounts are created with an account balance of 100 ETH. The connection between the blockchain and the script is established through the `web3` package, allowing interaction with the SC. In Listing 5.1 on line 3 the IP-address to the blockchain is given, which implies that the user could change to a blockchain simulator (*e.g.*, Ganache, testnet or mainnet) by adjusting to the corresponding IP-address.

After successfully connecting the script to the blockchain network, the contracts have to be fetched, in order to allow interaction with the implemented functions. This requires

the contract to be deployed to the blockchain, which is done by executing the command `truffle migrate`. The deploying process yields the contract address, which can be fetched by reading the created JSON-file of the deployed SC shown on lines 10 and 15. Putting together the address and the contract ABI, the contract object is set for the further procedure based on the different encryption mechanisms.

```
1     ...
2     # truffle development blockchain address
3     blockchain_address = 'http://127.0.0.1:7545'
4     # Client instance to interact with the blockchain
5     web3 = Web3(HTTPProvider(blockchain_address,request_kwargs={'timeout
          ':3600}))
6     # Set the default account (so we don't need to set the "from" for
          every transaction call)
7     web3.eth.default_account = web3.eth.accounts[0]
8
9     # Path to the compiled contract JSON file
10    compiled_contract_path = 'build/contracts/homenc.json'
11
12    with open(compiled_contract_path) as file:
13        contract_json = json.load(file)  # load contract info as JSON
14        contract_abi = contract_json['abi']  # fetch contract's abi -
              necessary to call its functions
15        deployed_contract_address = contract_json['networks']['5777']['
              address'] # Deployed contract address (see `migrate` command
              output: `contract address`)
16
17    # Fetch deployed contract reference
18    contract = web3.eth.contract(address=deployed_contract_address, abi=
          contract_abi)
19    ...
```

Listing 5.1: Connection setup and contract fetching

5.1.2 Symmetric encryption

In order to achieve comparability between the different encryption approaches, the runtime sample is taken from a shortened snippet in 5.2. The contract is set up and deployed in a reduced manner, by only setting the **amount**. It is then fetched through the `getAmount()` function for a decryption done off-chain by using the initially created secret key. After transforming `bytes` to `int`, the surcharge amount can be added. As mentioned above, it is required to transform this new amount to `bytes` before sending it to the SC by using `setAmount()`. The time is measured starting from the first retrieval of the **amount** from the chain until the new encrypted **amount** is sent to the SC.

```
1 ...
2 start_time = time.time()
3
4 # Fetch the amount to decrypt it, add the surcharge and send it to the
      SC
5 chain_amt = contract.functions.getAmount().call()
6 dec_amt = fernet.decrypt(chain_amt)
7 new_amt = int.from_bytes(dec_amt, "big") + surch_amount
```

```
8  new_amt = bytes([new_amt])
9  new_enc_amt = fernet.encrypt(new_amt)
10 contract.functions.setAmount(new_enc_amt).transact()
11
12 diff = time.time() - start_time
13 ...
```

Listing 5.2: Runtime measurement for symmetric encryption

5.1.3 Asymmetric encryption

The setup of the SC follows the same procedure as in Section 4.3.1, but due to the different encryption approach, the examined snippet yielding the runtime of changing an encrypted amount retrieved from the chain deviates from the sample shown before. As shown in Listing 5.3 on line 4, the **seller** struct has to be first retrieved, since the public key is stored there. The `string` value then is transformed to the `rsa`-internal representation of a public key, such that it can be used for encrypting the new calculated amount on line 11.

```
1  ...
2  start_time = time.time()
3
4  # Fetch the seller's struct from the blockchain to retrieve the public
       key
5  seller_struct = contract.functions.getSeller().call()
6  bc_key = rsa.PublicKey._load_pkcs1_pem(self.seller_struct[2])
7
8  # Fetch the amount to decrypt it, add the surcharge and send it to the
       SC
9  chain_amt = contract.functions.getAmount().call()
10 dec_amt = rsa.decrypt(chain_amt,seller_priv)
11 new_amt = int.from_bytes(dec_amt, "big") + surch_amount
12 new_amt = bytes([new_amt])
13 new_enc_amt = rsa.encrypt(new_amt,bc_key)
14 contract.functions.setAmount(new_enc_amt).transact()
15
16 diff = time.time() - start_time
17 ...
```

