IN, LCA1: Decentralized Verifiable Computation on Distributed Ledger

Due on Summer 2018

D.Froelicher, J.Troncoso-Pastoriza

Max Premi

May 28, 2018

Abstract

Data sharing systems are becoming more popular these past years, and are used in several domains, such as economics [3], software validation [2], and even in the medical field [6]. They can be used for different purposes and might have goals that vary.
Eliminating single points of failure is one of the central goals of decentralized systems, but they can provide the following properties: enforce transparency, provide an efficient way to ensure security, force authentication, and keep the privacy of either party's data.
The latter is required to respect privacy laws but introduces overhead such as encryption, verification of computation (*correctness*), and tracking of error (*robustness*).
UnLynx [4] is such a system that uses Elliptic curve ElGamal encryption, zero-knowledge proof, and noise addition, as well as several other protocols to maintain all properties stated above. However, it only supports a small subset of operations (sum, count, average), and it has a strong threat model.
This project is a contribution to the design of a new decentralized system Lemal, supporting a large set of operations (mean, variance, logistic regression,...), while ensuring privacy and security in an efficient way, in addition of universal verifiability of computations and results. In this project, we introduce a way to make all the proofs public and verifiable by anyone, through distributed ledger. The goal of this project is to:

- First, propose a theoretical system implementation of the Skipchain, to create a Collective Authority of Verifying nodes (VNs) that guarantees correctness and robustness of computation.
- Then, implement protocols that handle the Skipchain operations, based on the previous implementation done by DeDiS [5].
- Implement the interface between the Collective Authority doing the computation and the Collective Authority of verifying nodes.
- Finally, measure the performance of this system, in terms of bandwidth and computation time.

Contents

1 Introduction

Blockchain [3] technology has emerged in 2008 with the Bitcoin creation by Satoshi Nakamoto. It uses distributed and decentralized ledger to create tokens and exchange them in a trusted way with immutability, and avoid double spending problem [7] thanks to consensus through nodes.
It has been widely popularized since 2008, and there is a total of 1604 cryptocurrencies [8] using different techniques of distributed ledger at the time of this report's writing.
However, blockchain applications can be extended to other topics, such as support for correctness and robustness of computations, as well as completely public zero-knowledge proofs.
Indeed, in a data sharing system, one of the biggest challenges is to ensure correctness without centralized authority.
Let's take the example of an identity infrastructure. Nowadays, identity is proved through cards that are issued by a central entity, the government. However, people are developing blockchain application to verify identity without a central authority, to avoid leaking those document, or simply not to leak private information to the state, as well as avoiding falsification of identity.
Some challenges of this application would be to be robust against classical attacks, but also to provide efficient results when asked for an identity and proof that the result was correct (correctness).
Under a weak threat model, it can be assumed that almost all parties are malicious, and it is needed to make a tremendous effort to prove that what has been done is correct. A distributed ledger can easily be used as a decentralized authority that can be consulted at any time to get data that were inserted and cannot be modified. This technique also protect against falsification of identity, if the protocol to insert identity is well made.
One of the major challenges is thus to handle the insertion of the data with a correct consensus, as well as the security of the protocol that interacts with the ledger.
In this paper, we present the implementation of Skipchain [2, 9] into Lemal's framework, to deal with robustness and correctness of computation done by Lemal system. The chain will store the verification of some of these proofs as well as information to verify all the zero-knowledge proof, without leaking anything more than what the proofs actually leak. Then a performance evaluation is done to look at the efficiency of such an implementation

2 Contribution

In this paper, the following contributions are made:

- A theoretical explanation of the design that will be used to make the VNs CA and Skipchain secure, as well as robust and correct.

- An implementation of protocols to handle verification of proofs, and local storage of the later by the VNs.

- The deployment of a Skipchain using the previous implementation of DeDis as a base, to store information about query verification by the verifying nodes as well as a way to retrieve all proof if one want to verify them.

