PUBLICLY VERIFIABLE AUTHENTICITY OF DATA FROM MULTIPLE EXTERNAL SOURCES FOR SMART CONTRACTS USING AGGREGATE SIGNATURES

BJORN VAN DER LAAN

to obtain the degree of Master of Science in Computer Science
Data Science & Technology Track
to be defended publicly on July 12, 2018

DELFT UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering, Mathematics & Computer Science
Department of Intelligent Systems
Cyber Security Group

STUDENT NUMBER:
4151348

THESIS COMMITTEE:
Assoc.Prof.Dr.Ir. J.C.A. van der Lubbe
Assist.Prof.Dr. J.C. van Gemert
Assist.Prof.Dr. Z. Erkin
H.F. Spenkelink, MSc
O. Ersoy, MSc

SUPERVISORS:
Assist.Prof.Dr. Z. Erkin
H.F. Spenkelink, MSc
O. Ersoy, MSc

AN ELECTRONIC VERSION OF THIS THESIS IS AVAILABLE AT:
http://repository.tudelft.nl/

ABSTRACT

Smart contracts are applications that are deployed and executed on a blockchain's decentralised infrastructure. Many smart contract applications rely on data that resides outside the blockchain. However, while traditional web applications can communicate with trustworthy data sources directly through the Internet, this is not possible for smart contracts because their execution must be deterministic. Bringing external data into the blockchain has been a topic of research since the first introduction of Ethereum, and a system that can provide this data to smart contracts is called an oracle. The primary requirement in designing oracles is that the authenticity of the data must be publicly verifiable, which can be achieved through signatures. However, transmitting data to the blockchain and performing the verification is costly, especially if applications require data from multiple sources as, in that case, current approaches would need to retrieve the data from each source separately.

This research aims to reduce the cost of retrieving external data for smart contracts from multiple sources while ensuring that the authenticity of the data is publicly verifiable. Two factors influence the total cost. The first is the size of the data, which determines the cost of transmitting the data to the blockchain and storing it, while the second factor is the cost of verifying the authenticity. In this work, we focused on the first factor, as transmission and storage of data are among Ethereum's most expensive operations.

We present two oracles for retrieving data from multiple sources, which we believe to be the first to focus on the multi-source scenario. The oracles both lower the cost of retrieving external data by compressing the proofs of the data's authenticity using aggregate signatures. Even though the oracles achieve the same goal, they are based on different primitives. The first uses bilinear pairings and produces an aggregate signature of constant size, regardless of the number of data sources that are involved. The second is based on the more standard assumption of trapdoor permutations. However, the aggregate signature grows slightly with the number of signers, and the oracle must interact with the data sources sequentially. We confirm the feasibility of our work by implementing and practically evaluating the two oracles in the Solidity programming language. Our experiments show that both oracles expend less gas than non-aggregating oracles based on the same main primitives.

PREFACE

In front of you lies the result of a challenging and fruitful period that marks the end of my time as a student. The report has been written to fulfil the graduation requirements of the master Computer Science at the Delft University of Technology and has kept me engaged from November 2017 until July 2018.

Some words to express my gratitude are appropriate here, first of all for Zeki and Oğuzhan. Thank you for always being available for questions, coming up with a lot of ideas to improve my thesis, and allowing me to be part of your community. Also, thank you to the rest of the NerdGroup. Whether we were discussing our work or eating cake, we always had a great time.

I would also like to take some time to thank Hardwin, my supervisor from KPMG. Because of your background, you were able to view my work from a different perspective. As such, your feedback and comments formed a valuable addition to my research. Also, thank you to Dennis for trusting me with an assignment that some junior consultants can only dream of. Furthermore, thank you to the whole Innovation Advisory team for letting me be part of your team. You all showed great interest in my thesis, even though you did not always understand exactly what I was doing. Special thanks to Sophie, who often accompanied me on the long journey from Amstelveen back to Den Haag.

Thanks as well to the two other members of my committee, Jan van der Lubbe and Jan van Gemert for taking the time to read my report and attend the presentation.

I also want to thank all friends I made during my time at university. Special thanks to club Mach, de Westerstraat, and StuD. Being a student is more than just studying.

My last words of thanks are to my family, who made this all possible by supporting me both mentally and financially, and to Iris, who has been amazingly supportive and made sure I also took time for relaxation and fun.

Bjorn van der Laan
Delft, July, 2018

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

Smart contracts are applications that are deployed and executed on a blockchain's decentralised infrastructure. They are immutable and tamper-proof in the sense that no party, not even their creator, can alter their code or interfere with their execution without the consent of all the nodes in the network. Smart contracts can formalize complex procedures. Szabo [29], who introduced smart contracts, gave as an example a smart contract that enforces a car lock to let in only the owner and exclude third parties. However, if the car is used as collateral and the owner fails to make a timely payment, the smart contract returns control of the car to the bank.

1.1 EXTERNAL DATA RETRIEVAL

Similar to the car lock example, many smart contract applications rely on data that resides outside the blockchain. While traditional web applications can communicate directly with trustworthy data sources through the Internet, this is not possible for smart contracts because their execution must be deterministic. The need for determinism exists because a blockchain is replicated among all nodes of the network. Any time a node starts at the start state, called the genesis block, and applies the transactions of each block consecutively, it should always arrive at the same result, as disagreement among nodes about the blockchain's state could cause the network to split. As a consequence of determinism, smart contracts cannot make requests to the world external to the blockchain, as this introduces non-determinism.

Bringing external data into the blockchain has been a topic of research since Buterin [7] first introduced Ethereum, and a system that can provide this data to smart contracts is called an *oracle*. We refer to anything within the blockchain environment as being *on-chain*, while anything that exists outside the blockchain is *off-chain*. For example, smart contracts are stored on-chain, and web servers are found off-chain on the Web. Following this terminology, the goal of oracles is to allow on-chain entities to access information stored by off-chain entities. In this context, we can distinguish three types of actors. The first two are the requesters, which are smart contracts that live on-chain and want access to external data, and the data sources, that provide the external data that the contracts want and generally already publicly offer this data to web applications. The third actor is the oracle, a system that bridges the gap between the on- and off-chain world. Oracles are generally a combination of two components, of which the

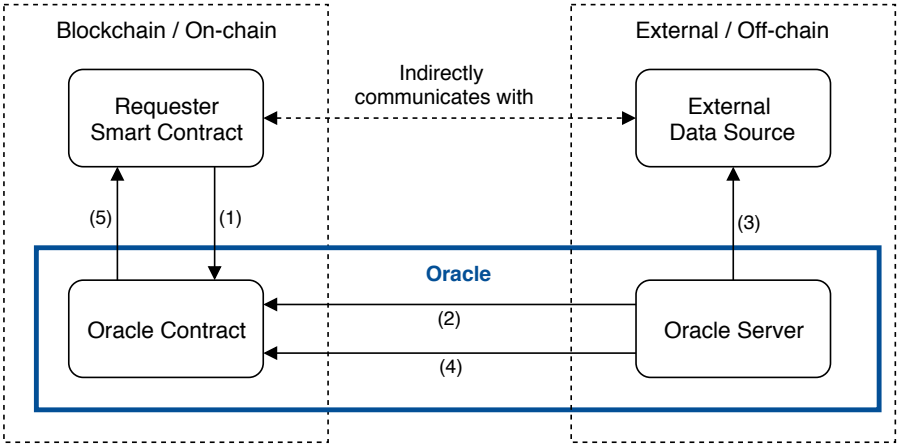


Figure 1.1: Data flows in external data retrieval.

first is a smart contract acting as an interface towards other smart contracts. The second component is an external server that retrieves the requested data and delivers it to the blockchain. How to construct an oracle can be explained by looking at the information flows occurring between the moments where certain data is requested and received:

1. The requester sends a request to the oracle contract. The request specifies the requested data. The transaction containing the request is recorded into the blockchain.
2. The oracle server continuously observes the blockchain and at some point observes the new data request.
3. Upon detection, the oracle server retrieves the requested data from a data source.
4. The oracle server sends the information back to the oracle contract.
5. The requester smart receives the information from the oracle contract.

The concept is visualised in Figure 1.1, where the numbers correspond to the list above. Observe that the arrows between the on-chain and off-chain environment are one-directional. As previously mentioned, smart contracts on the blockchain cannot directly communicate with external sources due to the need for determinism. However, off-chain entities can observe the contents of public blockchains. In other words, the oracle server can inspect the blockchain, but oracle contract cannot examine entities on the Internet.

PREVIOUS WORK ON ORACLES The main requirement in designing oracles is that the authenticity of the data must be publicly verifiable. The oracle described in Figure 1.1 does not meet this requirement as there is no way to verify the authenticity. Several approaches

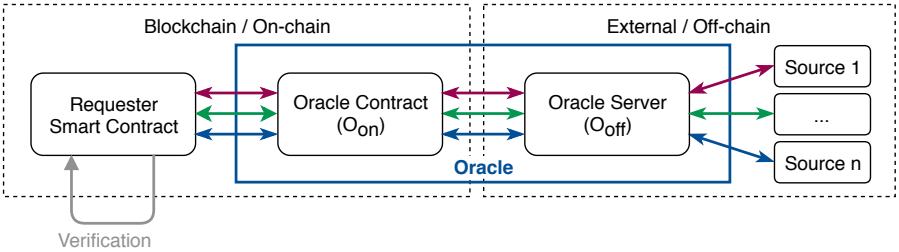


Figure 1.2: The requester lets the oracles retrieve the data from each source separately and then verifies the proofs.

to creating more secure oracles have been proposed, which all fall into one of three categories. The first category ensures authenticity by making changes to the data source, while oracles that fall into the second category propose to use trusted hardware to prove that particular code was executed. In the literature review that we conducted as part of this thesis, we could only find a single example of both categories in the form of, respectively, TLS-N [24] and Town Crier [32]. The third category of oracles are structured as a decentralized network [11, 22], which are built in such a way that nodes profit most if they act faithfully.

Besides authenticity of the retrieved data, Zhang et al. [32] introduced two security properties that apply to any service that, such as oracles, comprises both on-chain and off-chain components. We refer to such services as *hybrid services*. The first property is *gas sustainability*, which ensures that a malicious requester is unable to deplete the service's resources. The second property is *fair expenditure for an honest requester*. Informally, the requester can be sure that it pays at most a certain amount for using the service, even if the service itself is malicious. A more detailed discussion of these properties is found in Chapter 4.

1.2 RETRIEVING DATA FROM MULTIPLE SOURCES

Oracles in previous work are designed to retrieve data from a single source. However, there are many situations where the smart contract's execution depends on data from multiple sources. Consider, for instance, a situation where the smart contract pays out a certain amount if the temperature in a specific area is above a certain threshold. Multiple sensors are installed at different locations in the area to measure the temperature. In this scenario, the smart contract could let an oracle retrieve the data from each source separately, after which the smart contract can process it. This approach is illustrated in Figure 1.2.

However, in Ethereum, users pay a certain amount to perform a transaction. This amount consists of two parts, namely a constant base cost and a variable cost per byte [31]. Using the approach for

handling multiple data sources described above, the number of transactions performed depends linearly on the number of data sources as each request is handled separately. A trivial solution to reduce the number of transactions to a single one is by constructing an oracle that collects the messages and proofs from all data sources and puts it into a single transaction. Yet, the total amount of data to be transmitted remains the same, meaning that we only achieve savings in the constant component of the transaction cost. If we also want to lower the variable costs, we need to compress the data itself, which includes messages and signatures. Lowering the variable cost has the additional advantage that we also need to store less. According to Wood [31], storing data is among the most expensive operations in Ethereum.

1.3 RESEARCH GOAL

Previous work on oracles has mainly been focused on ensuring the authenticity of external data retrieved from a single source but has thus far not explored approaches for scenarios in which data comes from multiple sources. Using the current techniques, the data would be retrieved from each source separately, after which it is stored until the data from other data sources is retrieved as well.

The goal of this research is to develop a more efficient approach for dealing with multiple external data sources, which reduces both the number of transactions and the total size of the data transferred to the blockchain. More specifically, our goal is to lower the total costs, which comprises both the costs for sending data on-chain and for performing the verification. The main requirement is that the approach provides public verifiability of the data's authenticity. Also, data sources should only have to sign their data and not perform any additional computational steps.

The underlying research question of this thesis is as follows:

How can we reduce the cost of retrieving external data for smart contracts from multiple sources, while ensuring that the authenticity of the data is publicly verifiable?

This research question raises the following sub-questions:

1. How can we provide external data to smart contracts in a way that the authenticity can be publicly verified?
2. How can we design oracles that guarantee gas sustainability and fair expenditure?
3. How can we reduce the cost of transmitting data to the blockchain and verifying its authenticity in a multi-source setting?

1.4 CONTRIBUTIONS

In this research, we first present CHAINBRIDGE, a framework for creating hybrid services, in Chapter 4. CHAINBRIDGE is a generalization of Town Crier [32], an oracle based on trusted hardware. Similar to Town Crier, CHAINBRIDGE-based services guarantee gas sustainability and fair gas expenditure for honest requesters. While CHAINBRIDGE is used to build the other contributions, these two properties apply to any hybrid service. As such, CHAINBRIDGE is designed as a generic framework and presented as a separate contribution that is of independent interest.

Subsequently, we present two oracles for retrieving data from multiple sources, which we believe to be the first to focus on the multi-source scenario. The two oracles both lower the cost of retrieving external data by compressing the proofs of the data's authenticity using aggregate signatures. Even though the oracles achieve the same goal, they are based on different primitives:

- MUSCLE-BP, presented in Chapter 6, is based on bilinear pairings. By aggregating the signatures, we reduce the total size to a constant amount, regardless of how many data sources are involved. On the other hand, to verify the authenticity of the data, one has to compute a number of bilinear pairings linearly related to the number of data sources, which is regarded to be computationally expensive.

- MUSCLE-TP, presented in Chapter 7, is based on the more standard assumption of trapdoor permutations [10]. Similar to MUSCLE-BP, the total size of the proofs is reduced via aggregation. However, the signature is not of constant size but grows with the number of signers. Furthermore, MUSCLE-TP must interact with the data sources sequentially, i.e., the data sources are contacted in-order, and each data source adds its signature on the aggregate-so-far.

Based on proof-of-concept implementations of both presented oracles, we show that a decrease in costs in a multi-source scenario can be obtained from using aggregate signature schemes in comparison with both the current state of the art [24] and oracles that are based on the same primitives, but do not employ signature aggregation.

Even though the design of CHAINBRIDGE, MUSCLE-BP, and MUSCLE-TP applies to any blockchain in which smart contracts cannot access external sources directly due to determinism constraints and where nodes execute smart contract code after consensus, we use Ethereum [7, 31] as our application setting. This is because all current oracles are designed for Ethereum, which allows us to evaluate MUSCLE-BP and MUSCLE-TP with existing oracles in terms of costs and performance.

1.5 THESIS OUTLINE

The structure of this thesis is as follows. In Chapter 2, the preliminaries used in this research are discussed. Chapter 3 reviews prior art on oracles. After that, we propose CHAINBRIDGE, a framework for hybrid services on Ethereum, in Chapter 4. Then, we specify the system model, assumptions, and requirements in Chapter 5 and, based on this, create MUSCLE-BP and MUSCLE-TP, which are presented in Chapter 6 and Chapter 7, respectively. Finally, we evaluate their performance in Chapter 8 and discuss the obtained results and provide an outlook for future research opportunities in Chapter 9.

2

PRELIMINARIES

In this chapter, we introduce the preliminary knowledge and techniques used in this research. First, we discuss the cryptographic primitives and protocols in Section 2.1, then we introduce blockchain technology in Section 2.2.

2.1 CRYPTOGRAPHIC PRIMITIVES

In this section, we first give a definition of cryptographic hash functions. Then, we elaborate on digital signatures and aggregate signatures, after which we introduce bilinear pairings and trapdoor permutations.

2.1.1 *Cryptographic hash functions*

A hash function is a function that maps input data of arbitrary size to output data of fixed size. This output is called a hash value, digest, or hash. Hash functions in general are used for a variety of applications, such as indexing in hash tables and detection of corrupted data using checksums. A special class of hash functions, called cryptographic hash functions, provides three additional properties [20]:

- Pre-image resistance: given hash function h and hash value y, it is computationally infeasible to find any value x such that $y = h(x)$.
- Second pre-image resistance: given hash function h and value x, it is computationally infeasible to find a value x' such that $h(x) = h(x')$ and $x \neq x'$.
- Collision resistance: it is computationally infeasible to find two values x, x', such that $h(x) = h(x')$ and $x \neq x'$.

Cryptographic hash functions have many applications in information security, such as in digital signatures (see Section 2.1.2).

2.1.2 *Digital signatures*

A digital signature scheme is a cryptographic scheme that is used to prove the authenticity of messages. A valid digital signature ensures the message's recipient of three properties:

- Authentication: the message was created by a known sender.

- Non-repudiation: the sender cannot deny having sent the message.
- Integrity: the message was not altered in transit.

In such schemes, each user has two keys. The first key is called the private key. This key should be kept secret and is used to sign messages. Other users can verify the message's authenticity using the signer's second key, called the public key. The principle of digital signatures is that anyone can verify a signature, while only the owner of the private key can create signatures.

RSA SIGNATURES As an example of digital signature schemes, we examine a scheme that uses RSA (Rivest-Shamir-Adleman) [25]. The security of RSA is based on the difficulty of the factoring problem. A public and private key pair in RSA can be generated using the algorithm by first choosing two random prime numbers p and q of approximately equal bit length and compute $N = p \cdot q$, which is called the modulus. Then, a random integer e is selected such that $e < (p-1)(q-1)$. The pair (N, e) forms the public key. Finally, the integer d, acting as the private key, is computed such that $e \cdot d \equiv 1 \mod (p-1)(q-1)$. Using the generated key pair, a signature s of message m, is defined as $s = m^d \mod N$. The signature can be verified by computing $m' = s^e \mod N$ and checking that m and m' are equal.

2.1.3 *Aggregate signatures*

An aggregate signature scheme is a digital signature scheme that allows the aggregation of n signatures of n distinct messages from n different users into one single signature. The resulting aggregate signature can convince a verifier that the n users did indeed sign the n messages. Aggregate signature schemes can be used to provide authentication in bandwidth-limited environments and were first introduced by Boneh et al. [5], who also presented the first aggregate signature scheme based on the BLS signature scheme [4]. In this scheme, which we refer to as BGLS, all n messages must be distinct, i.e., no two sources can pick the same message. However, the authors note that this requirement can be worked around by letting signers prepend their public key to every message they sign before hashing and signing it. As each public key is unique, the resulting concatenation is also unique.

Subsequent aggregate signature schemes are generally based on either bilinear pairings or trapdoor permutations. The use of bilinear pairings often leads to aggregate signatures of smaller size, while schemes based on trapdoor permutations can generally be verified

more efficiently. We distinguish three categories, which are introduced in the remainder of this section.

SYNCHRONIZED AGGREGATE SIGNATURES A synchronised aggregate signature enables any user to combine different signatures with the same synchronising information into a single signature. However, as all signers must share the same synchronising information, such as a time clock, synchronised aggregate signature schemes are only applicable to scenarios where such information is available. Hohenberger and Waters [14] note that synchronised aggregate signature schemes might be interesting to use in combination with blockchains as new blocks are created at a roughly constant pace, which acts as a natural synchronisation event. The oracle could use the number of the block that the data request is stored in as the current time period t. However, a problem arises when a block contains two or more requests that include the same data source.