Listing 5.3: Runtime measurement for asymmetric encryption

5.1.4 FHE

Instead of decrypting the **amount**, as it was done in Section 5.1.3, the addition can be done without the usage of a private key, due to the properties of homomorphic encryption. However, it is required to transform the encrypted value stored as a primitive in the SC to an `EncryptedNumber`. This is done by putting the values of the retrieved JSON to a dictionary and insert the needed entries together with the retrieved public key as arguments of the corresponding object. After performing the desired calculation on line 15 in Listing 5.4, the needed entries of the JSON are set again, such that it matches the type of the SC's field.

```
...
start_time = time.time()

# Fetch the seller's struct from the blockchain to retrieve the public
    key

seller_struct = contract.functions.getSeller().call()
chain_fhe_sel_pubk = paillier.PaillierPublicKey(n=int(seller_struct[3]))

# Fetch the amount to decrypt it, add the surcharge and send it to the
    SC
chain_enc_amt = contract.functions.getAmount().call()
received_dict = json.loads(chain_enc_amt)
chain_enc_amt = paillier.EncryptedNumber(
    chain_fhe_sel_pubk,
    int(received_dict['values'][0][0]),
    int(received_dict['values'][0][1])
    )
enc_surch_amt = chain_fhe_sel_pubk.encrypt(surch_amount)
new_enc_amt = chain_enc_amt + enc_surch_amt
new_enc_amt_json = {}
new_enc_amt_json['values'] = [
    (str(new_enc_amt.ciphertext()), new_enc_amt.exponent)
    ]
new_enc_amt_json = json.dumps(new_enc_amt_json)
contract.functions.setAmount(new_enc_amt_json).transact()

diff = time.time() - start_time
...
```

Listing 5.4: Runtime measurement for homomorphic encryption

5.2 Results

In this section the results of the conducted research on different encryption approaches for SCs are presented and evaluated.

5.2.1 Contract Size

Whilst running the test suite, the SC is deployed to the blockchain and values are set. The measurements were taken after performing these steps. In order to determine the contract size, the plugin `truffle-contract-size` was used to calculate the SC size by taking the bytecode as a reference. Since there is a limitation of Ethereum for a SC to use less storage than 24 kBytes due to the current Gas limit, this metric is needed to assess if the SC can be constructed like envisioned to be put on the blockchain [27].

Figure 5.1 shows that the contract size of the *unencrypted* approach and the *symmetric* encrypted approach is marginally different. Since there is the same amount of fields on the contract, as the key can not be shared and therefore is not put on the SC, the

only difference is the size of the values set to these fields. Investigating the size of one encrypted value using the `Fernet` package yields a field size of 133 bytes, which in the applied scenario summed up to a difference of 0.01 kBytes.

Since the field size was calculated in the Python-script using the function `sys.getsizeof()`, comparability to the storage needed on the SC is not necessarily given, due to the values being translated to the contract's bytecode. Nevertheless, this method gives insight on the size of the input-data before the translation and therefore is at least a guideline.

In contrast *asymmetric* encryption with the `rsa` package is using 97 bytes per encrypted value, which is less storage used than for the *symmetric* approach. However, the keys are stored on the SC, since public keys can be shared. This increases the contract by one additional field per struct `ContractingParty`, hence two fields in total with each of these being of size 195 bytes.

The same effects can be observed for *homomorphic* encryption, which is requiring two additional fields in the struct `ContractingParty`. Both of them are of type `string`, as it is for the asymmetric keys. However, the difference in increments of size changes when comparing the increases of $|size(symmetric) - size(asymmetric)|$ and $|size(asymmetric) - size(homomorphic)|$. Investigation on this showed, that the contract size is not only dependent on the amount of fields, but also the defined type for each field. Therefore, changes of types are also affecting the size of the SC, reflected in Figure 5.1, since the type of the amount needed to be changed from `bytes` to `string`.

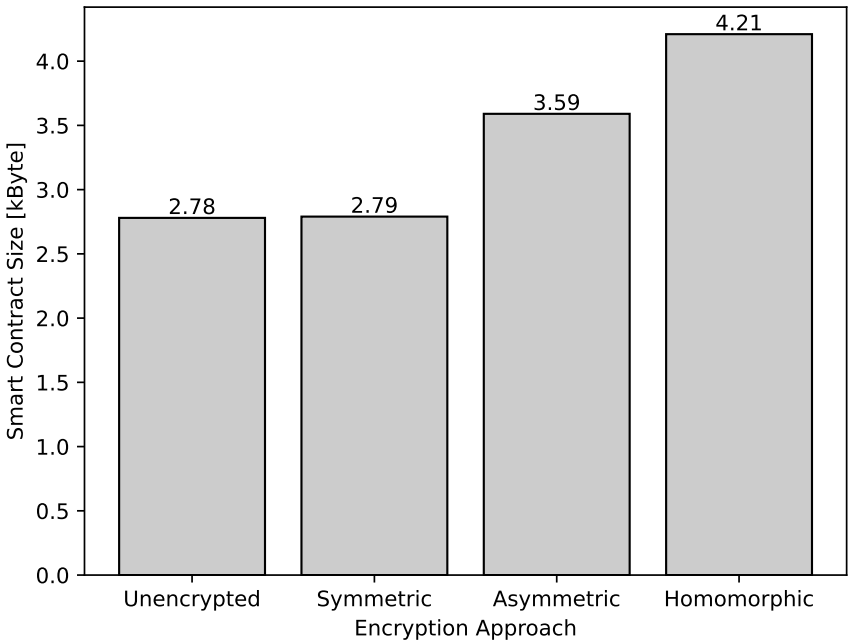

Figure 5.1: Size of the SCs after one run of the test suite

5.2.2 Used Gas

The amount of Gas used for a transaction allows observation of economic aspects for working with SCs. As shown in Section 2.2.1, the amount of Gas used is limited and therefore is required to be considered prior to implementing a SC. If a contract's complexity affords an exceeding amount of Gas, the implementation will fail and therefore impacts the design process directly. The transaction price in ETH can be calculated easily by the formula $GasUnits(limit) \times (BaseFee + Tip)$, where the base fee is denoted by the block and the tip is usually set by the wallet [26].