- An evaluation of the performance in terms of efficiency and bandwidth, with a comparison to prior systems such as Unlynx.

3 Background

This section introduces some fundamental concepts used throughout the rest of the report. *Collective Authority* that is the base of both system functionality. *ElGamal encryption* is used in Lemal to ensure privacy, while *Skiphain* is used in the verifying node as distributed ledger. This section also introduces some fundamental background about Blockchain, as Skipchain is a structure derived from Blockchain.

3.1 Collective Authority

Nowadays, applications and systems rely on third-party authorities to provide security services. For example the creation of certificates to prove ownership of a public key. A collective authority is a set of m servers that are deployed in a decentralized and distributed way, to support a given number of protocols.
Each of them possesses a private-public key pair (k_i, K_i), where $K_i = k_iB$ with k_i a scalar and K_i a point in a given Elliptic Curve. This authority constructs a public key $K = \sum_{i=1}^{m} K_i$ which is the sum of all the server's public keys. To decrypt a message, each server i partially decrypts a message encrypted using k_i. Thus the collective authority key provides strongest link security, as no intermediate can decrypt the data without the contribution of all the servers.

3.2 ElGamal Encryption

All the involved scalars belong to a field $\mathbb{Z}_p$.
For Unlynx, data are encrypted using Elliptic Curve ElGamal, more precisely, P is a public key, x is a message mapped to a point and B is a base point on the curve γ. The encryption is the following, with r a random nonce:
$E_P(x) = (rB, x+rP)$. The additive homomorphic property states that $\alpha E_P(x_1)+\beta E_P(x_2) = E_P(\alpha x_1+\beta x_2)$
To decrypt, the owner of the private key p satisfying $P = pB$ multiplies rB and p to get rP and substracts it from $x + rP$ to recover x.

3.3 Skipchain and Blockchain

A blockchain is a continuously growing list of record (blocks), which are linked and secured using cryptographic functions. A block usually contains a hash of the previous block, a timestamp and data.
It is a public, distributed ledger recording block efficiently and that is verifiable and permanent.
A block is immutable and consensus between nodes is achieved with high Byzantine fault tolerance [REF].
A Skipchain is a mixed between blockchain and skiplist, meaning that the block contains forward and backward links, that can jump more than one block.

4 Lemal System

This section presents Lemal [REF] system in general. It goes through the system design, the assumptions made about the parties taking part in the different protocols, the properties that hold, and an example of a query that the system can handle.