IDENTITY-BASED AGGREGATE SIGNATURES For aggregate signature schemes to work, public keys must be bound with the identity of their owner. This can be achieved by having a public key infrastructure (PKI) in which parties, known as certificate authorities (CAs), issue digital certificates that prove the relationship between a public key and its owner's identity. However, an alternative is to base aggregate signatures identity-based cryptography [26], in which the user's public key is generated from unique, but publicly known, identity information. Such schemes are called identity-based aggregate signature schemes. An advantage of identity-based aggregate signatures is that the public keys of signers do not need to be retrieved to verify the signature since publicly available identity information replaces these. One such scheme is by Gentry and Ramzan [13]. Similar to BGLS, their scheme is proven secure in the random oracle model and by assuming the hardness of computational Diffie-Hellman over groups with bilinear maps. The scheme allows more efficient verification than BGLS, as verification requires only three pairing computations regardless of the number of signers, while BGLS requires a linear number of pairing computations. However, identity-based schemes must rely on a trusted third party, called the private key generator (PKG), which generates and issues the corresponding private keys to all users based on a master secret key.

SEQUENTIAL AGGREGATE SIGNATURES Sequential aggregate signatures, introduced by Lysyanskaya et al. [18], are a variant of aggregate signatures in which a signer i receives an aggregate signature signed by signers 1 to $i-1$, adds its own signature and passes the aggregate signature to signer $i+1$.

After the first sequential aggregate signature scheme by Lysyanskaya et al. [18], Boldyreva et al. [3] presented an interactive identity-based sequential aggregate signature scheme in the random oracle model, which was later proven insecure by Hwang et al. [15]. Lu et al. [17] presented the first sequential aggregate signature scheme outside the random oracle model based on the Waters signature scheme [30]. Its security is proven under the computational Diffie-Hellman assumption. Compared to BGLS, verification of a signature requires only a constant number of pairings rather than linear. However, the signatures and public keys in this scheme are longer.

Some sequential aggregate signature schemes provide lazy verification. In that case, a signer i can add its own signature without verifying the aggregate-so-far produced by signers 1 to $i-1$.

2.1.4 *Bilinear pairings*

A bilinear pairing, also referred to as bilinear mappings, is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that maps elements of two cryptographic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to a third group $\mathbb{G}_T$ with the following properties:

1. Bilinear: $e(u^a, v^b) = e(u, v)^{ab}$ for all $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}$.
2. Non-degenerate: $e(g_1, g_2) \neq 1$.

Bilinear pairings can be either symmetric or asymmetric [8]. A pairing is symmetric if $\mathbb{G}_1 = \mathbb{G}_2$. These pairings are also called type 1 pairings. If, on the other hand, $\mathbb{G}_1 \neq \mathbb{G}_2$, then the pairing is asymmetric. Within asymmetric pairings, we distinguish between type 2 and type 3 pairings, depending on, respectively, whether or not there exists an efficiently computable function ψ, called an isomorphism, that sets up a one-to-one correspondence between the elements of the groups in a way that respects the given group operations.

BLS SIGNATURE SCHEME The BLS (Boneh-Lynn-Shacham) signature scheme [4] is based on bilinear pairings and comprises three algorithms, *KeyGen*, *Sign*, and $Verify$. The scheme is defined by two groups $\mathbb{G}_1$ and $\mathbb{G}_2$, their respective generators g_1 and g_2, the computable isomorphism ψ from $\mathbb{G}_1$ to $\mathbb{G}_2$, and the bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, with $\mathbb{G}_T$ being the target group. Furthermore, the scheme uses a full-domain hash function $H : \{0,1\}^* \to \mathbb{G}_2$.

KeyGen selects a random integer $sk \in \mathbb{Z}_p$, which acts as the private key. The corresponding public key is $pk = g_1^{sk}$. Given secret key sk and a message $m \in \{0,1\}$, the *Sign* algorithm computes signature $\sigma = H(m)^{sk}$, which can be verified by comparing whether $e(g_1, \sigma)$ is equal to $e(pk, H(m))$.

BLS signatures are also referred to as short signatures because the approximately 160 bits long signatures are relatively short compared to for example RSA (2048 bits using a 2048-bit modulus) and DSA

(320 bits). Even so, Boneh et al. [4] note that BLS signatures provide a level of security similar to 320-bit DSA signatures.

2.1.5 *Trapdoor permutations*

A trapdoor permutation or trapdoor one-way function [10] is a bijective function π which is hard to invert unless some secret information, called the trapdoor, is known. A well-known trapdoor permutation is RSA [25]. For RSA public key (N, e) and private key d, $f(x) = x^e \mod N$ is a trapdoor permutation, because it is both a permutation, as domain and range are equal, and a one-way trapdoor function, because the function is easy to compute using the public key, but difficult to invert without knowing the private key.

2.2 BLOCKCHAIN TECHNOLOGY

Blockchain was originally introduced in 2008 as the underpinning for Bitcoin [21] by a person or group under the pseudonym Satoshi Nakamoto. Essentially, a blockchain is a database that is not centrally managed by a particular entity. Instead, every node of the blockchains network maintains its own copy of the whole database. As there is no central party, we say the blockchain is decentralised.

In the remainder of this section, we first discuss the fundamental building blocks within blockchain in Section 2.2.1, followed by an introduction to consensus mechanisms in Section 2.2.2. Then, different types and architectures are introduced in sections 2.2.3 and 2.2.4, respectively. Finally, we look at Ethereum, our application setting, in Section 2.2.5.

2.2.1 *Building blocks*

In this section, we discuss essential building blocks of blockchain technology based on the paper by Nakamoto [21].

TRANSACTIONS AND ADDRESSES Transactions are a transfer of value from one party to another that is recorded in the blockchain. Much like traditional transactions, such as wire transfers, each transaction involves a sender, a transaction message, and a recipient, and transfers the ownership of an asset from the sender to the recipient. In blockchain, this asset is usually a cryptocurrency. Blockchain technology makes use of digital signatures (see Section 2.1.2) to send and receive transactions. The sender can authorise the transfer of assets by digitally signing the transaction using its private key, while the recipient's public key acts as blockchain's equivalent of bank account numbers in wire transfers. In blockchain terminology, we refer to the public key as a user's address.

BLOCKS Blocks are data structures that contain transactions. New blocks of data are added continuously by nodes of the blockchains network. Each block contains a timestamp of when it was added, as well as the hash of the previous block so that they form a chain of blocks (see Figure 2.1), which explains the term blockchain. Existing blocks are preserved forever, and modification of a block is immediately noticed by other nodes, as the hash contained in the next block would become invalid.

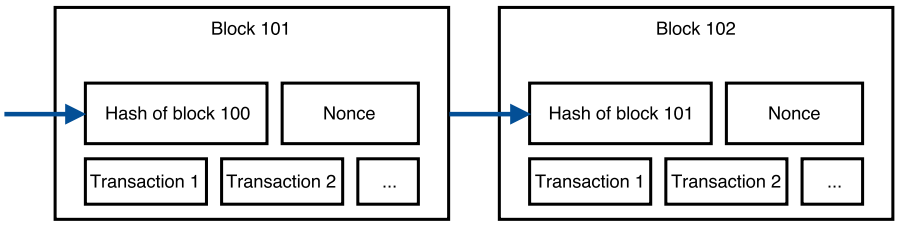


Figure 2.1: Blocks are chained together based on their hash.

MERKLE TREES Transactions within a block are stored in a hash tree, or Merkle tree, in which every leaf node contains a transaction and every non-leaf node contains the hash of its child nodes. The root of the tree, called the Merkle root, is stored in the header of the block. The integrity of the data is ensured as any modification to any of the transactions' contents or to the order of the transactions causes the Merkle root to change as well. Merkle trees also allow an efficient way to verify whether a specific transaction is included in the block or not. An example of a block containing a Merkle tree is shown in Figure 2.2.

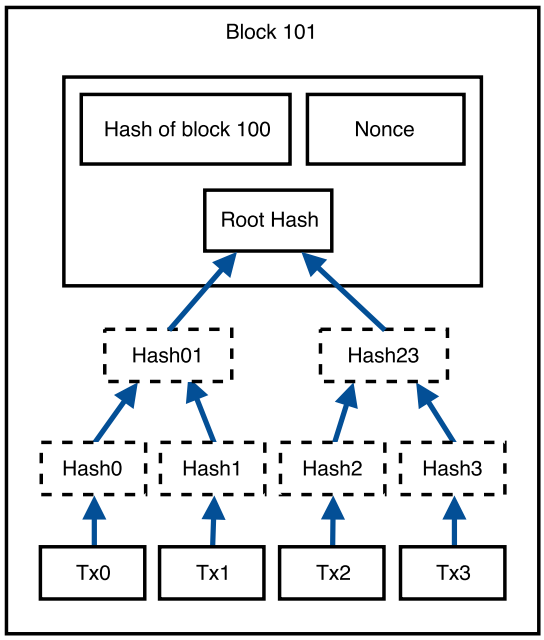


Figure 2.2: Transactions are stored in a Merkle Tree.

2.2.2 *Consensus mechanisms*

In the Byzantine Generals problem [16], several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. After observing the enemy, the generals must decide upon a plan of action, as the attack would fail if only part of the generals attacks the city. They do this by exchanging messages via messengers. However, there might be malicious actors among their ranks, who could send different decisions to different generals. The challenge is to reach consensus in an environment where no one can be trusted.

Nodes in a blockchain network face a similar challenge. As the network is distributed, no central party can determine which transactions should be included and in what order. The nodes do not trust other nodes, and so they need a mechanism to reach consensus, even in the presence of potential traitors. If a distributed system, such as a blockchain, can tolerate failures related to the Byzantine Generals Problem, we say it provides Byzantine Fault Tolerance (BFT). In case of blockchain, the mechanism to provide BFT is called a consensus mechanism. Below, we present the most well-known mechanism for consensus, called Proof-of-Work (PoW), as used in Bitcoin [21]. It works as follows:

1. Users sign new transactions using their private key and broadcast signed transactions to every node in the network.
2. Special nodes, called miners, collect these transactions and combine them into a block.
3. Miners must then solve a mathematical puzzle. In particular, the header of the block is hashed twice using the $SHA256$ hash function. If the resulting hash is lower than a certain threshold, called the difficulty, the puzzle is solved. In that case, the block is added, and the blockchain allows the miner to claim the block reward in the form of new bitcoins.

In theory, two nodes could add a new block at approximately the same time. Different nodes in the network then temporarily work on different versions of the blockchain. Such event is called a fork. However, it is unlikely that both forks add a second block at the same time again. Blockchains using Proof-of-Work need to have some mechanism in place to decide which branch becomes the legitimate blockchain. Blocks in the other branch are then called orphan blocks. The mechanism that nodes use to select which branch to follow differs per blockchain implementation. In Bitcoin, nodes follow the longest chain. An alternative selection rule, used by Ethereum [7, 31], is Greedy Heaviest-Observed Sub-Tree [27] (GHOST), which takes into account stale children of the block's ancestors in calculating which block has the most extensive total proof of work backing it.

2.2.3 *Two-dimensional classification*

There are various ways to categorise blockchains. In this work, we use the categorisation proposed by Peters and Panayi [23], which differentiates blockchains according to two dimensions:

- Public or Private: in a public blockchain, anyone can read the state and submit transactions to be included, while private blockchains restrict these rights to a set of nodes.
- Permissionless or Permissioned: in a permissionless blockchain, any node can participate in the verification process, while, in permissioned blockchains, a central authority authorises a certain set of verification nodes.

This categorization is visualized in Table 2.1.

Table 2.1: Different types of blockchains [23].

Access to transactions	Able to validate transactions	
	Permissionless	Permissioned
Public	All nodes can read and validate current transactions, as well as submit new ones	All nodes can read existing and submit new transactions, but only a preselected set of nodes can validate them.
Private	All nodes can validate transactions, but only a restricted set can read or submit them. Not yet seen in practice.	Only a certain set of nodes can read and validate existing transactions, as well as submit new ones.

In practice, only two types are regularly seen, as most permissionless blockchains feature public access, while most permissioned blockchains intend to restrict data access to the company or consortium of companies that operate the blockchain. In the remainder of this work, when we refer to permissionless and permissioned blockchains, we assume these combinations, unless explicitly stated.

2.2.4 *Blockchain architectures*

Blockchains that support smart contracts are generally based on an 'Order-Execute' architecture, where nodes in the blockchain's network first order the transactions, using a consensus protocol, and then execute them in the same order on all nodes sequentially as part of the verification [1]. As every node needs to execute smart contract

code and should arrive at the same result, the code must be completely deterministic. A consequence is that smart contracts cannot make requests to the world which is external to the blockchain, as this introduces non-determinism, which is the principal motivation for why we need oracles.

One might wonder if we can solve the oracle problem by introducing an architecture that allows non-determinism. Androulaki et al. [1] proposed such architecture, that can be referred to as 'Execute-Order-Validate'. However, while this architecture enables non-deterministic smart contract code, its applicability is limited to permissioned blockchains. Even among permissoned blockchains, only Hyperledger (starting at version 1.0) is currently using this architecture.

2.2.5 *Ethereum*

Already in 1996, Szabo [29] realised that enforceability of smart contracts required built-in incentives, self-enforcing protocols, and verifiability. He even already linked to cryptographic protocols, public key encryption, digital signatures, and zero-knowledge proofs.

In 2014, Buterin [7] used blockchain technology to implement smart contracts. His implementation is called Ethereum, and is often referred to as 'Bitcoin with a built-in Turing-complete programming language'. More technically, Ethereum is a protocol for executing a virtual computer in an open and distributed manner by reaching consensus among nodes. This virtual computer is called the Ethereum Virtual Machine (EVM), and its programs are called smart contracts. Buterin describes smart contracts as autonomous agents that reside on the blockchain, and execute a specific piece of code when triggered by a message or transaction. Smart contracts have direct control over their own ether balance and persistent key/value store. In the remainder of this section, we discuss some essential concepts of Ethereum, based mainly on the whitepaper by Buterin [7].

ETHEREUM ACCOUNTS The state of Ethereum comprises of a list of accounts, which are identified by an address. There are two kinds of accounts:

- Externally owned accounts or wallets. These accounts are owned and controlled by a user that has its private key. Wallets can send transactions to other accounts.
- Contract accounts or smart contracts. These accounts are controlled by their contract code. If a contract account receives a message, a part of its code is activated, which can contain operations for interacting with the contract's internal storage, performing certain computations, or sending messages to other contracts.

Each account has three attributes: a nonce, a balance in ether (Ethereum's cryptocurrency), and internal storage. Additionally, contract accounts have contract code.

ETHER AND GAS Ethereum has two native tokens: ether and gas. Ether is a currency. Just as bitcoins, ethers are minted by miners and traded on exchanges. Gas, on the other hand, is more similar to a commodity, like oil. Where oil is used to run your car, gas is used to run smart contract code. Each operation in a smart contract has a fixed cost. For instance, the addition of two variables costs 3 gas, multiplication costs 5 and computing a SHA3 hash costs 30 gas plus 6 gas for every 256 bits of input. If you want to activate a piece of smart contract code, you need to specify a gas price, which is the amount of ether you want to pay per gas. If the gas price you offer is too low, no one processes your transaction. The total transaction fee equals the total gas costs multiplied by the offered gas price.

Distinguishment between ether and gas allows the gas cost of an operation to be hard-coded into the Ethereum protocol. If only ether existed, the code must be updated every time ether's price fluctuates as to keep the price of computation in a healthy range and keep the system usable. By separating ether and gas, we decouple the price of ether from the price of using Ethereum. As a consequence, the costs listed in the Ethereum protocol can be static, and a user can determine per transaction how much ether a unit of gas is worth at that time.

TRANSACTIONS AND MESSAGES The way that accounts interact with each other depends on the type of account. Externally-owned accounts can initiate transactions to both externally-controlled and contract accounts. These transactions are signed by the sender's private key. A transaction contains:

- The recipient of the message.
- The amount of ether that is sent.
- A variable called *STARTGAS*, which represents the maximum amount of gas the user is willing to spend.
- A variable called *GASPRICE*, which is the amount of ether that the account is willing to pay per gas.
- An optional data field.
- The signature of the sender.

The recipient, amount of ether and the sender's signature are together comparable to what a transaction in Bitcoin entails. If the transaction is sent to a contract account (smart contract), the data field optionally contains the parameters of the invoked contract function. The STARTGAS and GASPRICE variables are used to pay for computation in

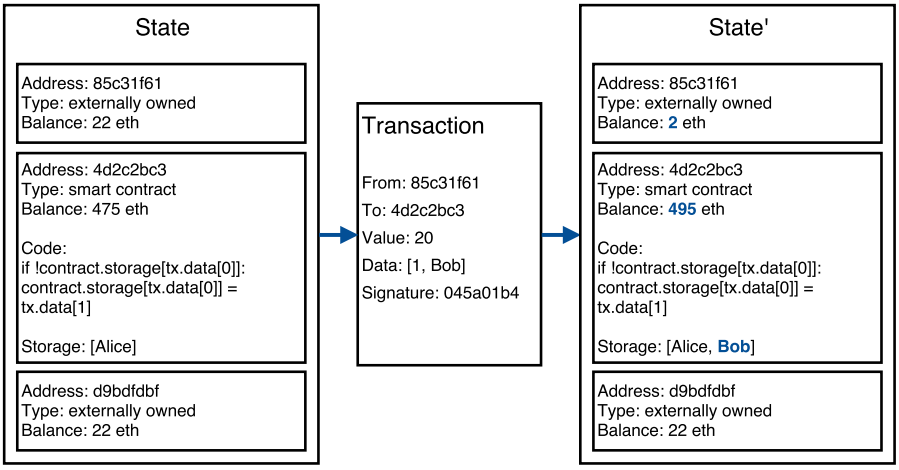


Figure 2.3: A state transition in Ethereum.

smart contracts activated by the transaction. As we have seen before, each computational operation costs a certain amount of gas. In other words, gas is a way to pay for the computational resources you use. The amount of gas paid to perform a transaction consists of two parts [31]. 21.000 gas is charged for any transaction as a base fee to cover the cost of an elliptic curve operation to recover the sender address from the signature. On top of that, 4 gas is paid per zero byte and 68 gas per non-zero byte for additionally included data.

Contract accounts, on the other hand, only communicate in response to other transactions that they have received. The response sent by a contract is called a message and contains the attributes mentioned above less the STARTGAS variable and the sender's signature. Messages are typically used by smart contracts to either call another smart contract as part of its operation or return data to an externally-owned account.

EVENTS Smart contracts in Ethereum can emit events, which are then stored in the blockchain together with the transaction that triggered the event. Events have a specific name and may contain data. Once the transaction is mined, the event data becomes available to read. Clients on the Web can listen to these events, which allows actions to be taken by clients when events are created. As such, events are a means of communication between smart contracts and clients on the Web.

STATE TRANSITION FUNCTION As previously mentioned, the state of Ethereum is equal to the state of all accounts. In that sense, transactions can be seen as state transitions as they make changes to account states. Formally, Ethereum's state transition function is defined as $APPLY(S, TX) \rightarrow S'$, where S is the current state, TX is a transac-

Table 2.2: The precompiled contracts used in this work.

Operation	Address	Gas cost	Description
EXPMOD	0x05	$\left\lfloor \frac{f(\max l_M, l_B) \cdot \max(l'_E, 1)}{G_{QUADDIVISOR}} \right\rfloor$	Big integer modular exponentiation.
ECADD	0x06	500	Elliptic curve addition.
ECMUL	0x07	40.000	Elliptic curve scalar multiplication.
PAIRING	0x08	$100.000 + 80.000 \cdot k$	Optimal ate pairing check.

tion, and S' is the resulting state. Broadly, the state transition function contains the following steps:

1. Check whether TX is well-formed and the signature is valid. If not, return an error.
2. Compute the transaction fee that the sender offers to process the transaction as STARTGAS $\times$ GASPRICE.
3. Determine the sender's address from the signature. Subtract the fee from the sender's account balance. If the sender's balance is insufficient, return an error.
4. Initialize GAS = STARTGAS and subtract a certain quantity of gas per byte to pay for the bytes in the transaction.
5. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a smart contract, run the contract's code either to completion or until the execution runs out of gas.
6. If the sender did not have enough ether or the code execution ran out of gas, then:
   - Revert all state changes except the payment of the fees and add the fees to the miner's account.
   - Else, refund the fees for all remaining gas to the sender and send the fees paid for gas consumed to the miner.