The Truffle Suite provides data on each transaction done with the SC in either the Ganache GUI or using the Ganache-CLI. Since the `get`-calls are done by a script and therefore not by another SC, the Gas used for `get`-functions are not included in the evaluation, since they do not apply in this case. Similar to the observation made in Section 5.2.1, the deployment of the SCs of both approaches, *unencrypted* and *symmetric* are using the same amount of Gas, since the fields are the exact same on both implementations as shown in Figure 5.2. However, the setting of values requires a different Gas amount, due to the different size of the values. Setting the **buyer** and the **seller** uses almost the same amount of Gas, since they are identical in amount and types of fields.

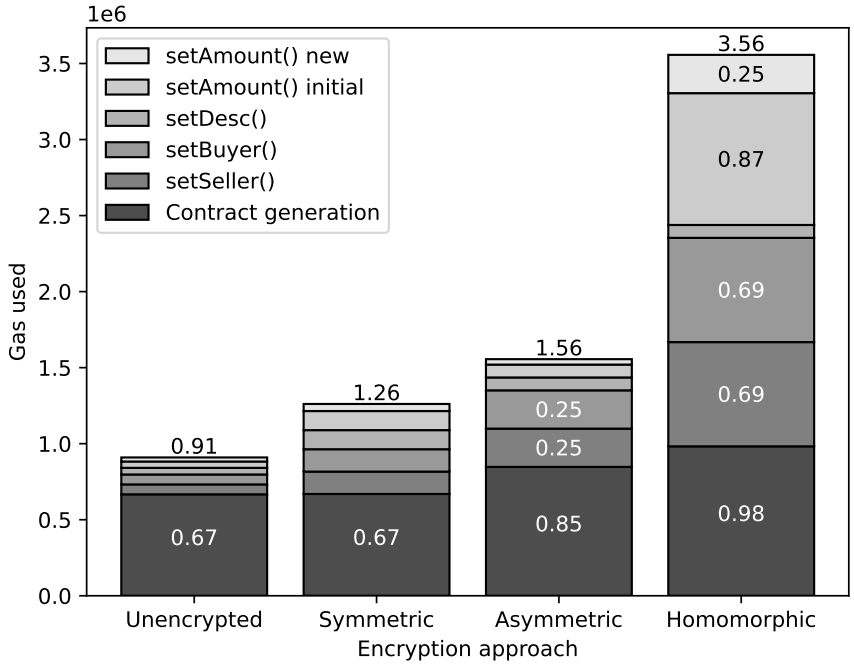

Figure 5.2: Used Gas after one run of the test suite

The only slight difference is observed when investigating the values, which can differ in length, *e.g.*, when using a longer unencrypted name or special characters in a string, which require a different amount of bytes than a ASCII-only string. This effect is highlighted by the amount of Gas used for the initial call of `setAmount()` while using the *homomorphic* approach. Considering only this call, the usage of Gas is almost as high as it is for the

unencrypted procedure. Additionally, setting the public keys in the `ContractingParty` struct results in a significant increase, when comparing the *asymmetric* and *homomorphic* approach. Setting the **desc** however, uses more Gas for the *symmetric* approach than for the *asymmetric* and *homomorphic* ones, since the latter ones are both using the `rsa`-package for encrypting this field. This implies that the ciphertext created out of the same `string`-value as it has been used for the *symmetric* approach is less costly to be set to the SC.

5.2.3 Runtime

The snippets shown in Section 5.1 were used to create a data set of 30 entries, measuring the time of consecutive runs in order to compare the different encryption approaches. In the chosen scenario, the execution time of a SC determines the outcome of a transaction, *e.g.*, the timespan between receival of the required payment and the subsequent delivery of the traded good.

Deriving from Figure 5.3, the *unencrypted* approach performs the fetching, manipulation and update of the **amount** in less than a second. The *symmetric* and *asymmetric* approaches were using double this time, with the *asymmetric* approach being more consistent than the *symmetric* one. This small difference implies that fetching of a public key, as well as transforming it before encrypting the **amount** costs only around 100 milliseconds.