4.1 System Model

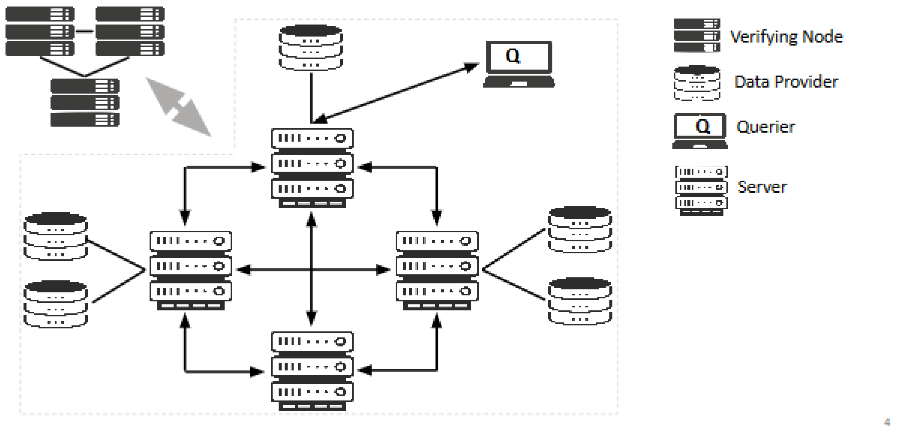

Figure 1: System model of Lemal

Lemal is a privacy-preserving data sharing system developed in collaboration with this project by LCA1. It consists of a collective authority (CA) formed by m servers, $S_1, ...S_m$ and n data providers $DP_1, ...DP_n$ containing sensitive data, encrypted using Elliptic Curve (EC) ElGamal. Another collective authority of verifying nodes (VNs) is linked to this system and maintains a distributed ledger. It is described in Section 5. This system as a whole, is used to answer queries made by a querier Q, to produce results of some aggregate functions. Each DP chooses one server of the CA to communicate with and can change this at any given time.
Functionality: Lemal permits a large set of SQL queries, like *Where, group by, like, mean, variance, set intersection*, with some machine learning (*linear and logistic regression*) and private recommender system functionality such as *cosine similarity, CBF-Based recommendations, ...*.
For any query, some proofs are randomly verified by each node of the VNs and stored in a Skipchain so that any party can verify what is correct. It is also possible to access all the proof and verify them.
The pipeline of the model is as follow:

- Data providers send data with/without Range Proofs to CA server and verifying node
- CA executes collective aggregation protocol and sends the proof to VNs
- CA executes verifiable shuffling protocol and sends the proof to VNs
- CA executes key switch protocol and sends the proof to VNs
- VN did a probabilistic verification upon receiving proofs, the protocol to insert a block containing data to verify the query is launched
- CA sends to querier its result

4.2 Threat Model

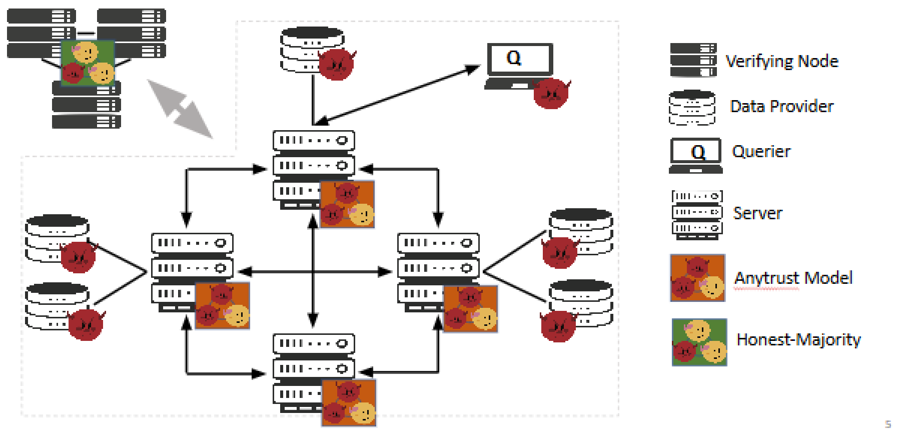

Figure 2: Threat model of Lemal

Collective authority servers It is assumed an Anytrust model [REF]. It does not require a particular server to be trusted or to be honest-but-curious. Whenever one of the servers is not malicious, functionality, security, and privacy are guaranteed.
Collective authority verifying nodes It assumes an Honest majority. Meaning a threshold of more than half of the nodes are honest, and the consensus is done via Byzantine Fault Tolerance (BFT) protocol. This way, a consistent timeline is kept for the chain of blocks, and all data inserted on the chain are verified. Indeed, only one node of the VNs will start the insertion of data (root node), that were collected from all the nodes, in the chain, so there is a verification function running on all node to verify that the root node has not tried to insert malicious data, but the one collected from all the nodes in a previous protocol. Here we ensure that the data inserted are coherent between the nodes of the VNs CA, meaning that each node will verify that the data it's sent to the root node is inserted in the block.
This is different from the proof verification system, as this verification of data inside a block can insert proofs that are not correct, to identify the entity from which this proof comes from. It can then be excluded from the system.
The Querier, and Data providers are assumed malicious. They can collude between themselves or with a subset of the CA servers.
It is also assumed that all network communication is encrypted and authenticated by using a protocol such as TLS or SSL.