   An example of a transaction from an externally owned account to a smart contract is depicted in Figure 2.3.

PRECOMPILED CONTRACTS Some calculation-intensive mathematical and cryptographic functions are useful in many smart contracts.

These functions could be implemented in a high-level language, such as Solidity, but the execution would cost a lot of gas. Therefore, in Ethereum, these functions have been implemented as a precompiled contract, which essentially creates a dedicated opcode for these operations. Precompiled contracts require less gas, as the code is not run on the Ethereum Virtual Machine (EVM), but instead on the machine hosting the Ethereum client. A precompiled contract is assigned a fixed address and gas price and can be invoked using the *call* operation. We introduce the precompiled contracts used in this research in Table 2.2. For the pairing check, k denotes the number of points or, equivalently, the length of the input divided by 192. The variables for computing the gas cost of EXPMOD are further specified in EIP 198[1]. The last three operations act on the elliptic curve alt_bn128.

1 https://github.com/ethereum/EIPs/blob/master/EIPS/eip-198.md

3

PRIOR ART

The ancient Greeks turned to oracles to gain insight into the will of their gods. Oracles got consulted about many important decisions throughout the ancient classical world. In other words, people did not have enough information to make decisions and turned to oracles for information beyond their understanding. Similar to their mythological namesakes, oracles in blockchain provide smart contracts with data outside of their environment to use in their decision-making. In Chapter 1, we explained that communication between blockchain and the outside world is not as trivial as might be expected due to the requirement of deterministic smart contracts. The need for external data on the blockchain and in smart contracts gave rise to several academic publications. This chapter presents an overview of existing efforts towards creating oracles according to three categories.

3.1 TRUSTED EXECUTION ENVIRONMENTS

One approach to alleviate trust in third parties, such as oracles, is to use so-called trusted execution environments (TEE), which aim to solve the problem of executing software on a remote computer owned and maintained by an untrusted party while ensuring integrity and confidentiality guarantees and usually take the form of a secure area in the main processor [9].

One example of TEE technology, introduced by Intel with its Skylake generation processors, is called Software Guard Extensions (SGX)[1]. SGX provides a new subset of hardware instructions that allows execution of software inside isolated environments called enclaves. These enclaves are isolated from other components running on the same machine, including the operating system. When a piece of code is executed in an enclave, an attestation is generated based on the program code and the output. The attestation is signed by a hardware protected private key, and can later be verified by a client using a public key that was provided by the process.

3.1.1 *Town Crier*

Zhang et al. [32] proposed an oracle called Town Crier (TC), which uses SGX to execute the data retrieval in an enclave to grant protection against malicious processes and the OS. Also, the attestation generated by SGX can be used to prove to a remote client that it is in-

1 https://software.intel.com/en-us/sgx

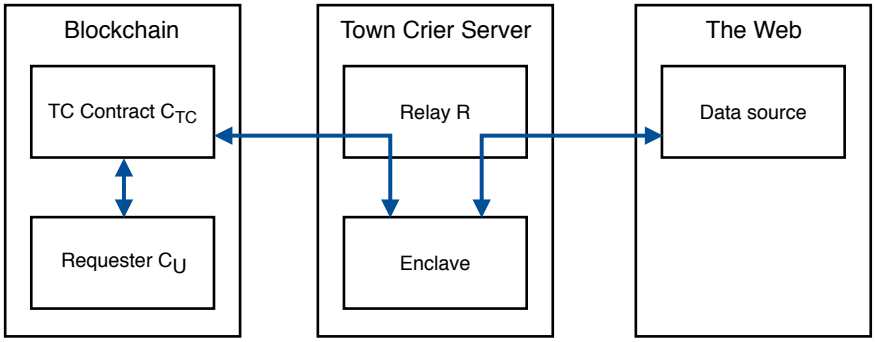


Figure 3.1: Architecture of Town Crier [32].

teracting with a legitimate, SGX-backed instance of Town Crier. As an oracle, Town Crier acts as a bridge between smart contracts and existing data sources on the Internet by combining a blockchain front end with an off-chain server, which in this case is equipped with trusted hardware. Town Crier's two primary goals are to provide authenticity of the retrieved data and confidentiality of private data. Additionally, Town Crier guarantees fair expenditure of gas for an honest requester and gas sustainability. In Chapter 4, we generalise Town Crier to create a framework which can provide these two properties to any service.

Town Crier is composed of three main components: the smart contract $\mathcal{C}_{TC}$, the Enclave from SGX containing code $prog_{encl}$, and a Relay $\mathcal{R}$. The first component resides on-chain, while the latter two form the off-chain part of the oracle. TC interacts with two kinds of entities: a requester $\mathcal{C}_U$, which is a smart contract that makes use of TC's service, and a data source, usually a web server, from which data is retrieved. The authors refer to the retrieved data as a *datagram*. An architectural overview of Town Crier is given in Figure 3.1. The roles of the three components in the figure are discussed below.

COMPONENT 1: TC CONTRACT $\mathcal{C}_{TC}$ $\mathcal{C}_{TC}$ acts as an interface for requesters to communicate with TC by accepting data requests from these requesters. In response, $\mathcal{C}_{TC}$ provides requesters with the corresponding datagram. Besides, $\mathcal{C}_{TC}$ also manages the ether that Town Crier uses to operate.

COMPONENT 2: THE ENCLAVE The Enclave is an SGX enclave that runs the TC code, denoted as $prog_{encl}$, to process data requests received by $\mathcal{C}_{TC}$, obtain the requested data via HTTPS, and return a data to the blockchain. Apart from interaction with the Relay, the Enclave is entirely isolated.

COMPONENT 3: RELAY $\mathcal{R}$ As the Enclave does not have network connectivity, the Relay provides the Enclave with a connection to three kinds of entities:

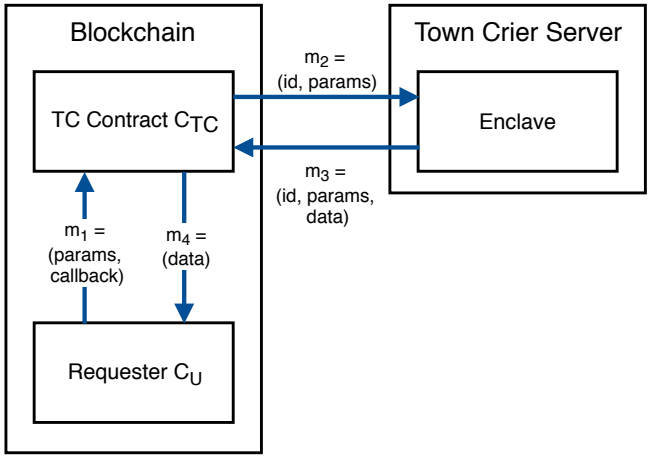


Figure 3.2: The data flows that occur when retrieving data through Town Crier [32].

1. The blockchain. The Relay inspects the blockchain to monitor $\mathcal{C}_{TC}$ for new datagram requests. Newly detected requests are sent to the Enclave. Conversely, the Relay also posts datagrams received from the Enclave back to the blockchain.

2. Clients. Requesters that retrieved data using Town Crier can access a web server that the Relay runs to handle attestation requests. An attestation provides a unique public key for the Enclave instance and proves that the Enclave is executing correct code in an SGX enclave.

3. Data sources. The Relay enables the communication between the Enclave and the data sources via an HTTP connection.

DATA FLOWS The Town Crier protocol can be described by the data flows that occur when a requester makes use of Town Crier's services. We can distinguish four data flows, which are visualised in Figure 3.2. In the figure, the Relay is omitted as it only relays messages and thus is not of conceptual importance. First, a request is sent from $\mathcal{C}_U$ to $\mathcal{C}_{TC}$. This message has form $m_1 = (params, callback)$, where *params* specifies the request and *callback* is the entry point to which the datagram should be returned. Upon receipt, $\mathcal{C}_{TC}$ generates a unique *id* and sends $m_2 = (id, params)$ to the Enclave. The Enclave contacts a data source based on *params* and returns a message of form $m_3 = (id, params, data)$ to $\mathcal{C}_{TC}$, where *data* is the datagram containing the requested data. $\mathcal{C}_{TC}$ compares the *id* and *params* in m_2 and m_3. If these values in both messages are equal, then $\mathcal{C}_{TC}$ sends the data to the entry point specified in *callback* of m_1 as $m_4 = (data)$.

For clarity reasons, some simplifications have been made in the protocol described above. First is that *callback*, as specified in m_1 is not necessarily an entry point in $\mathcal{C}_U$, but could also be in another smart

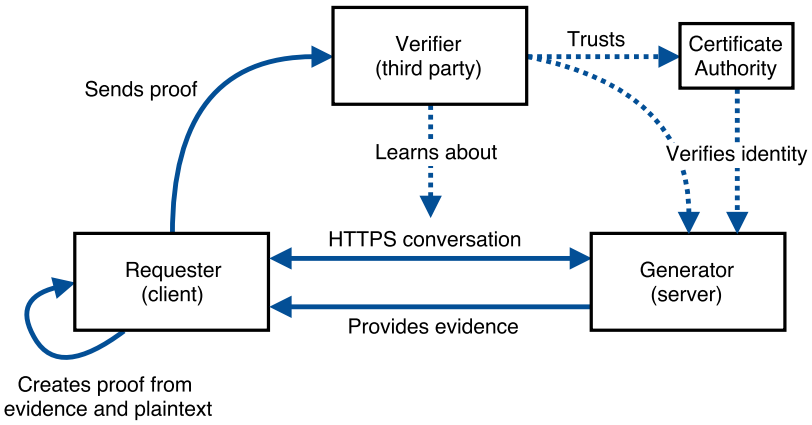


Figure 3.3: Non-repudiation model used in TLS-N [24].

contract. A second simplification is that $\mathcal{C}_U$ only makes a one-time datagram request. In that case, it can trivially match its request m_1 with the data received in m_4. To extend this to a scenario where multiple datagrams are supported, $\mathcal{C}_{TC}$ should send the *id* to $\mathcal{C}_U$ upon receiving m_1. If *id* is then also included in m_4, $\mathcal{C}_U$ can uniquely match requests and datagrams. Third is that we assumed that the Enclave could send signed messages to $\mathcal{C}_{TC}$. In practice, however, information can only be inserted into the (Ethereum) blockchain through transactions from a wallet. In other words, Relay needs a wallet $\mathcal{W}_{TC}$ to relay messages to the blockchain.

3.2 MODIFICATION AT THE DATA SOURCE

The trust issue that resides around constructing oracles exists because data sources do not sign the data they serve. However, if a data source would digitally sign the data it provides, then the authenticity of a message can always be verified. Here, we present a method of creating oracles by modification or extension of the data source's software.

3.2.1 *TLS-N*

In order to provide non-repudiation without the need to rely on a third party or trusted hardware, Ritzdorf et al. [24] proposed an extension to TLS 1.3 called TLS-N, that allows an entity on the Internet, called a *requester*, to generate redactable, non-interactive proofs of the contents of a TLS session with another entity, called the *generator*. The latter is usually a web server. A third party, called the *verifier*, can later verify the contents of the TLS session using this proof. The model of non-repudiation described above is illustrated in Figure 3.3.

The workings of TLS-N are illustrated in Figure 3.4. Initially, the requester establishes a TLS connection with the generator by negotiating the TLS-N parameters in the handshake, in addition to the

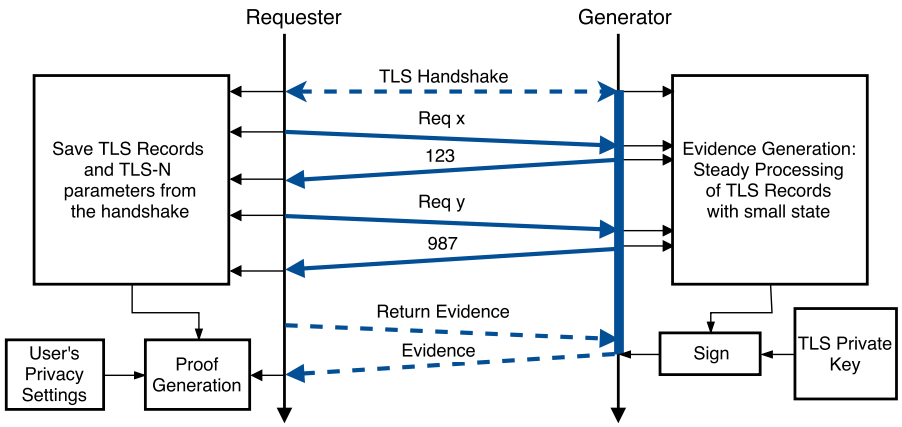


Figure 3.4: Overview of TLS-N [24].

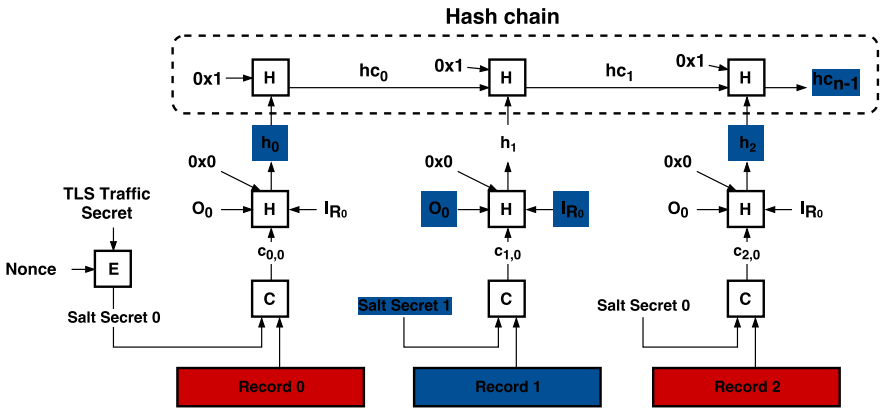
standard TLS parameters. One TLS-N parameter is the level of redaction granularity, which determines how precise a requester can later redact the proof, and can be set to either record-level or chunk-level. With record-level granularity, the requester can only apply redaction per full record, while chunk-level granularity is more precise at the cost of higher computational overhead.

During the TLS session, the generator keeps a small state, which is updated when the generator sends or receives a record. The state contains a hash value comprising all previous records, an ordering vector and a timestamp of when the connection was established. Computation of the hash value depends on the level of granularity. See Figure 3.5a and 3.5b for, respectively, record-level and chunk-level granularity.

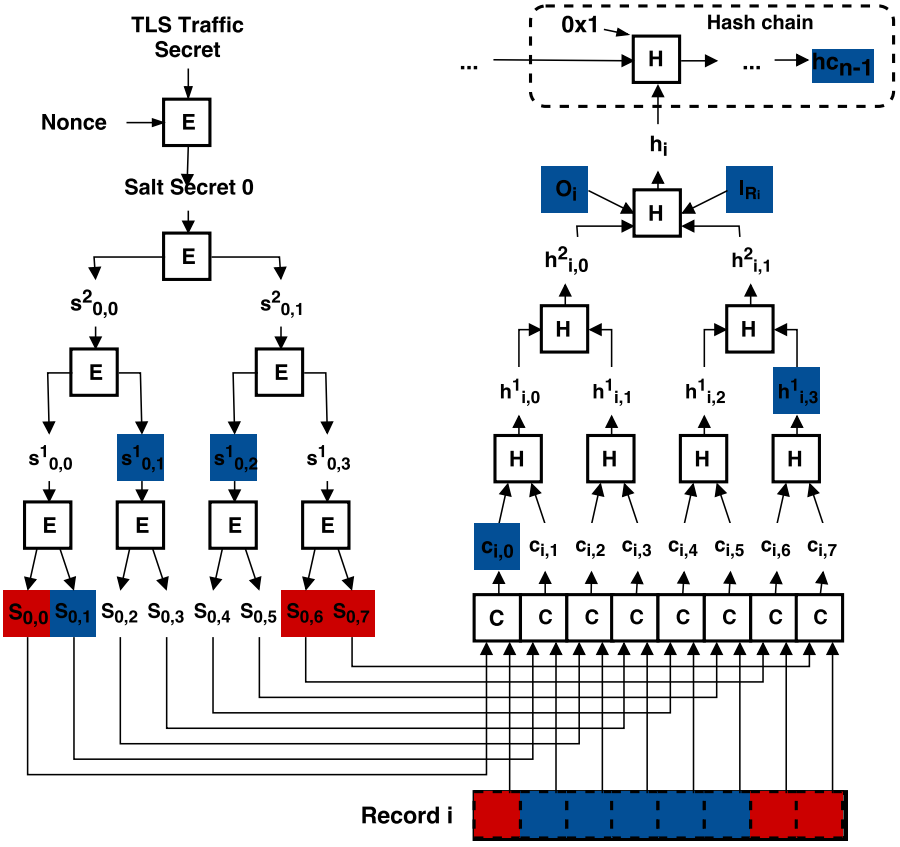
At any point in the session, the requester can create a proof of the contents so far by sending an 'evidence return' request to the generator. The generator creates the evidence by signing the state with its private key (see Figure 3.6). Together with the records, the requester can construct various kinds of proofs using the evidence. Depending on the situation, the certificate chain of the TLS connection can also be included. This could be useful, for instance, if third parties only possess the trusted root certificates and miss intermediate certificates required to verify the certificate chain.

As an example, non-repudiation for a single record i can be proven using:

1. Plaintext of record i.
2. Salt secret i.
3. Originator value O_i.
4. hc_{i-1}, the hash chain value from all previous records.
5. $h_{i+1}, ..., h_n$, the hashes of all next records.



(a) Record-level granularity.



(b) Chunk-level granularity.

Figure 3.5: Evidence Generation with record-level (a) and chunk-level granularity (b). All blue elements are included in the proof, while sensitive content, which is hidden in the proof, is marked red. [24].

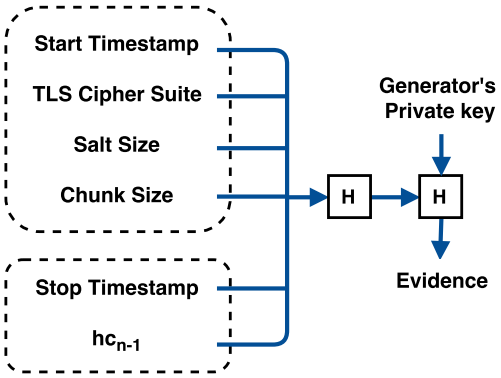


Figure 3.6: Structure of the evidence [24].

6. Evidence from the generator.
7. Certificate chain.

Using the values above, the verifier can use the first three items to rebuild the Merkle tree of the record, and thereby retrieve h_i. By combining h_i with item 4 and 5, the verifier can reproduce hc_{n-1}. The verifier believes the requester if the signature of the generator checks out with the certificate chain and hc_{n-1} from the evidence is equal to the reproduced version.

The example above can be extended to a proof of a full conversation by supplying the plaintexts for each non-sensitive record and the hash for each sensitive record.

IMPLEMENTING AN ORACLE WITH TLS-N In the same work, the authors also present how TLS-N can be used to implement an oracle. During the implementation of the oracle, they ran into two obstacles. First, note that, in this context, the inclusion of the certificate chain is not useful as there is not yet an on-chain way to verify TLS signatures based on the web-PKI. Therefore, the authors suggest that the verifying smart contract knows the generator's public key beforehand. Second, the authors use elliptic curve cryptography to reduce proof size. However, no curve is supported by both TLS and Ethereum. TLS supports secp256r1 and Ethereum uses secp256k1, which means that they could either use Ethereum's curve or implement verification of secp256r1 in a smart contract. The evaluation results are found in Table 3.1.