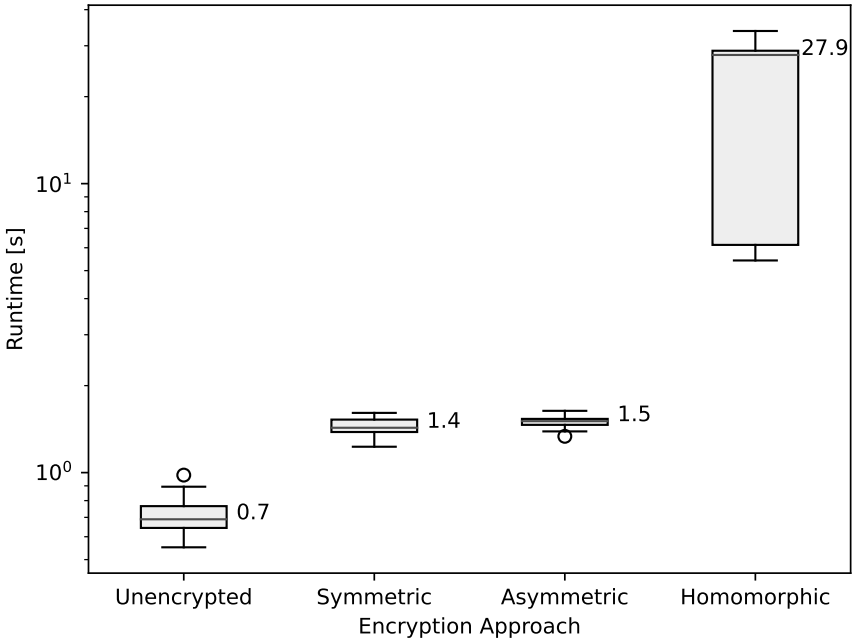

Figure 5.3: Runtime for the above introduced snippets

Comparing the first three approaches with the *homomorphic* one results in a huge difference of runtime. The *homomorphic* approach was taking 25 times as long as the other

approaches. However, the range of runtimes is significantly wider. Nevertheless, the median lies closer to the third quartile of the data set. After performing some investigation the lines of code, which caused this effect were found in the calls to the deployed SC. There is no evidence of a slow homomorphic calculation, as summing of two ciphertexts took less than 20 milliseconds after running the calculation 30 times. Therefore the amount of time used to perform the calculation correlates positively with the size of the SC's fields.

5.3 Discussion and Comparison

The results shown in Section 5.2 revealed a similar pattern for the comparison of the encryption approaches. The more security properties are added to be stored on-chain by each of the approaches, the higher the cost gets. However, the application scenario has to be carefully considered, since higher costs are in some use-cases negligible.

Managing the contract-size is not only important since the need of storage is directly linked to costs *i.e.*, Gas used, but also since there is a Gas limit and the bytecode must be put on one single block in Ethereum, which is a way to check the technical feasibility of a designed SC. Furthermore, when comparing Figures 5.1 and 5.3 there is a correlation of size and runtime, which adds the soft-factor of usability: If in a scenario with a hybrid on-off-chain solution, the triggering of the delivery is time-critical, the encryption approach should be less complex and rather be focused of having a sleek implementation in terms of storage usage for keys and encrypted values.

This thesis shows that by implementing different off-chain encryption approaches, there is a direct impact on economics, since the amount of Gas eventually is paid in ETH. Each change to the SC results in additional Gas costs. However, this has to be put in contrast with the current set-up of physical contracts. When a property of a physical contract is changed, there are legal costs due to the involvement of legal departments setting up a new contract or adjusting the existing one. Changes in privacy of such contracts are only reflected transparently in SC, as the contracting parties are able to share physical contracts on their own, because there is no underlying public system such as the blockchain.

Using an off-chain approach increases the flexibility of choosing the encryption method significantly, since there is no limitation of the SC programming language or the blockchain as it is when trying to implement it on-chain. For the scenario of this thesis using a Ethereum-based blockchain, there was no possibility presented to implement *symmetric* and *asymmetric* encryption on-chain as shown in Table 5.1. This is due to the inability of performing calculations on-chain, since the encrypted value has to be fetched in order to be decrypted with the secret decryption key.

The implementation showed that the used homomorphic encryption library only allows encryption of numeric values. This was creating overhead of storing two more public keys, as one key was needed for the string encryption with the *rsa* package and the other for the encryption of the **amount** with *python-paillier*. This is reflected in the increase of storage and Gas used in Section 5.2.

Table 5.1: Comparison of the encryption approaches

Property \ Encryption	Symmetric	Asymmetric	Homomorphic
On-Chain only	✖	✖	✔
Off-Chain	✔	✔	✔
Encrypting strings	✔	✔	✖
Encrypting numeric values	✔	✔	✔
Performing calculations	✖	✖	✔

Nevertheless, the implementation of the adjusted scenario using off-chain components breaks the trustless and distributed approach of a blockchain. The contracting parties are required to agree on certain properties, such as the usage of the same cryptography library, which can not be done in a trustless system. Therefore a communication channel dedicated to the contracting parties is required, which could reveal the counterparty's identity if not providing the respective security measures.

5.4 Challenges

In this section the challenges of the encryption approaches during the implementation and evaluation process are presented and compared.

5.4.1 Key size

In order to implement the scenario using the homomorphic encryption scheme, the package `Pyfhel` was chosen first. It uses the same asymmetric encryption approach for the key-pair, generating a public and a private key upon calling the corresponding function. Compared to the `python-paillier` package, `Pyfhel` offers more functionality such as multiplying two ciphertexts. Additionally, a function is offered to convert the key and generated ciphertext to the primitive `bytes`, allowing the distribution of the public key and the generated ciphertext.

After performing these steps, the storage needed for the transformed ciphertext and key was checked, resulting in 32 kBytes. As discussed above in Section 5.2.1, the maximum amount of the whole SC is set to 24 kBytes hence, only the key itself exceeds the limit. Ganache denied the transaction to the SC accordingly, resulting in a `ConnectionTimeout`. Since it was not possible to put the key to the SC, the contracting counterparty could not fetch the public key from the SC and therefore could not perform the calculations off-chain. Hence, it was similar to the symmetric approach due to the non-shareable key, but with the added possibility to perform calculations on ciphertext.