4.3 Properties

Confidentiality All data are encrypted using EC ElGamal, no party sees the data in clear, except the Querier that get the aggregate result encrypted under his public key. This property holds as long as one server is honest.
Privacy The system ensures **differential privacy** for any individual sharing its data.
In all cases, the privacy of the querier is not addressed in this system, as we only care for the sensitive data of the DPs.
Correctness At each step of the protocol, zero-knowledge, non-interactive proofs are generated by the CA server or the DPs. These proofs can be used between two steps of the protocol to verify that computation was correctly done, and if it's not it can identify the entity that did not compute correctly.
Robustness is ensured through the distributed ledger. In case of faulty computation, the block contains the

verification of proof at each server. If there is an incoherence in these verifications, the block also contains a way to retrieve all the proof from the VNs CA.
Anyone can verify and identify the server/DP or a set of them that cheated.
This way, one could exclude these malicious parties from the protocol.

4.4 Query example

This subsection details the pipeline of a query process.
First, the querier Q send a query to the CA server. The example taken in this section is "SELECT average(x) FROM y WHERE y.z < 321".
This query is received by a server s of the CA and broadcasted to the entire CA. Then each server sends to its DPs, the query. The DPs execute it locally, then send the result encrypted with the CA's public key to each server that sends them the query.
The first proof is sent to each node of the VNs, proving that the cipher resulting from each DP is in a given range.
At this point, each server of the CA contains ciphers encrypted with the CA's public key. Then, a collective aggregation is done. In the current example, the server will add the ciphers and the root server (the one the querier contacted) will initiate an aggregation protocol, to get all the final result, which is the sum at each server and the number of values. This protocol is done with a tree implementation. Children servers will send their average to their parent, it will aggregate, and the same thing will happen until the top of the tree is reached.
Again, a proof is generated, which is just the ciphers before and after aggregation, and is sent to the VNs.
At this point, there might be differential privacy applied to perturb end-result in order to satisfy privacy. In this example, servers might collectively add several time values that does not make the whole mean varies too much, but it gives privacy on the number of values, and individual result at each server.
This comes from Unlynx with Distributed Results Obfuscation (DRO) step, that enables the CA to collectively and homomorphically add noise, sampled from a probabilistic distribution. To do so, a shuffling phase is initiated, where multiple sequences of noise are randomly permutated. This ensures that the noise is randomly chosen for a given query.
This shuffle is also verifiable, a proof is generated at each server when its execution is finished.
Finally, a key switch protocol is engaged. It permits to sequentially add values to the final result to finish with a cipher encrypted under the public key of the querier. To do so, each server adds an element containing a nonce and its public key, with the proof of what it has computed is correct.
Eventually, the root server sends back the result to the querier.

Note that we did not include the Verifying nodes process in this query example, but only what it receives, as we detail it further in the following section.

5 Verifying Node

This section presents the implementation of the Verifying nodes CA. It was the main goal of this project and will be the major part of this report. We will first present in details the threat model and the theoretical system model, then the operations supported, and in the following section the way we choose to implement it.

5.1 System Model

The verifying node CA's goal is to relax the proof verification time of potential client, by doing the computations, and store data related to the proofs inside a block of a distributed ledger. This also ensures

correctness as proof verification and proofs are stored in an immutable way, and a way to identify faulty party, i.e. robustness.
To avoid taking a long time to verify all the proofs, several modifications are done.

When a query is received by the CA, the exact number of each type of proof expected from each entity is known. So the CA sends to the VNs this number to initialize a dictionary that will be referred as "*bitmap*" from now on. This bitmap is used to keep track of the proofs received from the CA. It is a way to link proof to its verification without having the to store all proofs' strucuture inside the block. So a proof is referenced as a string and its verification as an integer, inside the *bitmap*.