3.3 DECENTRALIZED ORACLE NETWORKS

The approaches above aim to mitigate trust in an intermediary by creating proofs that the oracle acted faithfully. A different approach is to replace the off-chain component of the oracle by a decentralised network of nodes. An oracle of this type is often referred to as a decentralised oracle network or DON. Sánchez de Pedro et al. [22] define

Table 3.1: Gas costs for validating public, record-level proofs within an Ethereum smart contract based on the conversation size and the elliptic curve [24].

	Conversation size			
	1 KB		10 KB	
	secp256r1	secp256k1	secp256r1	secp256k1
Basic Gas	119,758		737,159	
Total Gas	1,284,723	131,286	1,938,872	782,219
Ether	0.0257	0.0026	0.0388	0.0156

a DON as a computer network made up of nodes, which communicate and operate as peers in compliance with an agreed protocol to acquire knowledge of information that is external to the network. In such a network, the data that is acquired by multiple nodes is aggregated into a single answer using an aggregation function. One example of an aggregation function is majority voting, where if a majority of nodes return the identical value, this value will become the answer. If no majority is found, an error is returned. In majority voting, the oracle will return the correct value if a majority of the nodes is functioning correctly. Different DONs have been proposed [11, 22], each with their own incentive structure. Below, we present an established network called ChainLink.

3.3.1 *ChainLink*

ChainLink is a decentralized oracle network proposed by Ellis, Juels, and Nazarov [11], whose architecture is depicted in Figure 3.7.

To use ChainLink, a requester first creates a specification of the services it requires, called a Service Level Agreement (SLA) proposal. The proposal includes details such as query parameters, the number of oracle nodes required, and how their responses should be aggregated. The proposal is then submitted to a smart contract called *CHAINLINK-SC*, which serves as the ChainLink's on-chain interface. Besides *CHAINLINK-SC*, ChainLink includes three types of smart contracts:

1. Reputation contracts that keep track of each oracle node's performance.

2. Order-matching contracts that take an SLA proposal and collect bids from oracle nodes. Bids are selected based on the reputation contract, after which the SLA is finalised.

3. Aggregating contracts that collect the oracle nodes' responses and calculates the final collective result of the query.

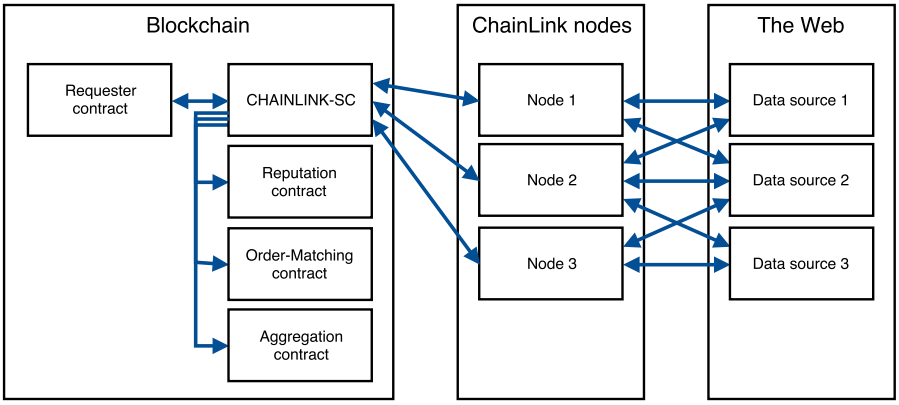


Figure 3.7: Architecture of ChainLink [11].

Several contracts of each type can exist, and the client can specify which should be used for this assignment. The flow from proposal to receival of the requested data looks as follows. First, the requester sends an SLA proposal to *CHAINLINK-SC*. The proposal is redirected to an order-matching contract, which lists all proposals to allow oracle nodes to bid on them. Bidding on a proposal means that oracle nodes commit to the assignment by paying a so-called penalty payment that can be used to fine misbehaving nodes. The order-matching contract only accepts bids that meet the SLA specifications.

After a period called the bidding window, the requested amount of oracle nodes is selected based on the reputation contract, and the SLA is finalised. At the moment of finalisation, the selected nodes receive a notice and carry out the client's assignment off-chain.

The off-chain part of ChainLink consists of a network of oracle nodes that run open source software, called ChainLink Core, which can be extended using components called adapters that allow the operators to offer additional specialised services, for instance, if retrieving certain data requires interacting with multi-step APIs.

The responses of the selected nodes are aggregated into a single answer according to the aggregation strategy described in the selected aggregation contract. The most straightforward approach to aggregating responses is *in-contract aggregation*, where the aggregation function is run in an aggregation contract. An advantage is that the contract code can be publicly inspected, which means that correct behaviour can be verified. On the other hand, there exists the problem of freeloading, as an oracle node can cheat by observing responses of other nodes as to avoid performing the query itself. This behaviour weakens the security of the network but could be solved, according to the authors, by using a commit-and-reveal scheme, where, in a first round, oracles send the aggregation contract cryptographic commitments to their responses. After receiving a certain amount of responses, a second round is initiated in which nodes reveal their responses.

Performing computation on-chain can be costly, as, in case of Ethereum, ether must be spent to pay the cost of transmitting and processing the oracles' messages, as well as executing the aggregation function. A more cost-effective approach is to aggregate responses off-chain and transmit the final answer to *CHAINLINK-SC*, which could be achieved by using a (t, n)-threshold signature scheme [28], where oracles share a public key pk, while the corresponding private key sk is shared among oracle nodes $O_0, ..., O_n$. These shares, which we refer to as $sk_0, ..., sk_n$, also have their own public key $pk_0, ..., pk_n$. A node O_i can generate a partial signature s_i, that can be verified using pk_i. Also, a set of t (where $0 < t \leq n$) partial signatures from different nodes on the same value can be combined to create a valid collective signature on that value under sk, which can be verified using the shared public key pk.

Finally, the performance of each node is reported to the reputation contract to be used for allocating future assignments, and the answer is returned to the contract function specified by the requester in the SLA.

3.4 ADVANTAGES AND DISADVANTAGES

In this section, we review the advantages and disadvantages associated with each type of oracle, which are summarized in Table 3.2.

TRUSTED EXECUTION ENVIRONMENTS The advantage of using TEE technology is that, contrary to the approach in Section 3.2, no modification at the data source is required. Therefore, the oracle can retrieve data from any data source on the Web. Also, this approach is generally more efficient than using a decentralised oracle network (see Section 3.3), both in terms of gas cost and delivery time.

However, the approach assumes that trusted hardware can, indeed, be trusted. Trusting a system based on trusted hardware creates a single point of failure if someone manages to breach the hardware's security. In other words, a client of the service needs to trust Intel and its SGX technology to be able to trust the attestations. Costan and Devadas [9] performed a security analysis based on the available public resources and found that SGX is vulnerable for attacks, including cache timing attacks, that rely on observing the time differences between accessing a cached memory location and an uncached memory location, and power analysis attacks, that indirectly measure the power consumption of a computer system or its components. Also, several crucial implementation details are not covered by publicly available documentation, which makes it impossible for anyone to reason about SGX's security properties. Finally, the authors found in Intel's official documentation that SGX includes a so-called launch control feature whose intended use is a licensing mechanism that re-

quires software developers to enter a (yet unspecified) business agreement with Intel to be able to author software that takes advantage of SGX's protections.

MODIFICATION AT THE DATA SOURCE The main characteristic of oracles in this category is that modification or extension of the data source's software is required. An advantage of this approach is that it does not require the use of a trusted third party or trusted hardware such as Intel SGX. Also, the security is provable.

On the other hand, this approach also has several disadvantages. Software must be installed at the data source, so it requires changes to existing infrastructure. While this seems a strong assumption, we argue that many sources already make their information publicly available on the Internet. With the adoption of smart contracts, it will become more attractive for them to extend their information provision to the blockchain. Within this category, we can distinguish sub-categories of oracles depending on the layer where the software modifications are made. For instance, TLS-N chose to provide authentication at the transport layer, which benefits from the fact that most interactions over the Internet are protected using TLS, and so their solution could be used with almost any data source. Also, because non-repudiation is provided at a lower level than where conventional applications reside, the solution can be used to benefit a variety of applications. Also, the already established public-key infrastructure (PKI) can be reused to provide authentication. However, a disadvantage of this approach is that if authentication is provided at the transport layer, aggregation or other forms of trusted computation over the data is not possible as the conversation is signed and transferred in verbatim, which also means higher parsing costs for the receiving smart contract.

Oracles could also be implemented at the application layer. According to Ritzdorf et al. [24], such approach could make sense, as one could argue that signing content should not be handled at the TLS layer. Also, solutions at the application layer are useful when more diverse functionality is required.

DECENTRALIZED ORACLE NETWORKS An advantage of Decentralized Oracle Networks is that the on-chain and off-chain components are all fully decentralised, which limits the trust in any single party. One could also argue that this approach complies with the blockchain philosophy of decentralisation.

Nonetheless, decentralised oracle networks are inherently inefficient as all the participating parties require a fee and, for each request, reaching a sufficient number of answers takes time. Also, full decentralisation makes these networks vulnerable to at least two Sybil attacks that involve an adversary controlling multiple oracles nodes. In

Table 3.2: Comparison of existing oracle solutions.

Category	Based on	Advantages	Disadvantages
Software modifications	Cryptography	Provable security	Requires modification at the source
Trusted hardware	Enclaves	No need for modifications at the source	Hardware as single point of failure
Decentralized Oracle Networks	Incentives	No modifications at the source and in line with blockchain philosophy	Inefficient, expensive, and not provable secure

the first attack, the adversary can attempt to dominate the oracle pool and provide false data at strategic times. Even with the penalty payment that oracle providers pay, such an attack could be feasible if it involves large transactions and high-value contracts. Besides a single adversary, a Sybil attack can also be performed by multiple colluding adversaries. Another form of Sybil attack, called mirroring, comprises an adversary that bids on a proposal using multiple nodes, but only retrieves the data once. The data is then shared among the sibling nodes whom all submit the same answer to get the reward multiple times. While posing less of a security threat than data falsification, mirroring does degrade security in the sense that it eliminates the error correction as all answers are now based on the same query.

4

CHAINBRIDGE: A FRAMEWORK FOR CREATING HYBRID SERVICES ON ETHEREUM

Two drawbacks of smart contracts are that they cannot directly retrieve data from the Web and computation must be paid for per computational operation. One approach to alleviating these concerns is to build services that are comprised of both on-chain and off-chain components and allow us to perform data retrieval and computation tasks off-chain. We refer to services with both on-chain and off-chain components as *hybrid services*.

In this chapter, we introduce CHAINBRIDGE, a framework to build hybrid services on Ethereum that can be used to run any algorithm off-chain. CHAINBRIDGE is a generalisation of an existing oracle called Town Crier [32]. It is designed to be generic and can, therefore, be seen as a separate contribution that is of independent interest. However, by applying it to build MUSCLE-BP and MUSCLE-TP in, respectively, Chapter 6 and Chapter 7, we gain evidence of CHAINBRIDGE's usefulness for constructing complex services. We introduce CHAINBRIDGE according to the following outline. First, we explain its relation to an existing oracle called Town Crier. Then, we introduce the components of CHAINBRIDGE and examine the lifecycle of a request. Finally, we present the framework and the security properties it guarantees.

4.1 RELATION TO TOWN CRIER

CHAINBRIDGE is a generalisation of Town Crier, an oracle that leverages trusted hardware (Intel SGX) to provide authenticity of the retrieved data, and is, therefore, closely related. However, CHAINBRIDGE is more general as it can, for instance, also be used to create services that provide off-chain computation or integration with legacy systems. CHAINBRIDGE differs from Town Crier in the following ways:

- The Relay and the Enclave with program $prog_{encl}$ from Town Crier have been replaced by a web server $\mathcal{S}$ that runs an algorithm $\mathcal{A}$. CHAINBRIDGE is a generic framework in the sense that it does not make any assumptions about $\mathcal{A}$ and treats it as a black box with two methods: *Initialize* and *Execute*. Specific services based on the framework are created by defining $\mathcal{A}$ and setting the system constants.

- All elements related to authentication are removed. As the properties required by specific services may vary, CHAINBRIDGE only

guarantees properties that are relevant for every hybrid service. Other properties, such as authentication, are to be provided by the algorithm $\mathcal{A}$.

- In Town Crier, when the off-chain components delivered a response to a request, the parameters of the request were included to be validated against the parameters stored by the system. As this is only useful in combination with trusted hardware, this check has been removed. Safeguarding against tampering by $\mathcal{S}$ should now be handled by the algorithm $\mathcal{A}$.

4.2 COMPONENTS

CHAINBRIDGE provides a foundation to create services that operate both on-chain and off-chain. We refer to a smart contract that utilises a CHAINBRIDGE-based service as a *requester*, which we denote by $\mathcal{R}$. The architecture of a CHAINBRIDGE-based service has two main components (see Figure 4.1): an on-chain smart contract $\mathcal{C}$ and an off-chain web server $\mathcal{S}$. Both are discussed below.

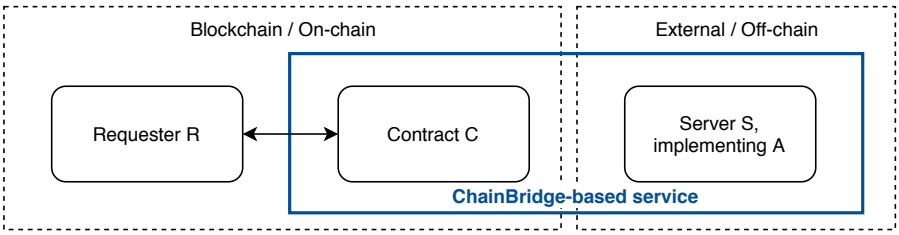


Figure 4.1: Overview of the ChainBridge architecture.

ON-CHAIN: CONTRACT $\mathcal{C}$ This smart contract acts as the blockchain front end of the service by accepting requests from a requester $\mathcal{R}$ and returning responses delivered by server $\mathcal{S}$. Additionally, $\mathcal{C}$ manages the service's monetary resources to guarantee the security properties presented in Section 4.6.

OFF-CHAIN: SERVER $\mathcal{S}$ The server continuously monitors $\mathcal{C}$ on the blockchain for new requests. Upon detection of a request, $\mathcal{S}$ executes $\mathcal{A}$ using the specified parameters. The result is then sent back to the blockchain. Server $\mathcal{S}$ is an ordinary web server application, and thus can be subverted by an adversary to cause delays or failures. A principal design aim of CHAINBRIDGE is that the expenditure of honest requesters, in relation to a request, is no higher than a certain limit.

4.3 LIFECYCLE OF A REQUEST

A typical request goes through the following phases:

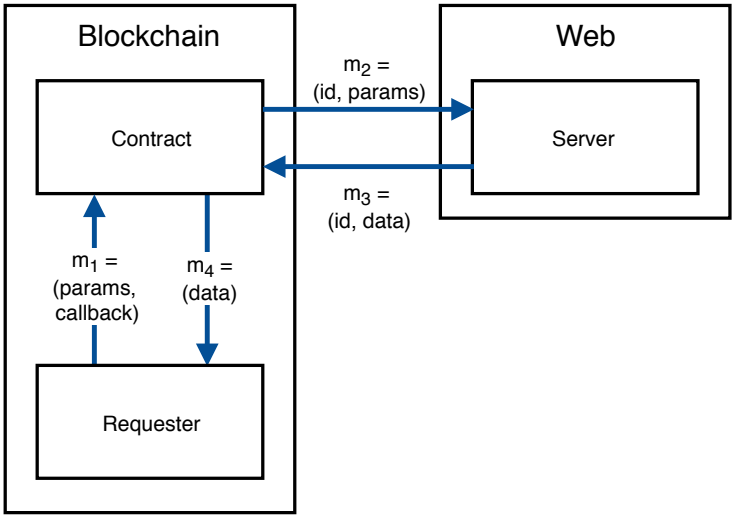


Figure 4.2: The data flows that occur between request and delivery.

1. *Initiate request*. $\mathcal{R}$ sends a request to $\mathcal{C}$ on the blockchain. The request is of form $m_1 = (params, callback)$, where *params* specifies the input parameters for $\mathcal{A}$ and $callback = (address, function)$ indicates the contract address and function to which the data is to be returned.

2. *Storage*. Upon receiving the request, $\mathcal{C}$ generates a unique *id* and stores the request as $m_2 = (id, params, callback)$. Also, *id* is returned to $\mathcal{R}$.

3. *Execute*. Server $\mathcal{S}$ monitors $\mathcal{C}$ for new requests. Upon detection of the request with parameters *params*, $\mathcal{S}$ applies algorithm $\mathcal{A}$ on *params* and then sends message $m_3 = (id, data)$ to $\mathcal{C}$, where *data* is the output of $\mathcal{A}$.

4. *Return*. $\mathcal{C}$ sends $m_4 = (data)$ to the *callback*.

The stages above are depicted in Figure 4.2, in which we assume that the entry point *callback* is in $\mathcal{R}$, as to simplify the illustration. Note, however, that *callback* could also be a function in another smart contract.

4.4 THE FRAMEWORK

In this section, we explain the workings of CHAINBRIDGE by defining the variables and describing the behaviour of each actor.

VARIABLES All variables in Table 4.1 denote an amount in gas, except for $\$f$ which is in ether. We assume that the average GASPRICE remains constant during the lifetime of a request, which allows us to

Table 4.1: Variables and constants in CHAINBRIDGE.

Symbol	Description
$\$f$	The amount of gas a requester deposits to refund the service's gas expenditure to deliver a data
$\$g_{req}$	STARTGAS when invoking *Request*
$\$g_{dvr}$	STARTGAS when invoking *Deliver*
$\$g_{cncl}$	STARTGAS when invoking *Cancel*
$\$g_{clbk}$	STARTGAS for callback while executing Deliver, set to the maximum value that can be reimbursed
$\$G_{min}$	Gas required for *Deliver* excluding callback for a request that was not previously cancelled
$\$G_{max}$	Maximum gas the service can provide to invoke *Deliver*
$\$G_{cncl}$	Gas needed to invoke *Cancel*
$\$G_{\emptyset}$	Gas needed for *Deliver* on a cancelled request

implicitly convert gas and ether for notational convenience by using GASPRICE as the conversion rate.

$\$G_{min}$, $\$G_{max}$, $\$G_{cncl}$, and $\$G_{\emptyset}$ are constants, set by $\mathcal{C}$, that denote the amount of gas needed to execute a certain part of $\mathcal{C}$'s code. $\$f$, $\$g_{req}$, and $\$g_{cncl}$ are chosen by the requester while interacting with $\mathcal{C}$. Note that although some values are set by the requester who could be malicious, a user-initiated transaction aborts if the values are too small. Finally, $\$g_{dvr}$ is set by server $\mathcal{S}$ in the *Handle* method.

INITIALIZATION A deposit of at least $\$G_{max}$ is made into the wallet $\mathcal{W}_S$ to initialise the service. This wallet is used by $\mathcal{S}$ for sending transactions to $\mathcal{C}$.

CONTRACT $\mathcal{C}$ The contract has three roles:

- If $\mathcal{C}$ receives a request with fee $\$f$ from $\mathcal{R}$, it first asserts whether $\$G_{min} \leq \$f \leq \$G_{max}$. If so, $\mathcal{C}$ records the request along with a unique *id* to be detected by $\mathcal{S}$.
- If $\mathcal{C}$ receives a response from $\mathcal{S}$ through $\mathcal{W}_S$, it verifies that the request is valid and then forwards the result by calling the callback address and function specified in the initial request with the gas limit set to $\$g_{clbk} := \$f - \$G_{min}$.

- If $\mathcal{C}$ receives a cancellation request for specific request, it first verifies whether the cancellation has been initiated by the creator of the request. The requester then receives a cancellation fee if the data has not been delivered yet and is not already cancelled.

Each of these three roles are related to one of $\mathcal{C}$'s functions. See Figure 4.3 for a full specification.