5.4.2 Representation of Ciphertext

When encrypting plaintext using a cryptographic library, the output differs based on the implementation. This applies for the two introduced libraries `python-paillier` and `Pyfhel`, with a 4091 bits long integer for `python-paillier` and the `bytes`-array for `Pyfhel`. Due to the discussed impacts of types on a SC, this affects the runtime as well as the Gas used. Additionally, a workaround for the integer value of `python-paillier`'s encrypted value was required, since the maximum amount bits of the EVM is 256 (`uint256`). Therefore, the `string` representation was used, as it allows to store a longer sequence of characters. The JSON-format then was used, due to the need of storing the corresponding exponent and providing a simpler accessibility of these values.

Since `Pyfhel`'s keysize and the size of the encrypted **amount** exceeded the limits of the SC, the `hashlib` package was used to hash these values, decreasing the size. Due to the property of being deterministic, the same input produces the same output. However, when performing the required steps for hashing the encrypted amounts, the equality check was never successful. This was due to the implementation of `Pyfhel`, as the bytestrings generated after encrypting the same input were different to each other. When decrypting them with `Pyfhel`, the original values were returned correctly. This implies that either `Pyfhel`'s decryption function or the `to_bytes()`-function is not deterministic.

Chapter 6

Summary and Future Work

By proposing the Smart Contract (SC) concept in 1994, Nick Szabo introduced a possibility of an automated implementation of physical contracts, which grew in popularity a decade later with the rise of blockchains. By disrupting the industry with forming an alternative to existing distributed systems, the technology was soon used to implement Szabo's proposal.

Being a public, permissionless system, privacy of SCs was not given, since every participant in the blockchain-specific network could access its content. Therefore, new ways of enhancing privacy were found in encryption, as the core properties introduced by the blockchain should be respected. While the resource consumption grew with the size of the blockchain as well as its popularity, adding cost-intensive encryption for SC had to be well-considered.

This thesis focused on assessing such encryption approaches by implementing a simple scenario of two parties agreeing on a contract to do a purchase transaction. The four approaches *unencrypted*, *symmetric*, *asymmetric* and *homorphic* were implemented by setting up a corresponding SC on a local Ethereum-based blockchain.

The implementation showed that on-chain approaches require tailored solutions based on the selected cryptographic mechanism. Therefore, a hybrid approach was chosen with the SC serving as a storage of the encrypted values as well as public shared keys, while cryptographic actions and calculations were performed off-chain.

Performance measurements yielded a significantly higher runtime for the *homomorphic* approach. This was caused by the higher amount of needed memory due to the requirement of storing more and bigger values, such as encrypted fields or public keys, on the SC. A direct impact on economic aspects could be observed, as the amount of Gas used is linked to the complexity, the amount of operations and the accesses to the storage being performed. Therefore, carefully considering the needed amount of security on different aspects of a SC decreases complexity, hence computational and monetary costs.

This thesis could emphasize on the need of increasing efficiency of encryption mechanisms, in order to achieve privacy without economic drawbacks. Reducing the size of encrypted values or keys, while maintaining their security aspects, increases the amount of use cases

for which the encryption mechanisms could be used. Additionally, this could result in digitizing legal artifacts such as contracts.

The implemented SCs were mainly designed for applying the encryption mechanisms for an envisioned scenario. However, there are possibilities to optimize SCs, such that the amount of Gas and storage needed can be reduced. These optimizations then could lead to a decrease of the measured aspects and showcase an economically enhanced version. Additionally, this thesis focuses on Solidity as the contract-language and did not investigate other frameworks for SCs. Current research showed, that the same encryption approaches with tailored contract-solutions offer more flexibility in terms of on- and off-chain implementations as well as more economic efficiency.

The usage of asymmetric keys is a drawback when thinking of having multiple contracting parties instead of two. Since the stored values can only be decrypted by using the corresponding private key, there is only one party, which is able to decrypt. This could be improved by either encrypting the value with a symmetric key, which is then shared on-chain encrypted by the asymmetric keypair or a design-change of the system such that multi party encryption is possible.

Bibliography

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Homomorphic Encryption Standard*, pages 31–62. Springer International Publishing, Cham, 2021.

[2] Andreas M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, 2nd edition, June 2017.

[3] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[4] Hannu A Aronsson. Zero knowledge protocols and small systems. *Department of Computer Science, Helsinki University of Technology*, 1995.

[5] Thomas Bocek and Burkhard Stiller. Smart contracts–blockchains in the wings. In *Digital marketplaces unleashed*, pages 169–184. Springer, 2018.

[6] Umesh Bodkhe, Sudeep Tanwar, Karan Parekh, Pimal Khanpara, Sudhanshu Tyagi, Neeraj Kumar, and Mamoun Alazab. Blockchain for industry 4.0: A comprehensive review. *IEEE Access*, 8:79764–79800, 2020.

[7] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964, 2020.

[8] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443, 2020.

