IMPLEMENTATION ANALYSIS OF CRYPTOGRAPHY TOOLBOX IN HYPERLEDGER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET ŞİMŞEK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2021

Approval of the thesis:

IMPLEMENTATION ANALYSIS OF CRYPTOGRAPHY TOOLBOX IN HYPERLEDGER

submitted by **AHMET ŞİMŞEK** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. A. Sevtap Selçuk Kestel _____
Director, Graduate School of **Applied Mathematics**

Prof. Dr. Ferruh Özbudak _____
Head of Department, **Cryptography**

Assoc. Prof. Dr. Oğuz Yayla _____
Supervisor, **Cryptography, METU**

Examining Committee Members:

Assoc. Prof. Dr. Murat Cenk
Institute of Applied Mathematics, METU _____

Assoc. Prof. Dr. Oğuz Yayla
Institute of Applied Mathematics, METU _____

Assoc. Prof. Dr. Sedat Akleylek
Computer Engineering, Ondokuz Mayıs University _____

Assist. Prof. Dr. Adnan Özsoy
Computer Engineering, Hacettepe University _____

Assoc. Prof. Dr. Fatih Sulak
Mathematics Department, Atılım University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: AHMET ŞİMŞEK

Signature :

ABSTRACT

IMPLEMENTATION ANALYSIS OF CRYPTOGRAPHY TOOLBOX IN HYPERLEDGER

Şimşek, Ahmet
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Oğuz Yayla

September 2021, 47 pages

Hyperledger was set up with the aim of being an open-source platform targeted at accelerating industry-wide collaboration hosted by The Linux Foundation for developing robust and dependable blockchain and distributed ledger-based technological platform that may be applied across several industry sectors to improve the efficiency, performance, and transactions of different business operations. Various distributed ledger frameworks and libraries have been developed for this purpose. In this thesis, the Ursa cryptographic library, which is one of the libraries being developed to offer its users with dependable, secure, user friendly and pluggable cryptographic applications to its users, has been examined and the performances of both the anonymous identity creation process and the presented cryptographic algorithms are examined.

Keywords: Hyperledger, CurveZMQ, Verifiable Credentials, Decentralized Identifiers, Aries, Ursa, Indy, libursa etc.

ÖZ

HYPERLEDGER ÜZERİNDEKİ KRİPTOGRAFİK ARAÇLARIN GERÇEKLEMELERİNİN İNCELENMESİ

Şimşek, Ahmet
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Oğuz Yayla

Eylül 2021, 47 sayfa

Hyperledger, verimliliği ve performansı artırmak için çeşitli endüstri sektörlerinde kullanılabilecek yüksek performanslı ve güvenilir blok zincir ve dağıtılmış defter tabanlı teknoloji çerçevesi geliştirmek için Linux Vakfı tarafından yürütülen endüstri çapında işbirliğini ve çeşitli iş süreçlerindeki işlemlerini hızlandırmayı hedefleyen açık kaynak kodlu bir platform olmak amacıyla kuruldu. Bu tezde, kullanıcılara güvenilir, güvenli, kullanımı kolay ve taşınabilir kriptografik uygulamalar sağlamak amacıyla geliştirilmekte olan kütüphanelerden biri olan Ursa kriptografi kütüphanesi incelenmiş ve hem anonim kimlik oluşturma sürecinin hem de sunulan kriptografik algoritmaların performansları incelenmiştir.

Anahtar Kelimeler: Hyperledger, CurveZMQ, Doğrulanabilir Kişisel Bilgiler, Merkezsizleşmiş Tanımlayıcılar Aries, Ursa, Indy, libursa vd.

To my family

ACKNOWLEDGMENTS

I would like to express my very great appreciation and thank to my thesis supervisor Assoc. Prof. Dr. Oğuz Yayla for his knowledge, emphasis on details, encouragement, vision, invaluable ideas during the development and preparation of this thesis. His willingness to give his time and guidance was unvaluable on the process.

I thank my examining committee members for their time they spared for me. I would also like to express my gratitude and thanks to Assoc. Prof. Dr. Ali Doğanaksoy for introducing me to cryptography.