Contract $\mathcal{C}$

Initialize:

(I1) $Counter := 0$

Request: On recv $(params, callback, \$f, \$g_{req})$ from $\mathcal{R}$:

(R1) Assert $\$G_{min} \leq \$f \leq \$G_{max}$

(R2) $id := Counter$

(R3) $Counter := Counter + 1$

(R4) Store$(id, params, callback, \$f, \mathcal{R})$

(R4) Return id

Deliver: On recv $(id, data, \$g_{dvr})$ from $\mathcal{W}_S$:

(D1) If isCancelled[id] and not isDelivered[id]

(D2) Set isDelivered[id]

(D3) Send $\$G_{\emptyset}$ to $\mathcal{W}_S$

(D4) Return

(D5) Retrieve stored $(id, params, callback, \$f, \_)$

(D6) Assert $\$f \leq \$g_{dvr}$ and isDelivered[id] not set

(D7) Set isDelivered[id]

(D8) Send $\$f$ to $\mathcal{W}_S$

(D9) Set $\$g_{clbk} := \$f - \$G_{min}$

(D10) Call callback$(data)$ with gas $\$g_{clbk}$

Cancel: On recv $(id, \$g_{cncl})$ from $\mathcal{R}$:

(C1) Retrieve stored $(id, \_, \_, \$f, \mathcal{R}')$

(C2) Assert $\mathcal{R} = \mathcal{R}'$ and $\$f \geq \$G_{\emptyset}$
and isDelivered[id] not set and isCancelled[id] not set

(C3) Set isCancelled[id]

(C4) Send $(\$f - \$G_{\emptyset})$ to $\mathcal{R}$

Figure 4.3: Contract $\mathcal{C}$of CHAINBRIDGE.

SERVER $\mathcal{S}$ Upon initialization of the service, $\mathcal{S}$ initializes algorithm $\mathcal{A}$, which can then optionally perform some setup. The server inspects the blockchain and monitors $\mathcal{C}$ for new requests. Newly de-

tected requests are processed by *Handle*, which invokes the *Execute* method of $\mathcal{A}$. $\mathcal{S}$ then forwards the results of $\mathcal{A}$ back to the blockchain as a transaction from $\mathcal{W}_S$. An honest server does this exactly once for each valid request, while a malicious server could attempt to deliver the data twice. However, CHAINBRIDGE is designed in such a way that this action is always unfavourable to perform. The program for $\mathcal{S}$ is shown in Figure 4.4.

Program for server $\mathcal{S}$ with algorithm $\mathcal{A}$
Initialize:
    Output $\mathcal{A}$.Initialize()
Handle($id, params$**):**
    $(data) \leftarrow \mathcal{A}$.Execute($params$)
    $\$g_{dvr} := \$G_{max}$
    Send($id, data, \$g_{dvr}$) to $\mathcal{C}$ as $\mathcal{W}_S$
Main:
    Loop Forever:
      Wait for new request at $\mathcal{C}$
      Upon request ($id, params, \_, \_, \_$):
        Fork a process of Handle($id, params$)
    End

Figure 4.4: Server $\mathcal{S}$of CHAINBRIDGE.

ALGORITHM $\mathcal{A}$ CHAINBRIDGE-based services are initialised with an algorithm $\mathcal{A}$. As the framework itself is generic, we view $\mathcal{A}$ as a black box with two methods:

- *Initialize*. This function initialises the algorithm. Here, the algorithm could, for instance, set certain parameters or generate keys.
- *Execute*. Applies the algorithm on the specified inputs and returns the result as output.

REQUESTER $\mathcal{R}$ An honest requester sends a request containing *params* and *callback*, along with $\$f$ ether, such that $\$f$ is equal to $\$G_{min}$ plus the cost of executing response transaction to *callback*. The value of STARTGAS is set to $\$g_{req}$, which should be sufficient to send $(params, callback)$ to $\mathcal{C}$'s *Request* method and, therefore, depends on the size of *params*. At some point, before the requested data is delivered, $\mathcal{R}$ can invoke *Cancel* with STARTGAS $= \$G_{cncl}$ to receive a partial refund. An honest requester invokes *Cancel* at most once for each of her requests and never for any request that it did not initiate.

CHAINBRIDGE is designed in such a way that a malicious requester never profits from cancelling a request more than once.

4.5 ASSUMPTIONS

CHAINBRIDGE makes the following security assumptions:

- The contract $\mathcal{C}$ is publicly visible on the blockchain and its source code is published for potential requesters to inspect. Therefore, we assume that $\mathcal{C}$ behaves honestly.
- The server $\mathcal{S}$ can be corrupted by the adversary. In that case, the adversary could tamper with or delay communications.
- A transaction m sent from a wallet $\mathcal{W}_X$ of account X preserves the integrity of transactions and messages as the sender digitally signs them. However, transactions and messages are not confidential.

4.6 SECURITY PROPERTIES

Town Crier [32] introduced key security properties that are applicable to any hybrid service. The first property, *gas sustainability*, ensures that a malicious requester is unable to deplete the service's gas resources. Second, Town Crier ensures *fair expenditure for an honest requester*. Informally, the fee paid by a user contract that interacts with a CHAINBRIDGE-based service is at most a small amount to cover the operating costs, even if the service is malicious. These security properties have already been proven the Town Crier paper.

To generalise CHAINBRIDGE from Town Crier, we primarily changed the off-chain component $\mathcal{S}$, whose code is not involved in the proofs of the security properties that CHAINBRIDGE provides. Also, the only change made to $\mathcal{C}$ is that we do not check whether the value of *params* in the request and response are equal. This modification is only involved in proving that Town Crier ensures authenticity of the retrieved data, which CHAINBRIDGE does not provide. Therefore, as none of our modifications affects the proofs of properties CHAINBRIDGE provides, we refer the reader to this research paper for security analyses and proofs. Below, we recall the formal definitions of these properties, as developed by Zhang et al. [32].

4.6.1 *Gas sustainability*

In Ethereum, a user who initiates a transaction pays the gas needed to execute the function, including gas costs resulting from dependent calls. However, if data is sent from server $\mathcal{S}$ to $\mathcal{C}$, $\mathcal{S}$ initiates the transaction. Therefore, $\mathcal{S}$ also pays the gas. For that reason, hybrid

services are potentially vulnerable against malicious requesters that attempt to deplete the service's financial resources by triggering calls for which the service is not reimbursed. A successful attack effectively results in a denial-of-service attack, as the service cannot operate without gas.

Definition 4.1 (K-Gas Sustainability)**.** Let $bal(\mathcal{W})$ denote the balance of an Ethereum wallet $\mathcal{W}$. A service with a wallet $\mathcal{W}$ and blockchain functions $f_1, ..., f_n$ is K-gas sustainable if the following holds. If $bal(\mathcal{W}) \geq K$ prior to execution of any f_i and the service behaves honestly, then after each execution of an f_i initiated by $\mathcal{W}$, $bal(\mathcal{W}) \geq K$.

Ethereum trivially guarantees 0-gas sustainability, because if a wallet submits a transaction with insufficient funds, the wallet's balance will drop to 0. To be K-gas sustainable for $K > 0$, each call made by the service on a requesters behalf must be compensated for. Moreover, the service must have sufficient gas for each call or such reimbursement be reverted along with the rest of the transaction. Equal to Town Crier, we ensure gas sustainability by requiring that requesters make gas payments up front.

4.6.2 *Fair expenditure*

An honest requester should pay only for what happens on its behalf. A service should guarantee a limit on what should be paid, both when the data is delivered successfully and if the request is cancelled. A more formal definition is given below.

Definition 4.2 (Fair Expenditure)**.** For any *params* and *callback*, let $\$g_{req}{}^*$ and $\$f^*$ be the respective values chosen by an honest requester for $\$g_{req}$ and $\$f$ when submitting the request $(params, callback, \$f, \$g_{req})$. For any such request submitted by an honest user $\mathcal{R}$, one of the following holds:

- The data is delivered by $\mathcal{S}$ before the request is cancelled and requester $\mathcal{R}$ spends at most $\$g_{req}{}^* + \$G_{cncl} + \$f^*$.
- The requester spends at most $\$g_{req}{}^* + \$G_{cncl} + \$G_{\emptyset}$.

Reimbursement is performed by $\mathcal{C}$ to guarantee that a malicious CHAINBRIDGE-based service is not able to steal money from an honest user without delivering data.

5

SPECIFICATION FOR MULTI-SOURCE ORACLES BASED ON AGGREGATE SIGNATURES

Using single source oracles, the number of transactions that must be performed is linearly dependent on the amount of data sources because each request is handled separately. This could be improved upon by simply putting data from all sources, along with required proofs, in a single transaction. Even though, the total payload still remains the same. In Chapter 6 and 7, we propose multi-source oracles that employ aggregate signature schemes based on, respectively, bilinear pairings and trapdoor permutations to compress the amount of data that is sent to the blockchain. After responding to the oracle's query, the data sources are not further involved in the creation of the aggregate signature or its verification. Consequently, the authenticity of the data can be publicly verified without any interaction with the data sources. On top of that, data sources are asked only to sign their message and do not have to verify intermediate signatures or perform other additional computation. Both oracles are built on the CHAINBRIDGE framework introduced in Chapter 4.

In this chapter, we create a common specification for both oracles. First, we present the system model and introduce assumption we made. Then, we list the requirements and explain why the selected aggregate signature schemes fit these requirements best.

5.1 SYSTEM MODEL

We consider a model that comprises four types of entities, namely a requester $\mathcal{R}$, MUSCLE-TP's on-chain component $\mathcal{C}_{MUSCLE}$, MUSCLE-TP's off-chain server $\mathcal{S}_{MUSCLE}$, and data sources $\{\mathcal{D}_1,.., \mathcal{D}_n\}$. $\mathcal{R}$ sends a request to $\mathcal{C}_{MUSCLE}$, containing a description of data sources. Upon detection of a new request, $\mathcal{S}_{MUSCLE}$ retrieves signed data from data sources $\{\mathcal{D}_1,.., \mathcal{D}_n\}$. $\mathcal{S}_{MUSCLE}$ then aggregates the individual signatures into an aggregate signature, which is sent back to $\mathcal{C}_{MUSCLE}$, along with the retrieved data itself. The system model is depicted in Figure 5.1.

5.2 ASSUMPTIONS

We make the following assumptions:

1. We assume the Random-Oracle model.
2. $\mathcal{R}$, $\mathcal{C}_{MUSCLE}$, and data sources $\{\mathcal{D}_1,.., \mathcal{D}_n\}$ are trusted entities.

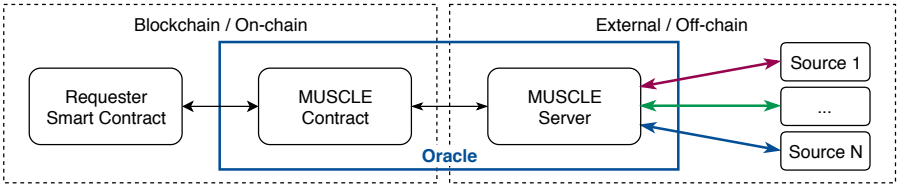


Figure 5.1: System model of MUSCLE-TP.

3. $\mathcal{S}_{MUSCLE}$ could be controlled by a polynomial-time adversary, whom we allow adaptive access to the encryption oracle.

4. The public keys of the data sources are publicly available, both on the blockchain and the Web.

5.3 REQUIREMENTS AND DESIGN DECISIONS

In designing MUSCLE-TP, we set the following requirements:

1. The data sources each have their own key pair under which they sign the data.

2. The verification is non-interactive, i.e., does not require interaction with either the oracle or the data sources.

3. Data sources sign data, but do not have to perform other tasks such as the verification of intermediate aggregate signatures.

4. The authenticity of the retrieved data should be publicly verifiable by any smart contract, including $\mathcal{R}$.

While the first two requirements are satisfied by most of the aggregate signature schemes, the third guided us towards making a more specific selection. We believe that, in practice, asking data sources to sign their data is already a strong assumption and consider asking the sources to also expend additional resources on verifying our intermediate signatures to be unrealistic, as they in no way benefit from performing the verification. We have found three aggregate signature schemes [5, 6, 12] that do not require verification before signing, a property that is also referred to as lazy verification. Of these three schemes, the scheme by Fischlin et al. [12] is a sequential, but less efficient, variant of the scheme by Boneh et al. [5]. As the sequential property is not beneficial to our situation, we decided to select the schemes by Boneh et al. [5] and Brogle et al. [6], where the first is based on bilinear maps and the latter on trapdoor permutations (e.g. RSA [25]).

The fourth requirement is that signatures are publicly verifiable. While signatures are publicly verifiable by definition, we note that verifying the authenticity of the data source alone is not enough, as a data source might offer different kinds of data. For instance, a data

source *www.example.com* could offer a list of cities at web address *www.example.com/cities* and a list of users at *www.example.com/users*. Following terminology from the World Wide Web Consortium (W3C)[1], we refer to a location at which a data source offers specific data as an *endpoint*. Verification in the context of a specific endpoint is not explicitly supported by any aggregate signature scheme, and we see two options to satisfy this requirement. The first option is to generate a separate key pair for each endpoint. While this approach satisfies our requirement, key management becomes difficult for data sources that have many endpoints. On top of that, if any endpoint contains a parameter, we need a key pair for each parameter. The second option and the one we elected makes use of the fact that, in CHAINBRIDGE, the requester already includes an array *sources* that contains string representations, referred to as labels, of each endpoint upon creating a request. The labels are used by the oracle to determine the data to be retrieved. If the label is passed along to the data source, the data source can verify that the correct route has been called by the oracle and concatenate the string to the message before signing. The verification algorithm then takes as additional input the set of labels and verification is only successful if the correct labels are used. Note that the set of labels do not need to be sent back to the blockchain, as the requester created the labels on the blockchain in the first place. Each data source has to maintain only one key pair using this option.

1 https://www.w3.org/

6

MUSCLE-BP: MULTI-SOURCE ORACLE BASED ON BILINEAR PAIRINGS

In this chapter, we discuss our first proposed oracle MUSCLE-BP (MUlti-Source oraCLE - Bilinear Pairings), which is based on the specification presented in Chapter 5. In MUSCLE-BP, the signature data that is sent back on-chain besides the actual data is of constant size, regardless of the amount of data sources involved. Furthermore, MUSCLE-BP is built on the CHAINBRIDGE framework introduced in Chapter 4. The chapter is structured as follows. First, the properties of the aggregate signature scheme that forms the basis of MUSCLE-BP are introduced. Then, we present MUSCLE-BP, after which we evaluate its computational and communicational complexity. An assessment of MUSCLE-BP's performance, in terms of gas costs, is deferred to Chapter 8, where we compare a sample implementation of the oracle with five other oracles.

6.1 AGGREGATE SIGNATURE SCHEME

MUSCLE-BP is built on an aggregate signature scheme called BGLS, which was presented by Boneh et al. [5] and is based on BLS signatures [4] (see Section 2.1.4). To verify signatures in the context of an endpoint, which is not explicitly supported by the aggregate signature scheme, we modify the scheme by concatenating a description of the endpoint, called a label, to the message. Furthermore, the original BGLS scheme is designed to work with type 2 pairings (see Section 2.1.4). However, Ethereum currently only supports type 3 pairings on a specific elliptic curve called *alt bn128*. To cope with this, we include an optional modification, proposed by Chatterjee et al. [8], to the *Sign* algorithm (see Algorithm 2) that makes BGLS compatible with type 3 pairings.

6.2 SYSTEM DESIGN

MUSCLE-BP comprises two components, an on-chain smart contract $\mathcal{C}_{MUSCLE}$ and an off-chain server $\mathcal{S}_{MUSCLE}$, and is divided into six phases: Initialization, Request, Data Retrieval, Aggregation, Deliver, and Verification. We describe each of these phases in the following subsections. An explanation of the symbols used is found in Table 6.1. Boneh et al. and Chatterjee et al. use groups $\mathbb{G}_1$ and $\mathbb{G}_2$ oppositely. For instance, the hash function in the first outputs a value in $\mathbb{G}_2$, while the latter outputs a value in $\mathbb{G}_1$. Here, we use follow Chatterjee et al.'s notation.

Table 6.1: Description of symbols used in MUSCLE-BP.

Symbol	Description
$\mathcal{C}_{MUSCLE}$	The on-chain component of MUSCLE-BP
$\mathcal{S}_{MUSCLE}$	The off-chain component of MUSCLE-BP
$\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$	Groups of the same prime order p
e	Bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$
g_1	Generator of $\mathbb{G}_1$
g_2	Generator of $\mathbb{G}_2$
H	One way hash function $H : \{0,1\}^* \rightarrow \mathbb{G}_1$
sk_i	Private key of data source i
pk_i	Public key of data source i
m_i	Message m provided by data source i
σ_m	Signature of message m
Σ	An aggregate signature

INITIALIZATION MUSCLE-BP is based on CHAINBRIDGE and, therefore, must execute the framework's initialization steps:

- A deposit of at least $\$G_{max}$ funds is made to $\mathcal{W}_{MUSCLE}$, the wallet owned by $\mathcal{S}_{MUSCLE}$.
- $\mathcal{S}_{MUSCLE}$ calls $\mathcal{A}.Initialize()$ to initialize its algorithm $\mathcal{A}$. *Initialize* outputs three groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ of prime order p. Also, it outputs generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$, bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, and $H : \{0,1\}^* \rightarrow \mathbb{G}_1$, which is a one-way hash function that maps arbitrary strings to elements in $\mathbb{G}_1$.

Furthermore, all data sources execute algorithm *KeyGen* (Algorithm 1), which returns a private key $sk_i \in \mathbb{Z}_p$ and public key $pk_i = (g_1^{sk_i}, g_2^{sk_i})$ for each data source i.

Algorithm 1 Key generation algorithm of MUSCLE-BP.

```
1: function KeyGen(g1, g2)
2:     sk ←R Zp
3:     pk ← (g1^sk, g2^sk)
4:     return (pk, sk)
5: end function
```

REQUEST The requester calls $\mathcal{C}_{MUSCLE}$'s *Request* method, which is part of the CHAINBRIDGE framework. The parameters of this call are an array of strings *sources*, that refers to the endpoints of the data sources, and *callback*, which includes the address and function where

the data should be delivered. A string in *sources*, which we refer to as a label, is equal to the web address of the endpoint. The ordering of the strings determines the ordering of corresponding elements throughout the system. In other words, the first element of *sources* specifies data source 1, the retrieved data is referred to as m_1, and the signature as σ_1. We create this explicit ordering because verification of the aggregate signature requires a set ordering of the signatures. $\mathcal{C}_{MUSCLE}$ assigns an *id* to the request and stores it. Later, the request is detected by $\mathcal{S}_{MUSCLE}$. This phase corresponds to m_1 and m_2 of Figure 6.1.

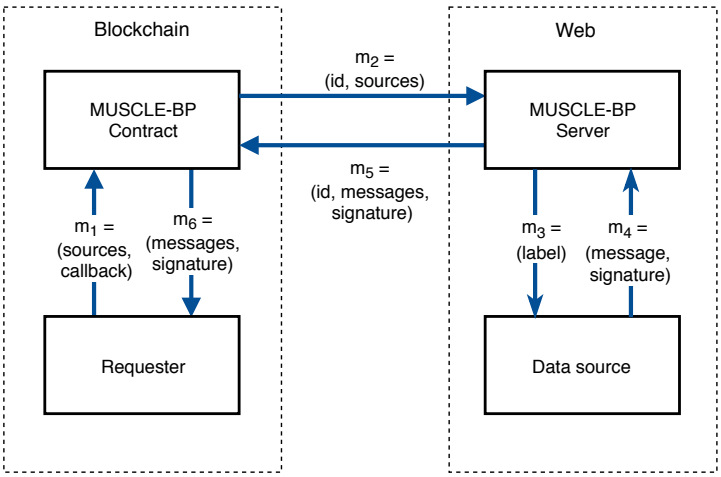


Figure 6.1: Data flows in MUSCLE-BP.