[9] Vitalik Buterin. On Public and Private Blockchains, August 2015. `https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/`, Last visit November 10, 2021.

[10] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200, 2019.

[11] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 185–200, 2019.

[12] CoinMarketCap. Today's Cryptocurrency Prices by Market Cap. `https://coinmarketcap.com/`, Last visit November 11, 2021.

[13] Sankarshan Damle, Sujit Gujar, and Moin Hussain Moti. Fasten: Fair and secure distributed voting using smart contracts, 2021.

[14] N1 Analytics developers. Basic JSON Serialisation. `https://python-paillier.readthedocs.io/en/stable/serialisation.html#basic-json-serialisation`, Last visit March 21, 2022.

[15] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *Service-Oriented and Cloud Computing*, pages 3–15, Cham, September 2017. Springer International Publishing.

[16] OpenJS Foundation. Downloads. `https://nodejs.org/en/download/`, Last visit March 25, 2022.

[17] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Off the chain transactions. *IACR Cryptol. ePrint Arch.*, 2019:360, 2019.

[18] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2):100179, 2021.

[19] ConsenSys Software Inc. Ganache - Overview. `https://trufflesuite.com/docs/ganache/`, Last visit March 25, 2022.

[20] Bhaskar Kashyap. Introduction to Smart Contracts. `https://ethereum.org/en/developers/docs/smart-contracts/`, Last visit November 11, 2021.

[21] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016.

[22] Chao Li, Balaji Palanisamy, and Runhua Xu. Scalable and privacy-preserving design of on/off-chain smart contracts. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 7–12, 2019.

[23] Chao Liu, Jianbo Gao, Yue Li, Huihui Wang, and Zhong Chen. Studying gas exceptions in blockchain-based cloud applications. *Journal of Cloud Computing*, 9, 06 2020.

[24] Martin McBride. Fernet system for symmetric encryption. `https://www.pythoninformer.com/python-libraries/cryptography/fernet/`, Last visit March 19, 2022.

[25] Greg Michaelson. Programming paradigms, turing completeness and computational thinking. *CoRR*, abs/2002.06178, 2020.

[26] Joshua minimalsm. Gas and Fees. `https://ethereum.org/en/developers/docs/gas/`, Last visit March 23, 2022.

[27] Joshua minimalsm. Introduction to Smart Contracts. `https://ethereum.org/en/developers/docs/smart-contracts/`, Last visit March 23, 2022.

[28] Mintlayer. Why DeFi's Future Is With Non-Turing-Complete Smart Contracts. `https://www.mintlayer.org/news/2020-11-05-why-defis-future-is-with-non-turing-complete-smart-contracts/`, Last visit November 11, 2021.

[29] Debajani Mohanty. Ethereum for architects and developers. *Apress Media LLC, California*, pages 14–15, 2018.

[30] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[31] Qian Ren, Han Liu, Yue Li, and Hong Lei. Cloak: A framework for development of confidential blockchain smart contracts, 2021.

[32] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[33] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 26(1):96–99, 1983.

[34] David Cerezo Sánchez. Raziel: Private and verifiable smart contracts on blockchains, 2020.

[35] Eder J Scheid, Bruno B Rodrigues, Christian Killer, Muriel F Franco, Sina Rafati, and Burkhard Stiller. Blockchains and distributed ledgers uncovered: Clarifications, achievements, and open issues. In *Advancing Research in Information and Communication Technology*, pages 289–317. Springer, 2021.

[36] Ravital Solomon and Ghada Almashaqbeh. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2021:133, 2021.

[37] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1759–1776, 2019.

[38] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, 1st edition, February 2015.

[39] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

[40] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers, 2021.

[41] David Wong. *Real-World Cryptography.* Manning Publications, 2021.

[42] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, Last visit November 22, 2021.

[43] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
EVM	Ethereum Virtual Machine
FHE	Full Homomorphic Encryption
JSON	JavaScript Object Notation
MPT	Multi-Party Transactions
NIZK	Non-Interactive Zero Knowledge Proof
P2P	Peer-to-Peer
PoS	Proof-of-Stake
PoW	Proof-of-Work
SC	Smart Contract
TEE	Trusted Execution Environments
TTP	Trusted Third Party
ZKP	Zero Knowledge Proof

List of Figures

List of Tables

Appendix A

Smart Contract Unencrypted

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract unenc {
6
7      // Struct needed to store contracting party-related data
8      struct ContractingParty{
9          address addr;
10         string name;
11     }
12
13     ContractingParty buyer;
14     ContractingParty seller;
15     string desc; // Description of good to be subject of the SC
16     uint256 amount; // The price the buying party has to pay to the
           selling party
17
18     // Getter and Setter-functions for storing and fetching the
           respective data of the SC
19
20     // @param address is the seller's address
21     // @param name is the seller's name
22     function setSeller(address addr, string memory name) public{
23         seller = ContractingParty(addr, name);
24     }
25
26     function getSeller() public view returns (ContractingParty memory){
27         return seller;
28     }
29
30     // @param address is the buyer's address
31     // @param name is the buyer's name
32     function setBuyer(address addr, string memory name) public{
33         buyer = ContractingParty(addr, name);
34     }
35
36     function getBuyer() public view returns (ContractingParty memory){
37         return buyer;
```