Then, proofs are sent as soon as they are created to all nodes of the VNs CA, to be sure that the proofs are coming from the correct entity, all of them are signed by the server/DP that issued it, and the signature is verified before processing the proof.
Each of the node verifies a set of the proof chosen probabilistically with a fixed threshold $p \in (0, 1]$. Each server chose the subset in a uniform way, so the sets are different with high probability, and might intersect. The verification results in a 1 if the proof is correct, 0 if incorrect, 2 if it has not been verified, and 3 if the proof has not been received (set by a timeout).
After verifying these proofs, it stores the result of the verification in bitmap with the query ID, identity of verifier and creator, as well as the type of the proof as key and the verification result as value in a local database. The proofs' structure are also stored in this database.

To be more precise, when a proof p is send by a server s to a node of the VNs v, for a query with ID *queryID* it randomly verifies it, and store 2 values:
bitmap[*queryID*+*qi*+"*typeOfProof*"+identity(s)+identity(v)] = verif(p), where *"typeOfProof"* can be in this context: *range, aggregation, shuffle* or *keyswitch.*
This bitmap is saved in the local database for each query when all proofs have been processed, and will be stored in the Skipchain. This is done so that the service can access the bitmap stored by itself in another service. The value are deleted as soon as the block has correctly been inserted.
All proofs are stored at different key, but can be retrieved by given the queryID. (*queryID*,*p*) is stored in the local database, but is kept after the query is finished.

The VNs then wait for other proof of the same query to be received, and do the same for all other types of proofs.
Upon receiving the last proofs, i.e. when the bitmap has been entirely filled, each server gather all results of verification inside a unique bitmap, and a protocol to gather all bitmap to one root server is launched.
At the end of this protocol, the root node launches an insert (or a create chain if there is none yet) block operation given the bitmap aggregated, the query ID, the probability used to select the proof to verify and a timestamp.
If the block is correctly formed, and the automatic verification done by each server pass, then it inserts the block, else it will not insert it. As previously described, the verification done by each server is checking whether its bitmap appears in the data of the new block request. If it does not appear it means the root node tried to falsify information.
Any client can fetch the last block of the Skipchain at any time.

To resume the protocol is presented step by step in chronological order:

- Query is received by the CA, exact number of proof is known and set for this query
- Query is broadcasted to the DPs, that send their data to the CA, and range proof to the VNs if and only if there is a need for range proof. They are signed by the DPs

- CA process the data, creating proof at each step, that are signed and sent to the VNs
- Upon receiving a proof, a node verify the signature, then verify the proof, if it was expecting more proofs of this type, and store the proof in a local DB
- When all proof have been processed, each node store its final bitmap in its database and a protocol to insert a block is launched by the root node.
- The root node gather all the *bitmap* and pass them as data to insert a new block, as well as a timestamp and the queryID.
- Inside the block insertion service, each node verifies that the data from the final *bitmap* contains its data, by comparing the received data with the one stored inside the local database
- If the block was well formed and the previous verification was correct, the service return the newly added block, with the bitmap from all server, timestamp, threshold as data

5.2 Threat Model

As described previously, the threat model of the VNs CA is Honest-majority. This means that at least half of the servers are honest. This property must hold in order to maintain the Skipchain correctly. Indeed, an adversary must compromise at least half the servers, to violate correctness and to make the insertion of a malicious block possible, without being rejected.

5.3 Type of Proof

As described in section 4.4, the CA server generates a different kind of proofs. This subsection will present in details what they contain and how they are verified. All these proofs come from Unlynx [REF] framework.
Range proofs are computed by each DP. They prove that a commitment C is in a range $[a, b]$. This is used to handle malicious DP input. In a query with range predicate, it is perfect to prove that an encrypted data indeed lies in a given range, without leaking the value in itself. This proof was designed to compare Prio and Unlynx in a previous project [REF]. It is optional, as it is possible that a query does not contain a range predicate.
Shuffle proofs are computed by the CA server to prove privacy. This ensures the privacy of previously verified data, with differential privacy via noise addition. It verifies that the noise added has been randomly chosen.
Aggregation proofs are also computed by the CA server. It ensures correctness of computation, without leaking any information on data, except the information that the operation leaks in itself. For example, a mean operation will leak the number of data that the CA aggregates. It verifies the ciphers before aggregation and the resulting aggregation.
Key Switch proofs are computed by the CA server. These proofs ensure the security of data, and correctness of computations. Its goal is to verify that the protocol to switch encryption key is correct without leaking the data in clear. It verifies that data before and after addition of a value by server are coherent.