DATA RETRIEVAL Upon receiving the request, $\mathcal{S}_{MUSCLE}$ retrieves the data from the specified data sources. The label $s_i \in sources$ is included in the request to data source i. The data source signs the data using its private key through the *Sign* algorithm (Algorithm 2). Here, the data source first asserts whether the label is valid, i.e., equal to the requested endpoint. In Algorithm 2, we assume the data source can get the requested endpoint using a function called $GetEndpoint()$.

In addition to message m, the data source's public key and the label are signed. The public key is included because the original aggregate signature scheme only works if all messages are distinct. By concatenating the public key, we ensure that no two sources can have the same message. Bellare et al. [2] proved that data sources can securely contribute the same message if the public key is concatenated. Furthermore, the label is included to prevent attacks where a malicious $\mathcal{S}_{MUSCLE}$ retrieves data from a different endpoint of the same data source. Concatenation of the label makes the signature publicly verifiable in the context of the endpoint.

This phase corresponds to m_3 and m_4 of Figure 6.1. Note that while the figure shows only one data source, m_3 and m_4 are exchanged for every data source.

Algorithm 2 Sign algorithm of MUSCLE-BP.

```
1: function SIGN(m, s_i, pk, sk)
2:     if s_i! = GetEndpoint() then
3:         return false
4:     end if
5:     h ← H(pk||s_i||m)
6:     σ_m ← h^sk
7:     return σ_m
8: end function
```

AGGREGATION After messages $m_1, .., m_n$ are retrieved along with signatures $\sigma_{m_1}, .., \sigma_{m_n}$, server $\mathcal{S}_{MUSCLE}$ computes the aggregate signature Σ using the *Aggregate* algorithm (Algorithm 3). Using this signature, the authenticity of the data can be publicly verified.

Algorithm 3 Aggregation algorithm of MUSCLE-BP.

```
1: function AGGREGATE(σ_{m_1}, .., σ_{m_n})
2:     Σ ← 1
3:     for all σ_i ∈ {σ_{m_1}, .., σ_{m_n}} do
4:         Σ ← Σ · σ_i
5:     end for
6:     return Σ
7: end function
```

DELIVER Messages $m_1, .., m_n$ are sent to $\mathcal{C}_{MUSCLE}$'s *Deliver* method, along with signature Σ. The data is then forwarded to the address and function specified in *callback*. This phase corresponds to m_5 and m_6 of Figure 6.1.

VERIFICATION After receiving the response, the requester stores the proof to facilitate verification by any interested entity. The authenticity of the data can be publicly verified using the $Verify$ algorithm (Algorithm 4), using signature Σ and the arrays $messages = [m_1, .., m_n]$, $sources = [s_1, .., s_n]$, and the public keys. Note that these variables have all been publicly recorded into the blockchain as they were part of messages in the preceding phases. Also, we assumed that the public keys are publicly available, i.e., for each data source i, the algorithm has access to $pk_i = (g_1^{sk_i}, g_2^{sk_i})$. In the algorithm, we use pk to denote the array of public keys.

6.3 SECURITY ANALYSIS

MUSCLE-BP is built on the publicly verifiable aggregate signature scheme by Boneh et al. [5], which has been proven to be secure in its respective paper and by Bellare et al. [2], who proved that a check in the

Algorithm 4 Verification algorithm of MUSCLE-BP.

function VERIFY($e, g_2, messages, sources, pk, \Sigma$)
  $n \leftarrow |sources|$
  for $i = 1, i < n, i{+}{+}$ **do**
    $h_i = H(pk_i||s_i||m_i)$
  end for
  if $e(\Sigma, g_2) == \prod_{i=1}^{n} e(h_i, pk_i)$ **then**
    return true
  else
    return false
  end if
end function

verification algorithm can be removed by concatenating the public key to the message. Therefore, we refer the reader to these research papers for security analyses and proofs.

We note that our modification of adding the labels does not impact the security of the scheme, as the labels are simply concatenated to the message itself. Therefore, we can transform the modified scheme into the original scheme by defining the message of signer i as $m_i = s_i||m_i^{'}$, where s_i is the label and $m_i^{'}$ is the actual message.

6.4 COMPLEXITY ANALYSIS

We evaluate the computational and communication complexities of MUSCLE-BP. Both complexities depend on a number of variables, which are described in Table 6.2.

Table 6.2: Symbols used in the complexity analysis.

Symbol	Description
n	Number of data sources
m	Size in bits of a message m_i
e	Size in bits of an element in $\mathbb{G}_1$
s	Size in bits of a string in Ethereum

6.4.1 *Computational complexity*

To analyse the computational complexity MUSCLE-BP, we assess the number of cryptographic operations performed per invocation of every algorithm. The computational complexity of each algorithm is given in Table 7.3. The *KeyGen* algorithm, which is run by each data source during the Initialization phase, consists of generating a random element and one exponentiation. The complexity remains constant and

does not depend on any variable. The *Sign* algorithm comprises the computation of hash function H and one exponentiation. Similar to *KeyGen*, the number of operations remains constant. The *Aggregate* algorithm computes the product of a number of elements equal to the number of data sources, which means that the computational complexity of this algorithm is linear in the number of data sources. Similarly, the computational complexity of the $Verify$ algorithm also depends linearly on the number of sources, as the algorithm first recomputes the n hashes and then verifies the signature using $n + 1$ pairings of which n are combined through multiplication.

Table 6.3: Computational complexity for MUSCLE-BP.

Operation	Data source	Off-chain server	Verifier
KeyGen	$\mathcal{O}(1)$		
Sign	$\mathcal{O}(1)$		
Aggregate		$\mathcal{O}(n)$	
Verify			$\mathcal{O}(n)$

6.4.2 *Communication complexity*

We analyse the communication complexity of MUSCLE-BP by listing the number of bits transmitted by each party as part of a transaction per phase. The communicational complexity depends on a number of variables, which are described in Table 6.4. The Initialization, Aggregation, and Verification phases are omitted as no communication takes place.

REQUEST During the Request phase, the requester sends to $\mathcal{C}_{MUSCLE}$ two variables, namely *sources* and *callback*. *sources* is an array of n strings, where n is the number of data sources. Furthermore, *callback* is a combination of two strings that specify the address and the function of the smart contract to which the data is returned. The total amount of data that is sent in the transaction is, therefore, equal to $n + 2$ strings.

DATA RETRIEVAL In the Data Retrieval phase, $\mathcal{S}_{MUSCLE}$ retrieves data and signatures from all n data sources. When retrieving data from data source i, $\mathcal{S}_{MUSCLE}$ sends label $s_i \in sources$. Therefore, the total communication complexity is $n \cdot s$. Upon request, a data source responds with a message of size m bits and a signature of size e bits.

DELIVER In this phase, the oracle's off-chain server sends the retrieved messages and the aggregate signature to the blockchain. The labels, which are also needed to perform verification, do not need

to be sent as part of the transaction, as they were created by the requester. Aggregate signature Σ is a single element in $\mathbb{G}_1$. The total payload sent by $\mathcal{S}_{MUSCLE}$ to $\mathcal{C}_{MUSCLE}$ is, therefore, equal to the size of an element in $\mathbb{G}_1$ and the retrieved data.

Table 6.4: Communicational complexity for MUSCLE-BP.

Phase	Requester	Off-chain server	Data source
Request	$s \cdot (n+2)$		
Data Retrieval		$n \cdot s$	$m + e$
Deliver		$(n \cdot m) + e$	

6.5 CONCLUSION

In this chapter, we presented MUSCLE-BP, an oracle designed to reduce costs of data retrieval in a multi-source setting. MUSCLE-BP satisfies all the requirements defined in Chapter 5. Data sources are asked only to sign their data using the BLS signature scheme [4] and do not have to verify intermediate signatures or perform other computations. Also, we modified the scheme also to include the endpoint that was requested to prevent attacks in which a malicious $\mathcal{S}_{MUSCLE}$ retrieves data from a different endpoint of the same data source. The resulting signatures can publicly be verified at any point in the future in a non-interactive manner using the aggregate signature, the messages, and the labels. On top of that, we ensure gas sustainability and fair expenditure by building the oracle using CHAINBRIDGE. MUSCLE-BP is divided into six phases. After being initialised, requests can be sent to MUSCLE-BP. The requested data is retrieved from the specified data sources, and the oracle's server component aggregates the signatures. Finally, the data is delivered to the requester and verified.

7

MUSCLE-TP: MULTI-SOURCE ORACLE BASED ON TRAPDOOR PERMUTATIONS

In this chapter, we present MUSCLE-TP (MUlti-Source oraCLE - Trapdoor Permutations) according to the specification given in Chapter 5. MUSCLE-TP achieves a similar goal as MUSCLE-BP from Chapter 6. This chapter is organized as follows. First, the properties of the aggregate signature scheme that forms the basis of MUSCLE-TP are introduced. Then, the design of MUSCLE-TP is presented, after which we evaluate the computational and communicational complexity. An assessment of MUSCLE-TP's performance, in terms of gas costs, is deferred to Chapter 8, where we compare a sample implementation of the oracle with four other oracles.

7.1 AGGREGATE SIGNATURE SCHEME

MUSCLE-TP employs a sequential aggregate signature scheme based on trapdoor permutations by Brogle et al. [6], which we call BGR. Compared to the BGLS scheme used by MUSCLE-BP, the aggregate signature is created sequentially, i.e., the data sources are contacted in-order, and each data source adds its own signature on the intermediate aggregate signature produced by all previous signers, which we refer to as the *aggregate-so-far*. Furthermore, the way that lazy verification is provided makes that the signature grows slightly with the number of signers, while aggregate signatures produced by MUSCLE-BP are of constant size. The reason for this slight growth is that each signer adds a random string to the hash and includes it with the signature to provide lazy verification. Similar to MUSCLE-BP, we modify the scheme by concatenating a description of the requested endpoint, called a label, to the message to be able to verify signatures in the context of that specific endpoint.

7.2 SYSTEM DESIGN

MUSCLE-TP consists of two components, an on-chain smart contract $\mathcal{C}_{MUSCLE}$ and an off-chain server $\mathcal{S}_{MUSCLE}$, and is divided into five phases: Initialization, Request, Aggregation, Deliver, and Verification. We describe each of these phases in the following subsections. An explanation of the symbols used is found in Table 7.1.

INITIALIZATION All data sources execute the key algorithm associated with the trapdoor permutation π, which returns a public and

Table 7.1: Description of symbols used in MUSCLE-TP.

Symbol	Description
$\mathcal{C}_{MUSCLE}$	The on-chain component of MUSCLE-TP
$\mathcal{S}_{MUSCLE}$	The off-chain component of MUSCLE-TP
m_i	Message m provided by data source i
l_r	Length in bits of the randomness appended by each signer
π_i	Trapdoor permutation, operating on l_π-bit strings, that acts as public key
π_i^{-1}	Inverse of π_i, acting as private key
H	Cryptographic hash function that outputs l_H-bit strings
G	Cryptographic hash function that outputs l_π-bit strings
l_R	Number of randomly generated bytes from each signer
σ_m	Signature of message m
ϵ	The empty string for which we assume $\epsilon \oplus x = x$ for any x

private key pair for each data source i. The specific way of generating the key pair depends on the trapdoor permutation function that MUSCLE-TP is instantiated with.

Furthermore, MUSCLE-TP is based on CHAINBRIDGE. Therefore, CHAINBRIDGE's initialization steps are executed upon initialization:

- A deposit of at least $\$G_{max}$ funds is made to $\mathcal{W}_{MUSCLE}$, the wallet owned by $\mathcal{S}_{MUSCLE}$.
- $\mathcal{S}_{MUSCLE}$ calls $\mathcal{A}.Initialize()$ to initialize its algorithm $\mathcal{A}$. *Initialize* outputs constants l_π, l_H, l_R and cryptographic hash functions $H : \{0,1\}^* \to \{0,1\}^{l_H}$ and $G : \{0,1\}^* \to \{0,1\}^{l_\pi}$.

REQUEST The requester calls $\mathcal{C}_{MUSCLE}$'s *Request* method, which is part of the CHAINBRIDGE framework. The parameters of this call are an array of strings, called *sources*, that refers to the data sources and *callback*, which includes the address and function where the data should be delivered to. A string in *sources*, referred to as the *label* of that data source, is equal to the requested endpoint. The ordering of the labels determines the ordering of corresponding elements throughout the system. In other words, the first label in *sources* specifies data source 1, the retrieved data is referred to as m_1, and the signature as σ_1. As MUSCLE-TP is based on a sequential aggregate signature scheme, the ordering is also used during the aggregation by $\mathcal{S}_{MUSCLE}$ as the order in which the data sources are contacted and during verification by the requester. Following the CHAINBRIDGE framework, $\mathcal{C}_{MUSCLE}$ assigns an *id* to the request and stores it. Later, the

request is detected by $\mathcal{S}_{MUSCLE}$. This phase corresponds to m_1 and m_2 of Figure 7.1.

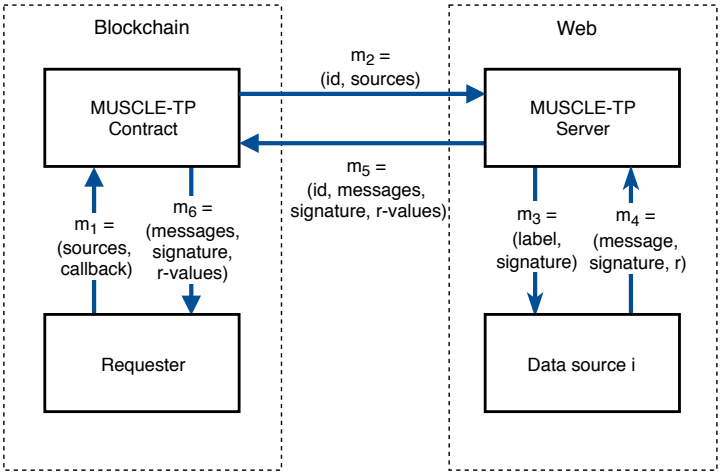


Figure 7.1: Data flows in MUSCLE-TP.

AGGREGATION $\mathcal{S}_{MUSCLE}$ retrieves the data from the specified data sources based on the array *sources* contained in the request. The label $s_i \in sources$ is included in the request to data source i. Due to the sequential nature of the underlying aggregate signature scheme, the signature obtained from data source i is part of the request sent to data source $i+1$, which aggregates its signature onto the signature of data source i. The data sources sign the data using their private key through the *AggregateSign* algorithm (Algorithm 5), after asserting whether the label is valid, i.e., equal to the requested endpoint.

For data source i, the algorithm takes as input label $s_i \in sources$ and the message m_i that is retrieved, along with public key π_i, private key π_i^{-1}, and the aggregate signature $\sigma_{i-1} = (x_{i-1}, h_{i-1})$ formed by data sources 1 to $i-1$. The label is included to prevent attacks where a malicious $\mathcal{S}_{MUSCLE}$ retrieves data from a different endpoint of the same data source. In Algorithm 5, we assume the data source can obtain the current endpoint using a function $GetEndpoint()$. In addition to the new aggregate signature $\sigma_i = (x_i, h_i)$, the algorithm also returns a value r_i, consisting of l_R bytes. The value r_i of each data source i is not passed to the next data source. Instead all r-values are sent to the blockchain in the next phase as these values are needed during the verification.

This phase corresponds to m_3 and m_4 of Figure 7.1. Note that the figure shows only one data source, m_3 and m_4 are exchanged for every data source.

DELIVER Messages $[m_1, .., m_n]$ are sent to $\mathcal{C}_{MUSCLE}$'s *Deliver* method, along with signature σ_n and values $[r_1, .., r_n]$. Note that including the r-values is what causes the total payload to be linearly dependent on

Algorithm 5 AggregateSign algorithm of MUSCLE-TP.

function AGGREGATESIGN($m_i, s_i, \sigma_{i-1}, \pi_i, \pi_i^{-1}, H, G$)
  if $s_i! = GetEndpoint()$ **then**
    return false
  end if
  if $i == 1$ **then** ▷ If this is the first signer.
    $x_{i-1} \leftarrow \epsilon$
    $h_{i-1} \leftarrow \epsilon$
  else
    $(x_{i-1}, h_{i-1}) \leftarrow$ **parse** σ_{i-i}
  end if
  $r_i \xleftarrow{R} \{0,1\}^{l_r}$
  $h_i \leftarrow h_{i-1} \oplus H(\pi_i, m_i, s_i, r_i, x_{i-1})$
  $x_i \leftarrow \pi_i^{-1}(G(h_i) \oplus x_{i-1})$
  $\sigma_i \leftarrow (x_i, h_i)$
  return r_i, σ_i
end function

the number of data sources, while the signature itself is of constant size. The data is then forwarded to *callback*. This phase corresponds to m_5 and m_6 of Figure 7.1.

VERIFICATION After receiving the response, the requester stores the data to facilitate verification by any interested entity. The authenticity of the data can be publicly verified by calling the $Verify$ algorithm (Algorithm 6), using signature σ_n, the arrays $messages = [m_1, .., m_n]$, $r = [r_1, .., r_n]$, and $sources = [s_1, .., s_n]$, along with the public keys of the data sources. Note that these variables have all been publicly recorded into the blockchain in the preceding phases. Also, we assumed that the public keys are publicly available, i.e., for each data source i, the algorithm has access to each public key $\pi_i \in \Pi$.

7.3 SECURITY ANALYSIS

According to the security definition of (sequential) aggregate signature schemes [5, 18], each signer is individually secure against existential forgery when the adversary is allowed to request signatures for messages of its choosing adaptively, and may also determine the aggregate-so-far upon which the signature is aggregated. The security of the scheme by Brogle et al. [6] has been proven to be secure under this definition in the Random Oracle Model in its respective paper. Therefore, we refer the reader to this research paper for security analyses and proofs.

Similar to MUSCLE-BP, adding the labels to the scheme does not impact the security of the scheme, as the labels are simply concaten-

Algorithm 6 Verification algorithm of MUSCLE-TP.

function VERIFY($messages, sources, r, \Pi, \sigma_i, H, G$)
  $n \leftarrow |sources|$
  $(x_{i-1}, h_{i-1}) \leftarrow$ **parse** σ_{i-i}
  for $i = n, n-1, .., 2$ **do**
    $x_{i-1} \leftarrow G(h_i) \oplus \pi_i(x_i)$
    $h_{i-1} \leftarrow h_i \oplus H(\pi_i, m_i, s_i, r_i, x_{i-1})$
  end for
  if $h_1 == H(\pi_1, m_1, s_1, r_1, \epsilon)$ & $\pi_1(x_1) == G(h_1)$ **then**
    return true
  else
    return false
  end if
end function

ated to the message itself. Therefore, we can transform the modified scheme into the original scheme by defining the message of signer i as $m_i = m_i^{'}||s_i$, where $m_i^{'}$ is the actual message and s_i is the label.

7.4 COMPLEXITY ANALYSIS

We evaluate MUSCLE-TP by providing a theoretical analysis of the computational and communicational complexities, which both depend on a number of variables. These variables are described in Table 7.2.

Table 7.2: Symbols used in the complexity analysis.

Symbol	Description
n	Number of data sources
m	Size in bits of a message m_i
l_π	Size in bits of the strings permutation π operates on
l_H	Size in bits of the output of hash function H
l_r	Size in bits of the randomness appended by each signer
s	Size in bits of a string in Ethereum

7.4.1 *Computational complexity*

To analyse the computational complexity MUSCLE-TP, we assess the number of cryptographic operations performed per invocation of every algorithm. The computational complexity of each algorithm is given in Table 7.3. The *AggregateSign* algorithm comprises the generation of a random bit string, one call to both functions H and G, two bitwise XOR operations, and one exponentiation. Therefore the number of op-

erations remains constant. On the other hand, the computational complexity of the $Verify$ algorithm depends linearly on the number of sources. The algorithm performs $n-1$ iterations in which it performs operations similar to those in the *AggregateSign* algorithm. Then, the algorithm calls both H and G once and performs an exponentiation and an equality check.