```
38      }
39
40      // @param gdDesc is the description of the good being subject of the
            transaction
41      function setDesc(string memory gdDesc) public{
42          desc = gdDesc;
43      }
44
45      function getDesc() public view returns (string memory){
46          return desc;
47      }
48
49      // @param amt is the price amount
50      function setAmount(uint256 amt) public{
51          amount = amt;
52      }
53
54      function getAmount() public view returns (uint256){
55          return amount;
56      }
57
58  }
```

Listing A.1: SC Used for the Implementation of the Unencrypted Approach

Appendix B

Smart Contract Symmetric Encryption

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract syenc {
6
7      // Struct needed to store contracting party-related data
8      struct ContractingParty{
9          address addr;
10         bytes name;
11     }
12
13     ContractingParty buyer;
14     ContractingParty seller;
15     bytes desc; // Description of good to be subject of the SC
16     bytes amount; // The price the buying party has to pay to the
           selling party
17
18     // Getter and Setter-functions for storing and fetching the
           respective data of the SC
19
20     // @param address is the seller's address
21     // @param name is the seller's name
22     function setSeller(address addr, bytes memory name) public{
23         seller = ContractingParty(addr, name);
24     }
25
26     function getSeller() public view returns (ContractingParty memory){
27         return seller;
28     }
29
30     // @param address is the buyer's address
31     // @param name is the buyer's name
32     function setBuyer(address addr, bytes memory name) public{
33         buyer = ContractingParty(addr, name);
34     }
35
36     function getBuyer() public view returns (ContractingParty memory){
37         return buyer;
```

```
38      }
39
40      // @param gdDesc is the description of the good being subject of the
            transaction
41      function setDesc(bytes memory gdDesc) public{
42          desc = gdDesc;
43      }
44
45      function getDesc() public view returns (bytes memory){
46          return desc;
47      }
48
49      // @param amt is the price amount
50      function setAmount(bytes memory amt) public{
51          amount = amt;
52      }
53
54      function getAmount() public view returns (bytes memory){
55          return amount;
56      }
57
58  }
```

Listing B.1: SC Used for the Implementation of the Symmetric Approach

Appendix C

Smart Contract Asymmetric Encryption

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract asyenc {
6
7      // Struct needed to store contracting party-related data
8      struct ContractingParty{
9          address addr;
10         bytes name;
11         string pubKey;
12     }
13
14     ContractingParty buyer;
15     ContractingParty seller;
16     bytes desc; // Description of good to be subject of the SC
17     bytes amount; // The price the buying party has to pay to the
           selling party
18
19     // Getter and Setter-functions for storing and fetching the
           respective data of the SC
20
21     // @param address is the seller's address
22     // @param name is the seller's name
23     // @param PubKey is the seller's public key
24     function setSeller(address addr, bytes memory name, string memory
           pubKey) public{
25         seller = ContractingParty(addr, name, pubKey);
26     }
27
28     function getSeller() public view returns (ContractingParty memory){
29         return seller;
30     }
31
32     // @param address is the buyer's address
33     // @param name is the buyer's name
34     // @param PubKey is the buyer's public key
35     function setBuyer(address addr, bytes memory name, string memory
           pubKey) public{
```

```
36          buyer = ContractingParty(addr, name, pubKey);
37      }
38
39      function getBuyer() public view returns (ContractingParty memory){
40          return buyer;
41      }
42
43      // @param gdDesc is the description of the good being subject of the
            transaction
44      function setDesc(bytes memory gdDesc) public{
45          desc = gdDesc;
46      }
47
48      function getDesc() public view returns (bytes memory){
49          return desc;
50      }
51
52      // @param amt is the price amount
53      function setAmount(bytes memory amt) public{
54          amount = amt;
55      }
56
57      function getAmount() public view returns (bytes memory){
58          return amount;
59      }
60  }
```

Listing C.1: SC Used for the Implementation of the Asymmetric Approach

Appendix D

Smart Contract Homomorphic Encryption

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract homenc {
6
7      // Struct needed to store contracting party-related data
8      struct ContractingParty{
9          address addr;
10         bytes name;
11         string pubKeyAsym;
12         string pubKeyHE;
13     }
14
15     ContractingParty buyer;
16     ContractingParty seller;
17     bytes desc; // Description of good to be subject of the SC
18     string amount; // The hashed and encrypted price the buying party
           has to pay to the selling party
19
20     // Getter and Setter-functions for storing and fetching the
           respective data of the SC
21
22     // @param address is the seller's address
23     // @param name is the seller's name
24     // @param PubKey is the seller's public key
25     function setSeller(address addr, bytes memory name, string memory
           pubKeyAsym, string memory pubKeyHE) public{
26         seller = ContractingParty(addr, name, pubKeyAsym, pubKeyHE);
27     }
28
29     function getSeller() public view returns (ContractingParty memory){
30         return seller;
31     }
32
33     // @param address is the buyer's address
34     // @param name is the buyer's name
```

```
35      // @param PubKey is the buyer's public key
36      function setBuyer(address addr, bytes memory name, string memory
            pubKeyAsym, string memory pubKeyHE) public{
37          buyer = ContractingParty(addr, name, pubKeyAsym, pubKeyHE);
38      }
39
40      function getBuyer() public view returns (ContractingParty memory){
41          return buyer;
42      }
43
44      // @param gdDesc is the description of the good being subject of the
            transaction
45      function setDesc(bytes memory gdDesc) public{
46          desc = gdDesc;
47      }
48
49      function getDesc() public view returns (bytes memory){
50          return desc;
51      }
52
53      // @param amt is the price amount
54      function setAmount(string memory amt) public{
55          amount = amt;
56      }
57
58      function getAmount() public view returns (string memory){
59          return amount;
60      }
61  }
```