Also a special thanks to my friends and colleagues who are there for me when I asked for and supported me in the completion of this thesis.

Last but not least, I would like to express my deepest gratitude to my family who stood besides me and supported me. I'm grateful for their love and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

DLTs	Distributed Ledger Technologies
DID	Decentralized Identifier
DIDComm	DID Communication
VCs	Verifiable Credentials
VON	Verifiable Organizations Network
CredDef	Credential Definition
UI	User Interface
SSI	Self-Sovereign Identity
ACA-Py	Aries Cloud Agent-Python
TLS	Transport Layer Security
MITM	Man-in-the-Middle
DOS	Denial of Service

CHAPTER 1

INTRODUCTION

Ever since the computer revolution began in the 1950s, databases have played an important part in business and society. Databases began as simple applications with all the data arranged in flat files, like a list of contacts. As companies called for more speed and power, database management played a key role as all the information arranged as rows and columns in tables. The globe is currently so interconnected that people often have to access the same info. To fulfill this demand, distributed databases have been created in which more than one individual may view particular portions of data simultaneously. Once a database is shared with others, a lot of practical questions emerge when you share a database. Over the years, many alternative solutions have been explored. One exciting new way to share databases that can help solve these problems is through blockchain technology. Blockchain technology, started to be implemented by companies/individuals in the industry according to their needs and created their own blockchain infrastructures. However, the lack of standards in the technology, which has gained great momentum in recent years, was an obstacle for all small and large companies and individuals to use this technology that could meet their problems/needs. Also because of lack of standardization, blockchain solutions, which were successfully applied to business models, raised concerns in terms of security. To relieve users and businesses and to increase reliability Hyperledger [3, 11], an umbrella project aiming to develop and combine open source blockchain technologies, was launched by the Linux Foundation in December 2015 in order to ensure that blockchain technology is more efficient and reliable, and to create a code base that companies/individuals can quickly adapt to their needs. In our studies on Hyperledger, we focus on the three identity-focused projects in the community, Indy

[1], a distributed ledger purpose-built for decentralized ID with transferable, private, and secure credentials, Aries [19], which is an infrastructure that supports interactions between peers and between blockchains and other DLTs, offers exchange protocols and implementations of agents for people, organizations and things, and Ursa [20], Ursa is a modular, flexible library that enables developers to share time-tested and secure cryptography which is also the underlying crypto library for the Indy [1] and Aries [19] projects. In our thesis, with the aim of getting familiar with the Hyperledger ecosytem's cryptographic toolbox and searching for possible bottlenecks that is open for improvements, we take the performance metrics of the crypto algorithms used by the ursa crypto library in the anonymous identity creation stages and share the benchmark values of the algorithms on different devices.

The outline of the thesis is as follows. In Chapter 2, we review the Aries and Indy projects and their related components and talk about emerging issues of identity in the digital world to highlight the issues Indy is trying to answer, and we present the ACA-Python application provided by the Province of British Columbia to show a living example of VCs and DIDs. In Chapter 3, we focus on overall operation of CurveZMQ [21, 22] protocol and the design aim of the protocol is explained. It is essential protocol that is used for communication between two digital identity in hyperledger network. In Chapter 4, we move on Ursa [20] project which is intended and used by open-source blockchain developers and enthusiasts for a range of applications that demand the same secure, effortless, and pluggable cryptographic implementations. We will try to explain why the Ursa lib was created. Further, we explain the concept of anonymous credentials and give the performance test results of the both anoncreds creation/revocation [23] and cryptographic primitives like digital signatures, key exchange algorithm used within libursa library [4]. Finally, in Chapter 5, we conclude our studies and give some future works.

CHAPTER 2

PRELIMINARIES

When we set out to examine the cryptographic infrastructure on the hyperledger platform, we examined some projects in order to see which problems the system answers. One of them was Aries and the other was Indy projects. The reason why we examined these projects was simply that the Indy and Aries projects offered solutions to the big question of how to create an identity in the digital environment and how to provide secure communication between identities in digital environment, and to see running instance of the system respectively. In this regard, in this chapter we look through Aries solution and its components which are building blocks of most of hyperledger projects, we show running instance provided by Province of British Columbia and what are the problems that Indy solves and important components of it explained.