Table 7.3: Computational complexity for MUSCLE-TP.

Operation	Data source	Verifier
AggregateSign	$\mathcal{O}(1)$	
Verify		$\mathcal{O}(n)$

7.4.2 *Communication complexity*

We analyse the communication complexity of MUSCLE-TP by listing the number of bits transmitted by each party during each phase. The communicational complexity depends on a number of variables and is given in Table 7.4. The Initialization and Verification phases are omitted as no communication takes place.

REQUEST During the Request phase, similar to what we have seen with MUSCLE-BP, the requester sends to $\mathcal{C}_{MUSCLE}$ two variables, namely *sources* and *callback*. *sources* is an array of n strings, where n is the number of data sources. Furthermore, *callback* is a combination of two strings that specify the contract address and the function to which data is returned. The total amount of data that is sent in the transaction is, therefore, equal to $n+2$ strings.

AGGREGATION During the Aggregation phase, $\mathcal{S}_{MUSCLE}$ sequentially retrieves data from the data sources by sending the aggregate-so-far, which is of size $l_\pi + l_H$ bits. The data sources then adds their own signature on the aggregate-so-far and return the resulting $l_\pi + l_H$ bits aggregate signature (x_i, h_i), in addition to the actual message of size m bits and the random value $r_i \in \{0,1\}^{l_r}$.

DELIVER MUSCLE-TP's off-chain component sends the aggregate signature to the blockchain, along with the messages and the randomly generated values. Note that we assumed that the public keys are available to any entity, regardless of whether it resides on-chain or off-chain. Aggregate signature σ_n comprises $x_n \in \{0,1\}^\pi$ and $h_n \in \{0,1\}^H$, while the n random values are each l_r bits long.

Table 7.4: Communicational complexity for MUSCLE-TP.

Phase	Requester	Off-chain server	Data source
Request	$s \cdot (n+2)$		
Aggregation		$l_\pi + l_H$	$l_\pi + l_H + l_r + m$
Deliver		$l_\pi + l_H + n \cdot l_r + n \cdot m$	

7.5 CONCLUSION

In this chapter, we presented MUSCLE-TP as a means of reducing the costs in a multi-source setting by employing a sequential aggregate signature scheme by Brogle et al. [6]. MUSCLE-TP satisfies all the requirements defined in Chapter 5. The scheme supports lazy verification to ensure that data sources are asked only to sign their message and do not have to verify intermediate signatures. Also, we modified the scheme also to include the endpoint that was requested. In that way, we prevent attacks where a malicious $\mathcal{S}_{MUSCLE}$ retrieves data from a different endpoint of the same data source. The resulting signatures can publicly be verified at any point in the future in a non-interactive manner using the aggregate signature, the messages, and the labels. On top of that, we ensure gas sustainability and fair expenditure by building the oracle using CHAINBRIDGE. MUSCLE-TP is divided into five phases. After being initialised, requests can be sent to MUSCLE-TP. The requested data is retrieved from the specified data sources, and the signatures are aggregated. Finally, the data is delivered and verified.

8

PERFORMANCE ANALYSIS

In this chapter, we evaluate the performance of MUSCLE-BP and MUSCLE-TP, in terms of gas expenditure, based on a set of measurements obtained from proof-of-concept implementations. We compare these measurements with that of four other oracles, of which two oracles are based on TLS-N [24], which represents the current state of the art. The other two oracles are based on bilinear pairings and trapdoor permutation, similar to MUSCLE-BP and MUSCLE-TP. However, these oracles do not apply aggregation, which allows us to assess the effect of aggregate signatures. While we expect that their transaction and storage costs are relatively higher, we want to see how the total costs compare.

To this end, we first describe how we instantiate and implement the six oracles. Then, we describe the data that we use. Finally, we examine the total costs as well as that of the components separately.

8.1 INSTANTIATION

In the subsections below we describe how we instantiated and implemented the six oracles.

MUSCLE-BP The groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are cyclic groups on the elliptic curve *alt bn128* defined by the curve equation $Y^2 = X^3 + 3$. The group $\mathbb{G}_1$ is a cyclic group of prime order p on the above curve over the field $\mathbb{F}_p$ with generator g_1, while group $\mathbb{G}_2$ is a cyclic group of prime order in the same elliptic curve over a different field $\mathbb{F}_{p^2} = \mathbb{F}_{p[X]}/(X^2 + 1)$, where p is the same as above, with generator g_2. The specific values for p, g_1, and g_2 are given in Table 8.1. Furthermore, we define the hash function $H : \{0,1\}^* \to \mathbb{G}_1$ as $H(x) = keccak256(x)$, where *keccak*256 is the hash function natively supported by Ethereum. These values are chosen because Ethereum has a pre-compiled contract to perform pairing checks over the *alt bn128* curve. We implement the $Verify$ algorithm in Solidity[1], while the *KeyGen*, *Sign*, and *Aggregate* algorithms are implemented in Go[2]. Implementations of all algorithms are based on existing code by a GitHub user called Project-Arda[3].

1 https://github.com/ethereum/solidity
2 https://golang.org/
3 https://github.com/Project-Arda

Table 8.1: Values used in the instantiation of MUSCLE-BP.

Symbol	Value
p	21888242871839275222246405745257275088696311157297823662689037894645226208583
g_1	$(1, 2)$
g_2	$(11559732032986387107991004021392285783925812861821192530917403151452391805634 \cdot i + 10857046999023057135944570762232829481370756359578518086990519993285655852781, 40823678758634336813322034031454355683168513275934012081057410762141200935 31 \cdot i + 8495653923123431417604973247489272438418190587263600148770280649306958101930)$

MUSCLE-TP Similar to Brogle et al. [6], we instantiate the permutation π with RSA with 2048-bit key length and public exponent 65537. A key length of 2048 is considered sufficiently secure by the NIST until 2030[4]. However, while Brogle et al. instantiate the hash functions H and G using $SHA256$, we use $Keccak-256$, as implemented by the *go-solidity-sha3* package[5]. We do this because $Keccak-256$ costs less gas to perform. Using this instantiation, the parameters become $l_\pi = 2048$, $l_H = 256$, and $l_r = 128$. Moreover, RSA has slightly different domains for different signers as their moduli differ. To deal with this, the authors propose a solution in [6, Appendix F] which adds one bit of information per signer. We implement the $Verify$ algorithm in Solidity[6] and the $AggregateSign$ algorithm in Go[7].

TLS-N We compare our designs to the oracle presented by Ritzdorf et al., called TLS-N (see Section 3.2). To our best knowledge, TLS-N is currently the only other oracle that assumes modifications at the data sources. TLS-N is a single source oracle and, as such, the number of transactions performed is linearly dependent on the amount of data sources.

MULTI-TLS-N As noted before, we can trivially improve TLS-N for operating in a multi-source scenario by merely putting data from all sources, along with required proofs, in a single transaction. We refer to this construction as MULTI-TLS-N, which we also include in our comparison.

4 https://www.keylength.com
5 miguelmota/go-solidity-sha3
6 https://github.com/ethereum/solidity
7 https://golang.org/

BLS The non-aggregated variant of MUSCLE-BP is instantiated similarly, but sends the separate BLS signatures instead of the aggregated BGLS signature.

RSA The performance of MUSCLE-TP is put into perspective using an oracle that is based on the RSA PKCS#1 version 1.5 standard, as implemented in Go's *crypto/rsa* package, and, similar to MUSCLE-TP, uses a 2048-bit key length and public exponent 65537.

8.2 EXPERIMENT

We measure the gas expenditure of the six oracles in scenarios with one to ten signers, which we view as sufficient as the expenditure increases in a linear manner. In our experiment, we define the total gas expenditure as the sum of the transaction costs made to send the data on-chain, the storage costs to store the messages and the proofs for later verification, and the verification costs. One could argue that, in practice, multiple entities need to verify the authenticity. As the number of performed verifications grow, at some point the RSA oracle, which has the lowest verification cost, achieves the lowest total costs. However, the presented oracles can be instantiated in a way that ensures verification only has to be performed once. In CHAINBRIDGE, the smart contract that the data is returned to and the smart contract making the request does not need to be the same smart contract. One could, therefore, create a smart contract specially made to receive and verify messages and signatures. Such smart contract only stores a message if the verification was successful.

The smart contracts of our six systems are deployed on Ganache[8], a JavaScript implementation of the Ethereum blockchain. Also, to perform a fair comparison of the six oracles introduced above, the messages used should be the same. The authors of TLS-N implemented a smart contract, called *BTCPriceFeed*[9], for retrieving the BTC-USD price based on the endpoint *https://index.bitcoin.com/api/vo/lookup?time=1483228800*). The data is 390 bytes in size and depicted in Figure 8.1.

8.3 RESULTS

We now present the results of our measurements of each scheme's transaction, storage, verification, and total cost as a function of the number of data sources, followed by a discussion.

8 http://truffleframework.com/ganache
9 https://github.com/tls-n/BTCPriceFeed

```
{
  "open": {
    "price": 95917,
    "time": {
      "unix": 1483142400,
      "iso": "2016-12-31T00:00:00.000Z"
    }
  },
  "close": {
    "price": 96760,
    "time": {
      "unix": 1483228800,
      "iso": "2017-01-01T00:00:00.000Z"
    }
  },
  "lookup": {
    "price": 96760,
    "k": 1,
    "time": {
      "unix": 1483228800,
      "iso": "2017-01-01T00:00:00.000Z"
    }
  }
}
```

Figure 8.1: Structure of the data that is used for analysis.

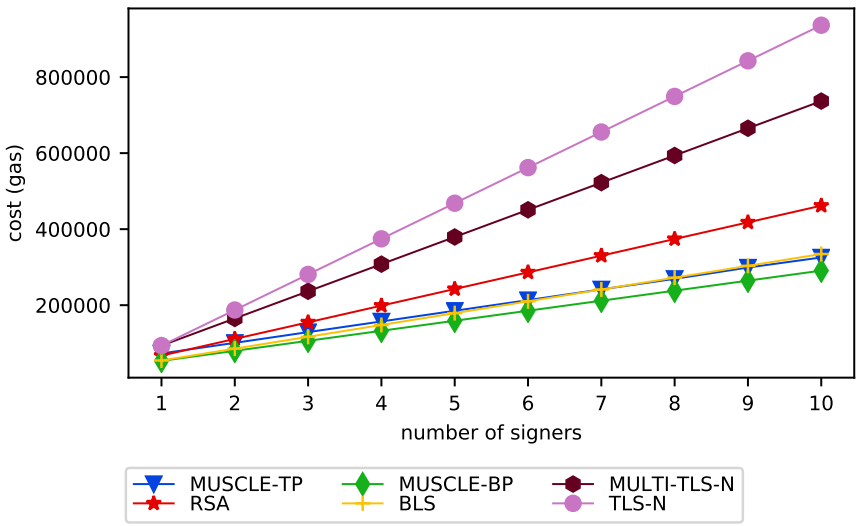


Figure 8.2: Comparison of transaction costs, measured in gas.

8.3.1 *Transaction cost*

To measure the transaction costs, we implemented functions that take in the same input as the verification function of each smart contract, but differ in that they execute only the *Revert* operation. Invoking this operation halts execution and returns all remaining gas. As we are calling it immediately, we receive back all gas, except the amount that was spent on the transaction. Recall from Section 2.2.5 that the transaction cost is 4 gas per zero-byte and 68 gas per non-zero byte, in addition to a base cost of 21.000 gas.

Figure 8.2 shows the amounts of gas spent on transaction costs. If there is only one signer, we see that TLS-N and MULTI-TLS-N perform equally, which was expected because they are effectively the same system in this scenario. We observe that the difference between TLS-N and MULTI-TLS-N expands by approximately 21.000 gas with every additional signer, equal to the base cost of performing a transaction.

For the RSA-based oracles, the difference in gas expenditure per additional signer lies between 14.053 and 16.173 gas. This difference can be explained by examining the transmitted data. For each signer, the RSA oracle sends a 256-byte signature and an extra message, while MUSCLE-TP sends a message, a 16-byte random value, and a 1-byte boolean value, so the difference is $256 - 17 = 239$ bytes. Therefore, the additional gas cost per signer is at most $239 \cdot 68 = 16.252$, assuming all transmitted bytes are non-zero.

Finally, we examine MUSCLE-BP and BLS, which perform equally in case of one signer as their payload is then equal. However, as the number of signers increases, we see that the cost of MUSCLE-BP grows 4.730 gas less per added signer. This was expected as the BLS oracle sends

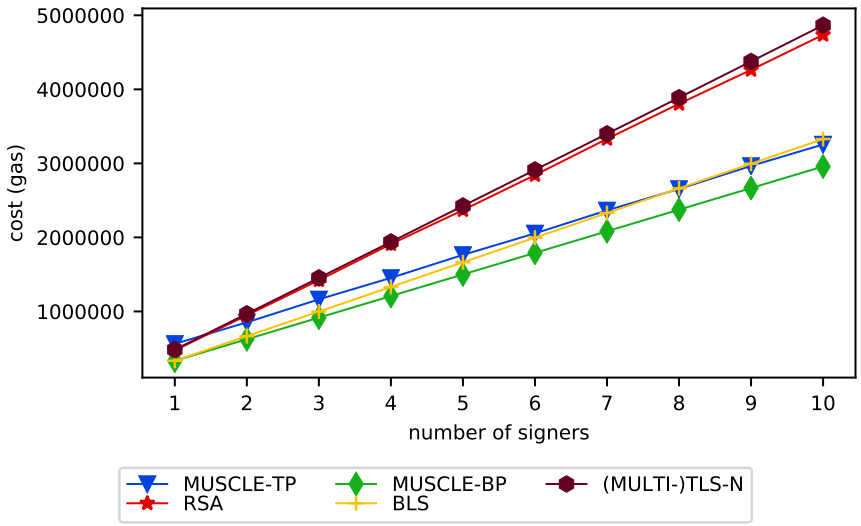


Figure 8.3: Comparison of storage costs, measured in gas.

an additional element in $\mathbb{G}_1$ per additional signer, while MUSCLE-BP only requires one element in $\mathbb{G}_1$. In our implementation, an element in $\mathbb{G}_1$ is represented as two 32-byte integers. Therefore, the difference in cost is equal to $(2 \cdot 32) \cdot 68 = 4352$ gas, increased by the cost of sending the single element by MUSCLE-BP, where the latter decreases as the number of signers grows.

Overall, we observe that MUSCLE-BP performs best in all cases. The BLS oracle comes in second place for less than 9 signers but is outperformed by MUSCLE-TP onwards. Also, the TLS-N-based oracles perform worse than the other four oracles.

8.3.2 *Storage cost*

We measure the storage costs by storing the sent data in dictionary-like data structures called mappings, where a separate mapping is created for each data type. The measurements are depicted in Figure 8.3. TLS-N and MULTI-TLS-N show the same storage costs, as these oracles only vary in the way the data is sent to the blockchain. For the rest, we observe a pattern similar to that of the transaction costs, which was expected as both depend on the size of the data.

8.3.3 *Verification cost*

To measure the verification costs, we implemented functions that do not take any input but execute the verification on data that is stored in the smart contract beforehand. Using that approach, the measured gas expenditure comprises the base cost of a transaction (21.000 gas)

and the cost of running the verification algorithm. By subtracting the base cost, we obtain the results shown in Figure 8.4.

Similar to the storage costs, TLS-N and MULTI-TLS-N expend the same amount of gas on verification costs. The most significant component of the verification cost is verification of the signature. The difference in costs in comparison with the other oracles can be explained by a design decision the authors made to use an elliptic curve that is generally supported by TLS implementations but not by Ethereum. As such, elliptic curve operations needed to be implemented as a library instead, while the other four oracles rely on pre-compiled contracts for their most expensive operations, i.e., modular exponentiations and pairing checks. This causes the cost of the TLS-N-based oracles to grow significantly faster than the cost made by the other four oracles.

When comparing MUSCLE-TP with RSA, we see that the verification costs are initially almost equal with MUSCLE-TP spending 3.396 gas more. However, as the number of signers increases, we observe that performing verification on the non-aggregated signatures of RSA is more efficient. The difference in verification costs for multiple signers can be explained by looking at the verification algorithms. RSA mainly needs to perform a modular exponentiation, while MUSCLE-TP also performs operations for hashing, bitwise XOR, and equality tests. Also, we observe that MUSCLE-TP's verification costs do not show a completely straight line, which is caused by the split operation [6, Appendix F] that was added in the verification algorithm as MUSCLE-TP is instantiated with RSA.

For the oracles based on bilinear pairings, we observe that BGLS performs better than its non-aggregated variant BLS, which performs worse than the RSA-based oracles. More specifically, for each additional signer, MUSCLE-BP expends around 179.700 gas less than BLS, and this difference slowly becomes smaller. The difference in costs can be explained by considering that the gas cost for a pairing check in Ethereum is equal to $80.000 \cdot k + 100.000$, where k is the number of points. BLS needs a separate call for each signature, and each call entails two points, and so the verification cost per signature is $80.000 \cdot 2 + 100.000 = 260.000$ gas. On the other hand, MUSCLE-BP can verify equality of the $n + 1$ bilinear pairings in a single call. Therefore, $80.000 \cdot (n + 1) + 100.000 = 80.000 \cdot n + 180.000$ gas is paid for n signers. The difference in gas per additional signer is then equal to $260.000 - 80.000 = 180.000$ gas, which matches with our findings.

In total, the RSA oracle expends the least amount of gas in verifying signatures, followed by MUSCLE-BP. Moreover, MUSCLE-TP consumes more than MUSCLE-BP and both non-aggregated variants while being more cost-efficient than the TLS-N-based oracles.

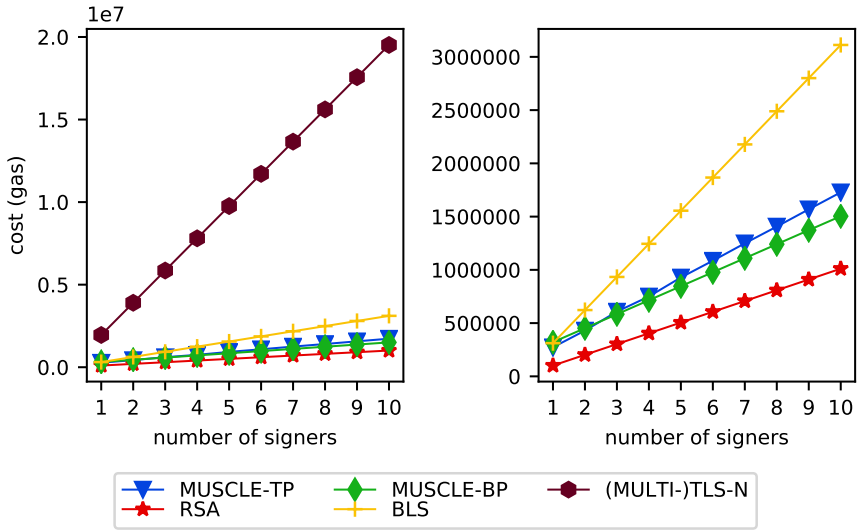


Figure 8.4: Comparison of verification costs, measured in gas.

8.3.4 *Total cost*

The total costs, which is defined as the sum of its components, of the six oracles is given in Figure 8.5, where we see that TLS-N and MULTI-TLS-N differ only due to the base cost of the transactions, which saves MULTI-TLS-N approximately 21.000 gas for each additional signer. Furthermore, non-aggregated variants are outperformed by MUSCLE-BP if more than one signer is involved, while MUSCLE-TP outperforms them for more than two signers. Overall, MUSCLE-BP boasts the lowest total cost.