5.4 Operations supported

Here, the operations supported by the verifying node are presented in details.
There are two services that we can distinguish. The first one is the service that handles the proofs receiving, verification and storage.
The other service handles the communication with the Skipchain, as well as the function and data accepted by the Skipchain.

5.4.1 Query and Proof handling

The first service handles query receiving as well as proofs. Upon receiving the query, it initializes the bitmap and prepares it for a fixed number of proofs.
Then each type of proof described previously in section 5.3 can be handled, verified and store independently, they each have a handler. Each proof must be signed by the entity that issued it to be processed correctly. The storage is done locally at each server using BoltDB [REF].

5.4.2 Skipchain operations

The second service is the link between the first one and the Skipchain. It permits to insert a block or create one if no chain existed. It is automatically launched when the bitmap has been fully filled. It calls the service implemented by DeDiS to create/maintain a Skipchain. It takes a structure containing all the data that a block contains as parameter, and return the block if it has been inserted correctly in the chain.

5.4.3 Fucntion for external clients

Some functions are available for external clients, to retrieve information from the Skipchain or the VNs.
It is possible to get the last block of the Skipchain, as well as a specific block, given its ID (hash), its index or the query ID. This way anyone can access the data in the block (the *bitmap*), to check the verification of proofs. The servers/DPs that issued the proofs and he nodes of the VNs that verified this proof are known as they are stored in the *bitmap* as key.

There is also a way to access all the proofs for the query the block was created for.
Each server of the VNs stores the proofs issued at each query, and a client can ask to any nodes all the proofs for a given query ID.
As they are stored in the same key/value way as verification, it is easy to identify which server verified what proofs, and which issued them.

6 Implementation

In this section, the technical implementation of the VNs CA is detailed. All the implementation was done in Unlynx branch called Lemal [REF].
We first present the service that handles, verify and store the proofs. Then the service that handles the communication with the Skipchain.
Finally, the modification made to the Skipchain service of DeDiS [REF] is addressed in the last part, as well as the possible call that external entities can make to the VNs CA.

To make the implementation easier, each DP and server are considered client of the Verifying nodes. This way, they can send data using API function.

6.1 Handlers

This subsection presents the different handler of the client and service, for the proofs.

We define a client of the service as the *API* structure, that contains its ID as well as the private key it uses to sign the data its sends.

On a verifying node, a Service runs and the structure contains several important fields: A concurrent map that stores all information of a query processing, meaning the size, the bitmap and the query itself. The bitmap and the size are also local fields of the service to be used when a proof for this query need to be handled and both these structure will be updated in consequence.
The threshold to verify proof is set at 0.4, but can be changed. Then, a local database is defined using boltDB.

The client can call several function to send data to the service. The first call expected is *SendQuery*, that broadcasts the query to all the nodes of the VNs CA.
These nodes handle the query with the *ReceiveQuery* function, that set the bitmap and the number of proof expected for this query, given its ID.