Listing D.1: SC Used for the Implementation of Asymmetric Encryption

Appendix E

Installation Guidelines

The installation guidelines for Ganache can be found on the Truffle website [19].

E.1 Setup

This project used Python 3.10.2, on Windows 11 21H2. The Ganache version used was v2.5.4.

E.1.1 Install Ganache

Download Ganache from the Truffle website [19] and install the downloaded `.appx`-file.

E.1.2 Truffle npm-package

If `node.js` is not installed, download it on the website [16]. Then install the `truffle` package by executing the command `npm install -g truffle` in the system's terminal with the required authorization rights.

E.1.3 Install pip

If pip is not installed (check by executing `pip help` in the system's terminal, if an error is retrieved, it is not installed), then download the `get-pip.py`-file, which then can be run after navigating to the directory of the file in the terminal with the command `py get-pip.py`.

E.1.4 Install required Python modules

Use the following commands to install the used packages:

- `pip install web3`
- `pip install fernet`
- `pip install rsa`
- `pip install phe`

In case of any failure due to a missing module, the command `pip install <pkg-name>` can be used, replacing the placeholder `<pkg-name>` with the required package.

E.1.5 Establishing a Truffle project

Either create a new Truffle project by executing the command `truffle init` at the desired project location or use the provided projects. Root folders of these projects are always containing the file `truffle-config.js`.

Important Note: In the file `2_deploy_contracts.js`, located in the project's directory `./migrations`, the `string`-parameter given to the `require()`-function has to match the name of the contract defined in the Solidity implementation of the SC as seen in Listings E.1 and E.2

```
1 var homenc = artifacts.require("homenc");
2 ...
```

Listing E.1: Input for the require()-function to deploy it

```
1 // SPDX-License-Identifier: MIT
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 contract homenc {
6 ...
```

Listing E.2: Corresponding name of SC in Solidity

E.1.6 Set up blockchain

Open the Ganache GUI and click on either "Quickstart" or on "New Workspace" if additional parameterization is needed.

Choose the desired `truffle-config.js` from the project's location to establish the connection.

E.1.7 Deploying contracts

After putting the SC in the project's contract-folder `./contracts` and adjusting the `2_deploy_contracts.js` accordingly, the SC have to be deployed to the blockchain by executing the command `truffle migrate` through the terminal opened in the project's directory.

E.2 Connection between script and blockchain

In order to connect the script to the Ganache GUI, check the `Network ID` and `RPC Server` for the IP-address and the corresponding port, which has to be set in the `truffle-config.js` as shown in Listing E.3

```
networks: {
    development: {
     host: "127.0.0.1",     // Localhost (default: none)
     port: 7545,            // Standard Ethereum port (default: none)
     network_id: "*",       // Any network (default: none)
     networkCheckTimeout:300
    },
```

Listing E.3: Config of the connection to Ganache

These values have to be set as well in the Python script using the `web3` package, to connect the script with the established blockchain.

```
...
# truffle development blockchain address
blockchain_address = 'http://127.0.0.1:7545'
# Client instance to interact with the blockchain
web3 = Web3(HTTPProvider(blockchain_address,request_kwargs={'timeout'
    :3600}))
# Set the default account (so we don't need to set the "from" for every
    transaction call)
web3.eth.default_account = web3.eth.accounts[0]

# Path to the compiled contract JSON file
compiled_contract_path = 'build/contracts/homenc.json'

with open(compiled_contract_path) as file:
    contract_json = json.load(file)  # load contract info as JSON
    contract_abi = contract_json['abi']  # fetch contract's abi -
        necessary to call its functions
    deployed_contract_address = contract_json['networks']['5777']['
        address'] # Deployed contract address (see `migrate` command
        output: `contract address`)
```

Listing E.4: Using web3 to connect the script to the blockchain

Appendix F

Contents of the CD

The data was provided as a .zip-file instead of a pyhsical CD as agreed upon with the supervisor during the meetings in course of this thesis.

The content is as follows:

- `thesis.zip` containing the files with the LaTeXsource code.
- `BA_Raphael_Imfeld_Encryption_SC_Final.pdf`, the final version of the thesis exported as .pdf.
- `SC_enc.zip` containing the source code of the implemented scenarios together with the test suites and performance-check scripts.
- `midterm.pptx`, the slides for the midterm presentation held on January 10, 2022.

The slides for the final presentation are not included at this stage as the presentation is scheduled for April 7, 2022 and will be provided after this date.