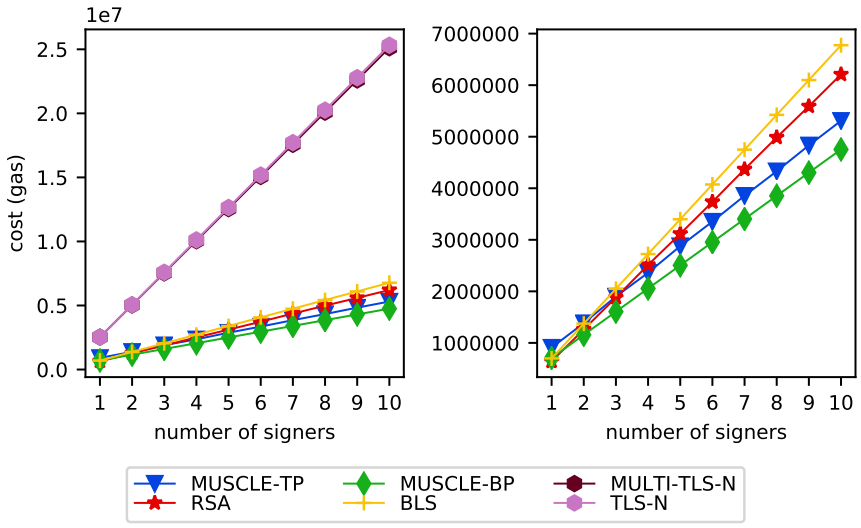


Figure 8.5: Comparison of total costs, measured in gas.

8.3.5 *Discussion*

Comparing to TLS-N and MULTI-TLS-N, MUSCLE-BP and MUSCLE-TP achieve lower costs for sending and storing the transaction, as well as for verifying the proof. Also, we constructed two oracles, referred to as BLS and RSA, that differ from, respectively, MUSCLE-BP and MUSCLE-TP in that no aggregation is performed upon the signatures. When comparing the RSA oracle with MUSCLE-TP, we found that due to the aggregation, MUSCLE-TP is more efficient in terms of transaction costs, while RSA is more cost efficient at verification. Moreover, we observed that MUSCLE-BP performs equally to BLS when considering a single singer. However, in a multi-source setting, MUSCLE-BP overtakes BLS in transaction, storage, and verification costs.

When we look at the total costs, we observe that oracles based on RSA and bilinear pairings perform better than TLS-N-based solutions, which represent the current state of the art. Also, our proposed oracles perform better than their non-aggregated variants in scenarios with multiple signers, with MUSCLE-BP performing best overall.

9 DISCUSSION AND FUTURE WORK

Many smart contract applications rely on external data. However, resides outside the blockchain. While traditional web applications can communicate directly with trustworthy data sources through the Internet, this is not possible for smart contracts because their execution must be deterministic. The need for determinism exists because a blockchain is replicated among all nodes of the network. Any time a node starts at the start state, called the genesis block, and applies the transactions of each block consecutively, it should always arrive at the same result, as disagreement among nodes about the blockchain's state could cause the network to split. As a consequence, smart contracts cannot make requests to the world external to the blockchain, as this would introduce non-determinism.

Bringing external data into the blockchain has been a topic of research since Buterin [7] first introduced Ethereum, and a system that achieves external data retrieval is called an oracle. The primary requirement in designing oracles is that the authenticity of data must be publicly verifiable, and several approaches to constructing oracles have been suggested in previous work, which all fall within one of three categories. Each category has its own advantages and disadvantages. Oracles that require software modifications at data sources are based on provable cryptography to guarantee the authenticity of retrieved data, but at the cost of flexibility as these oracles can only access data sources that implemented the required changes. On the other hand, oracles based on trusted hardware require no modifications to be made to the data sources, which makes them flexible because the oracle can provide information from any source and does not have to wait for that source to implement specific software changes. However, one must trust in trusted hardware technology, such as Intel SGX, to be secure. Finally, we have seen decentralised oracle networks in which the oracle is formed by a network of nodes. Similar to the trusted hardware oracles, this type does not require modifications at the data sources. Nonetheless, decentralised oracle networks are less efficient as the oracle must reach internal consensus about the answer to the query. Also, security is based on incentives instead of cryptography. Therefore, the security cannot be proven.

Previous work on oracles has mainly been focused on ensuring the authenticity of external data retrieved from a single source but has thus far not explored approaches for scenarios in which data comes from multiple sources. Using the current techniques, data would be

retrieved from each source separately, after which the data is stored and can be verified.

In this chapter, we revisit the main research question that we posed in the introductory chapter of this thesis:

How can we reduce the cost of retrieving external data for smart contracts from multiple sources, while ensuring that the authenticity of the data is publicly verifiable?

Below, we discuss how the presented oracles achieve the research goal and then provide future research directions by identifying remaining open problems and potential improvements.

9.1 DISCUSSION

In this thesis, two oracles for retrieving data from multiple sources are presented, that both employ aggregate signature schemes to compress the signature data of the data sources. We built both oracles using CHAINBRIDGE, a generalisation of Town Crier [32] which ensures gas sustainability and fair expenditure for an honest requester.

In Section 1.3, we identified three sub-questions of the main research question. In the first sub-question, we asked ourselves how we can provide external data to smart contracts in a way that the authenticity can be publicly verified. To answer this sub-question, we first performed a literature review and developed a generic oracle construction, which we presented in Chapter 1. During our research, we realised that verifiable authenticity of the data source alone is not enough. Instead, authenticity should be verified in the context of the specific endpoint, as a malicious oracle could otherwise send data retrieved from the same source, but with different parameters. To avert the attack, the signing algorithms of both MUSCLE-BP (Algorithm 2) and MUSCLE-TP (Algorithm 5) are designed in such a way that the data source first verifies that the label received from the requester via the oracle matches the requested endpoint. Also, this label is included in the signature and required during verification. As the requester created the labels, the oracle does not need to send the labels back to the blockchain and, hence, the use of labels does not increase the size of the transaction.

In the second sub-question, we inquired how we can design services, such as oracles, that guarantee gas sustainability and fair expenditure. Zhang et al. [32] have noted these properties as fundamental properties for any Ethereum service that operates partly on- and partly off-chain. As the two properties apply to any service and are, therefore, of independent interest, we designed a generic framework, called CHAINBRIDGE, as a separate contribution. We then used CHAINBRIDGE to build both oracles, which also helps us gain evidence of CHAINBRIDGE's usefulness for constructing complex services. CHAINBRIDGE is introduced in Chapter 4.

The two sub-questions above are related to the construction of oracles and how we can ensure certain properties. With the third sub-question, we examined existing solutions, and we asked ourselves how we can reduce the cost in a multi-source setting. Two factors influence the cost. The first is the size of the data, which determines the cost of transmitting the data to the blockchain and storing it, while the second factor is the cost of verifying the authenticity. In this work, we focused on the first, as transmission and storage of data are among Ethereum's most expensive operations [31].

From our sample implementations, we learned that in MUSCLE-BP and MUSCLE-TP, using signature aggregation, sending the data of one extra data source to the blockchain costs around 26.000 and 28.000 gas, respectively, while the additional costs for BLS and RSA are, respectively, roughly 31.000 and 43.000 gas. Finally, one would pay approximately 93.600 gas using TLS-N or 71.500 gas using MULTI-TLS-N for the same data. In other words, the growth in transaction costs per additional signer for our presented oracles less than 30% of TLS-N's growth, which represents the current state of the art. A similar pattern arises in measuring the storage costs, which was to be expected as both depend on the size of the data. The only difference with the transaction costs is that TLS-N and MULTI-TLS-N have the same storage costs per signer.

Regarding verification costs, TLS-N and MULTI-TLS-N are equal as they only differ in how they send data to the blockchain. These costs are both higher than that of the other four oracles. When comparing MUSCLE-TP with the RSA-based oracle, we observe that verification is less costly for the latter oracle. Both oracles perform modular exponentiation to verify the signature. Additionally, in MUSCLE-TP, verification of a single signature requires the computation of two hash functions, two equality checks, and one bitwise AND operation. Every further signature adds the computation of two hash functions and two bitwise XOR operations. On the other hand, the RSA-based oracle computes the hash of each message and verifies the padding using a number of inequality checks. Furthermore, MUSCLE-BP performs better than its non-aggregated variant BLS. Looking at their verification algorithms, the difference in costs can be explained as we need two pairing operations per signature for BLS, while MUSCLE-BP requires $n + 1$ pairing operations for n signatures.

Looking at the total cost, we observe that both MUSCLE-BP and MUSCLE-TP achieve lower costs than the non-aggregated variants and the TLS-N-based oracles in scenarios involving, respectively, more than three and four signers. Even though TLS-N is a more generic technique to provide non-repudiation as TLS-N allows to generate a proof over the contents of any TLS session with a TLS-N-enabled server, all examined oracles are based on the same underlying assumption that data sources are willing to sign the data they return.

9.2 FUTURE WORK

The presented oracles are, to the best of our knowledge, the first oracles that take into account the scenario where data is retrieved from multiple sources. Both MUSCLE-BP and MUSCLE-TP show promising results regarding gas expenditure. However, there is still room for improvement.

BATCH VERIFICATION We noted that the cost is influenced by two factors, namely the size of the data and the cost of verification. In this work, we focussed on the first, as transmission and storage of data is among Ethereum's most expensive operations. Future work could explore the savings that can be made by focussing on verification costs, for example through the use of batch verification. Where aggregate signature schemes save the communication overhead by compressing the signature size, the batch verification saves the computational overhead by providing the optimised verification process of many signatures [19].

VERIFICATION IN CONTEXT OF TIME In both oracles, the messages are verified in the context of the endpoint used to retrieve the data by including the endpoint in the corresponding signature to prevent a malicious oracle from delivering data retrieved from a different endpoint of the same source. The implicit assumption here is that the message returned by a specific endpoint is static, which is not always the case. Consider, for example, an endpoint that returns a list of users in a group. Over time, users can leave or join the group. One possible solution is to take a similar approach to how we included the endpoint context. However, such approach only makes sense if one knows when the endpoint's data was last updated. If the 'last update' time is available, a smart contract that wants to use the data could first retrieve the 'last update timestamp' and then decide whether to use the stored version or retrieve the new data. As the time stamp is generally smaller than the whole message, the smart contract can save costs using this approach, depending on the update frequency. Future work should explore ways to cope with situations where the response data is dynamic.

EXPLORING OTHER USE-CASES FOR CHAINBRIDGE In this work, we presented CHAINBRIDGE as a generalization of Town Crier [32] to serve as a foundation for MUSCLE-BP and MUSCLE-TP. However, as we noted, CHAINBRIDGE can be used to build any service comprising both on-chain and off-chain components. For instance, storage of data is among the most expensive operations in Ethereum. One approach for many types of applications is to save the data off-chain, us-

ing decentralised storage providers such as IPFS[1] and Swarm[2], while the blockchain only stores a reference to the data. As part of future work on decentralised storage, CHAINBRIDGE could be used to bridge between the on-chain and off-chain environment.

LOSSLESS COMPRESSION OF JSON The data contained in a transaction consists of messages and their corresponding proofs or signatures. The size, and thereby the cost, of a transaction, can be reduced by compressing either these components. In this research, we chose to focus on reducing the size of the signature data, because lossless compression of random data is believed to be impossible. While this decision makes our findings apply to any situation regardless of the data that is retrieved, in practice, most data sources return their data in JSON formatting. Future work should focus on the lossless compression of JSON messages to reduce the transaction costs even further. For example, JSON makes extensive use of quotes, which adds two bytes to every string. Also, JSON does not support the use of a schema by default. If the same message contains multiple objects of similar schema structure, the key names for each property are repeated, even though they are the same for each object.

In designing such solution, we recommend that the cost of decompressing the message on the blockchain should also be taken into account. As we have seen in our comparison of MUSCLE-TP and the RSA-based oracle in Chapter 8, the reduction achieved through aggregation was almost negated by the increased verification costs.

SYNCHRONIZED AGGREGATE SIGNATURES MUSCLE-BP and MUSCLE-TP are built using two aggregate signature schemes that we selected based on their support for lazy verification. However, they also come with their disadvantages. The scheme by Boneh et al. [5] is based on computationally expensive bilinear pairings and the scheme by Brogle et al. [6] used a sequential scheme, where the aggregate-so-far must be sent to each subsequent data source, while, ideally, the signatures are aggregated by the oracle itself.

We view the use of synchronised aggregate signature schemes as a potential direction for future work. In the synchronised setting, introduced by Gentry and Ramzan [13], the signing algorithm takes as input a time period t, in addition to the secret key and message. Signatures of different data sources can be aggregated so long as they were all created for the same period t. Hohenberger and Waters [14] note that, in a blockchain, new blocks at a roughly constant pace, which acts as a natural synchronisation event. The oracle could use the number of the block that stores the data request as the current time period t. However, a problem arises when a block contains two or more re-

1 https://ipfs.io/
2 https://github.com/ethersphere/swarm

quests that include the same data source. Another drawback of their scheme is that it works for a bounded number of periods k defined during the global system setup, and the size of secret keys grows linearly with k.

9.3 CONCLUDING REMARKS

The objective of this research has been to reduce the cost of retrieving data from multiple external sources and make the data available to smart contracts in a way that the data's authenticity is publicly verifiable. Although other solutions have been suggested, most are based either on trusted hardware, which creates a single point of failure, or decentralised oracle networks. The latter is relatively slow and expensive, as consensus must be reached upon the retrieved data between collaborating or competing oracles. We believe that assuming a modification at the data source provides a middle ground, as they can be proven to be secure without relying on trusted hardware, and do not suffer from the overhead of a decentralised oracle network.

The oracles presented in this thesis achieve the research objective by providing publicly verifiable authenticity of the data source, including the specific endpoint, and reduce the cost compared to prior solutions through the use of aggregate signatures. Additionally, the oracles provide gas sustainability and fair expenditure. We extracted the components providing these properties and introduced a generic framework based on an existing oracle as a separate contribution that is of independent interest. The results obtained from sample implementations show that the cost of bringing external data into the blockchain can be reduced through the use of aggregate signatures, and that these savings increase with every additional signer.

Just as the rise of public data sources on the Web caused a wave of innovation, we believe that as the adoption of decentralised applications on the blockchain progresses, the interest of making data available to these applications rises with it. The publicly verifiable authenticity of the endpoint and the reduction in costs achieved by the presented oracles provide a first step towards the integration of data sources on the Web with the blockchain and making their data available to smart contracts, without degrading the decentralisation of blockchain technology by creating a trusted entity.

BIBLIOGRAPHY

[1] Elli Androulaki et al. 'Hyperledger fabric: a distributed operating system for permissioned blockchains'. In: *EuroSys*. ACM, 2018, 30:1–30:15.

[2] Mihir Bellare, Chanathip Namprempre and Gregory Neven. 'Unrestricted Aggregate Signatures'. In: *ICALP*. Vol. 4596. Lecture Notes in Computer Science. Springer, 2007, pp. 411–422.

[3] Alexandra Boldyreva, Craig Gentry, Adam O'Neill and Dae Hyun Yum. 'Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing'. In: *ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 276–285.

[4] Dan Boneh, Ben Lynn and Hovav Shacham. 'Short Signatures from the Weil Pairing'. In: *J. Cryptology* 17.4 (2004), pp. 297–319.

[5] Dan Boneh, Craig Gentry, Ben Lynn and Hovav Shacham. 'Aggregate and Verifiably Encrypted Signatures from Bilinear Maps'. In: *EUROCRYPT*. Vol. 2656. Lecture Notes in Computer Science. Springer, 2003, pp. 416–432.

[6] Kyle Brogle, Sharon Goldberg and Leonid Reyzin. 'Sequential aggregate signatures with lazy verification from trapdoor permutations'. In: *Inf. Comput.* 239 (2014), pp. 356–376.

[7] Vitalik Buterin. *Ethereum: A next-generation smart contract and decentralized application platform*. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed: 2018-07-03. 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp and Alfred Menezes. 'Comparing two pairing-based aggregate signature schemes'. In: *Des. Codes Cryptography* 55.2-3 (2010), pp. 141–167.

[9] Victor Costan and Srinivas Devadas. 'Intel SGX Explained'. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[10] Whitfield Diffie and Martin E. Hellman. 'New directions in cryptography'. In: *IEEE Trans. Information Theory* 22.6 (1976), pp. 644–654.

[11] Steve Ellis, Ari Juels and Sergey Nazarov. 'ChainLink: A Decentralized Oracle Network'. In: (2017).

[12] Marc Fischlin, Anja Lehmann and Dominique Schröder. 'History-Free Sequential Aggregate Signatures'. In: *SCN*. Vol. 7485. Lecture Notes in Computer Science. Springer, 2012, pp. 113–130.

[13] Craig Gentry and Zulfikar Ramzan. 'Identity-Based Aggregate Signatures'. In: *Public Key Cryptography*. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 257–273.

[14] Susan Hohenberger and Brent Waters. 'Synchronized Aggregate Signatures from the RSA Assumption'. In: *EUROCRYPT (2)*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 197–229.

[15] Jung Yeon Hwang, Dong Hoon Lee and Moti Yung. 'Universal forgery of the identity-based sequential aggregate signature scheme'. In: *AsiaCCS*. ACM, 2009, pp. 157–160.

[16] Leslie Lamport, Robert E. Shostak and Marshall C. Pease. 'The Byzantine Generals Problem'. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401.

[17] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham and Brent Waters. 'Sequential Aggregate Signatures and Multisignatures Without Random Oracles'. In: *EUROCRYPT*. Vol. 4004. Lecture Notes in Computer Science. Springer, 2006, pp. 465–485.

[18] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin and Hovav Shacham. 'Sequential Aggregate Signatures from Trapdoor Permutations'. In: *EUROCRYPT*. Vol. 3027. Lecture Notes in Computer Science. Springer, 2004, pp. 74–90.

[19] Lukas Malina, Jan Hajny and Vaclav Zeman. 'Trade-off between signature aggregation and batch verification'. In: *TSP*. IEEE, 2013, pp. 57–61.

[20] Alfred Menezes, Paul C. van Oorschot and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[21] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[22] Adán Sánchez de Pedro Crespo, Daniele Levi and Luis Iván Cuende García. 'Witnet: A Decentralized Oracle Network Protocol'. In: *CoRR* abs/1711.09756 (2017).

[23] Gareth William Peters and Efstathios Panayi. 'Understanding Modern Banking Ledgers through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money'. In: *CoRR* abs/1511.05740 (2015).

[24] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley and Srdjan Capkun. 'TLS-N: Non-repudiation over TLS Enabling - Ubiquitous Content Signing for Disintermediation'. In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 578.

[25] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman. 'A Method for Obtaining Digital Signatures and Public-Key Crypto systems'. In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[26] Adi Shamir. 'Identity-Based Cryptosystems and Signature Schemes'. In: *CRYPTO*. Vol. 196. Lecture Notes in Computer Science. Springer, 1984, pp. 47–53.

[27] Yonatan Sompolinsky and Aviv Zohar. 'Secure High-Rate Transaction Processing in Bitcoin'. In: *Financial Cryptography*. Vol. 8975. Lecture Notes in Computer Science. Springer, 2015, pp. 507–527.

[28] Douglas R. Stinson and Reto Strobl. 'Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates'. In: *ACISP*. Vol. 2119. Lecture Notes in Computer Science. Springer, 2001, pp. 417–434.

[29] Nick Szabo. 'Formalizing and Securing Relationships on Public Networks'. In: *First Monday* 2.9 (1997).

[30] Brent Waters. 'Efficient Identity-Based Encryption Without Random Oracles'. In: *EUROCRYPT*. Vol. 3494. Lecture Notes in Computer Science. Springer, 2005, pp. 114–127.

[31] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger byzantium version (e94ebda - 2018-06-05)*. Accessed: 2018-01-03. 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[32] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels and Elaine Shi. 'Town Crier: An Authenticated Data Feed for Smart Contracts'. In: *ACM Conference on Computer and Communications Security*. ACM, 2016, pp. 270–282.