Then all the different type of proofs have they own handler.
We will describe precisely how one handler works, and generalize for all of them.
Range proof is sent by the client using *SendRange*. It broadcasts to all nodes of the VNs the proof, as well as the signature, the identity of the client and a potential **SkipBlock**. If it is the last proof for this query, after being handle, the insertion process is launched throught a function called *checkForInsertion*, appending a new block containing all the previously handled proof for this query to the block passed as argument. If it is not the last proof or there is no skipchain yet, this field can be *nil*.
This *checkForInsertion* is the return value of each handler. It return 2 values, the first one nil if no insertion have been done, or a *SkipBlock* if it has been correctly done. The second one is an error, if it is not nil, the block will be nil.
The service receives this proof with the *HandleRange* function, that verifies first if more proofs of this type where expected thanks to the concurrent map, verifies the signature, and verifiy probabilistically the proof. Then, it updates the bitmap, and the number of proof expected, and store them back inside the concurrent map. It also stores in its local database the proof.

To extend about how the proof verification and proof itself are stored, we explain here how we defined the structure to do so efficiently.
The verification of proofs are stored inside a map of (string,integer), as long as the whole query has not been processed. Then it is deleted, as it is stored inside a block of the ledger, and can thus be retrieved. The **key** of this map is a string of the form : *queryID+"typeOfProof"+clientID+deterministicInfo+nodeID*. The deterministic info is used if the sender of proof need to send several profo of the same type (for example the DPs can send more than 1 range proof) while keeping the unicity of the key. The value is 0,1,2 or 3 in function of the value of return of the verification.
For the storage of the proof in boltDB, the same system of key/value is used. The only difference is that the database is split in buckets, that have maximal capacity. So upon receiving a proof, a bucket is created/loaded with key *queryID+"typeOfProof"*, and the proof is stored with the same key as the bitmap, and the value is the proof in bytes.

The same implementation is used throught all handlers. *SendAggregation* and *HandleAggregation*, *SendShuffle* and *HandleShuffle*, *SendKeySwitch* and *HandleKeySwitch*.

6.2 Skipchain Operations

There are 2 type of operations on the Skipchain. The insertion of a block, or getting one from the chain.
The first one is done via the handler, as if all proof has been received, the insertion will begin inside *checkForInsertion* function. If no block were passed as argument a new chain is created, else it is happened at the end of the chain. This is done by calling the *skipchain* service implemented by DeDiS. It handles all the technical features of the chain itself, and is not the aim of this project and will not be discussed in

details.
In addition to this function, anyone can ask for the last block known by the VNs CA given a block as argument. The client can call *GetLatestBlock* giving the roster (the set of node) that handle the skipchain and a SkipBlock. The service will receive the request with *GetLatestBlockRequest* and send back the last block it knows.

6.3 Modification of the Skipchain

Some modification have been done to the service implemented by DeDis to fit the scenario described in the report. The only modification is basically the addition of a verification function that run on all nodes when a block is received to verify if its structure and data are correct.
This is done by adding a registration for this function in *skipchain.go*, and by writting this function inside the *verification.go* file.

6.4 API for external Clients

7 Performance Evaluation

8 Conclusion and Future Work

References

[1] Henry Corrigan-Gibbs, Prio prototype implementation
`https://github.com/henrycg/prio`

[2] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, *CHAINAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds*
`https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-nikitin.pdf`

[3] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*
`https://bitcoin.org/bitcoin.pdf`

[4] David Froelicher, Patricia Egger, Joao SaŚousa, Jean Louis Raisaro, Zhicong Huang, Christian Mouchet, Bryan Ford and Jean-Pierre Hubaux, *UnLynx: A Decentralized System for Privacy-Conscious Data Sharing*
`https://petsymposium.org/2017/papers/issue4/paper54-2017-4-source.pdf`

[5] *Decentralized and Distributed Systems*
`https://dedis.epfl.ch/`

[6] Heather Fraser, *How Blockchains can provide new benefits for Healthcare*
`https://www.ibm.com/blogs/think/2017/02/blockchain-healthcare/`

[7] *Double-spending problem*
`https://en.wikipedia.org/wiki/Double-spending`

[8] *Diverse Information about Crpyptocurrencies*

texttthttps://coinmarketcap.com/

[9] *Skipchain implementation repository*
`https://github.com/dedis/cothority/tree/master/skipchain`