2.1 Hyperledger Aries

Hyperledger Aries [19], according to its documentation, offers reliable interactions between peers on the basis of decentralized identities and verifiable credentials. An Aries solution has several important components which are building blocks of most of hyperledger projects :

- agents [16]
- DID Communication [17]
- protocols [15]
- and key management [18]

Agent concept is like a real estate to help us buy a house because in digital world humans and organization cannot directly store and manage data or perform the crypto that self-sovereign identity(SSI) [12, 13] demands. They needs delegates–agents– to help. An agent has three defining characteristic that don't tie an agent to any particular blockchain. Therefore it is possible to implement agents without any use of blockchain at all. These characteristics are:

1. It acts as a fiduciary on behalf of a single identity owner.
2. It holds cryptographic keys that uniquely embody its delegated authorization.
3. It interacts using interoperable DIDComm protocols.

We mentioned that agents are interacts using DIDComm. So lets explain how this typical DIDComm interaction works with a high level example :

Assume Alice wishes to bargain with Bob to sell things online, and DIDComm is used instead of actual human communication. This indicates that Alice's and Bob's agents will be exchanging communications. Alice may just click a button, oblivious to the facts, but her agent starts by creating an unencrypted JSON message detailing the intended sale. It then looks up Bob's DID Doc to access two key pieces of information:

- An endpoint (web, email, etc) where messages can be delivered to Bob.
- The public key that Bob's agent is using in the Alice:Bob relationship.

Now, Alice's agent encrypts the plain-text with Bob's public key so that only Bob's agent can read it, and it adds authentication with its own private key. The agent prepares for Bob's delivery. This "preparing" can include a number of hops and intermediates. It can be difficult. Bob's agent finally receives and decrypts the communication, verifying its source as Alice using her public key. It prepares its answer and routes it back through a reciprocal process :

- plaintext $\rightarrow$ lookup endpoint and public key for Alice $\rightarrow$ encrypt with authentication $\rightarrow$ arrange delivery

The specific interactions enabled by DIDComm–connecting and maintaining relationships, issuing credentials, providing proof, etc.–are called protocols. Protocols used in communication between agents are stateful interaction patterns. They specify things like, "If you want to negotiate a sale with an agent, send it a message of type X. It will respond with a message of type Y or type Z, or with an error message of type W. Repeat until the negotiation finishes."

Aries is a fledgling project that is rapidly developing. As a result, developers interested in solving business challenges should begin with an Aries agent framework.

2.2 Implementing the Hyperledger-Indy Framework using ACA-Py

ACA-py (Aries Cloud Agent Pyhton) is the suggested way to build enterprise applications on top of the decentralized identity-related Hyperledger projects which has production deployments. It is appropriate for any non-mobile agent application. ACA-Py run together with a controller, which are communicating across an HTTP interface. An ACA-Py agent instance that has been deployed creates an OpenAPI-documented REST interface for administering the agent's internal state and sparking communication with connected agents. Because API methods will often kick off asynchronous processes, the JSON response provided by an endpoint is not always sufficient to determine the next action. To handle this situation as well as events triggered due to external inputs (such as new connection requests), it is necessary to implement a webhook processor. The combination of an OpenAPI client and webhook processor is referred to as an ACA-Py Controller. The controller can start agent activities such as credential issuance and respond to agent events such as sending a presentation request upon the acceptance of a new pairwise DID Exchange connection. Agent events are sent to the controller as webhooks to a structured URL.

As we stated that ACA-Py is suggested way to build enterprise application. So, in order to understand ACA-Py better, we need to implement the framework but fortunately The Verifiable Organizations Network (VON) team based in the Province of British Columbia created the initial implementation of ACA-Py.

The Verifiable Organizations Network (VON) [7] is a portable development level Indy

Node network and its components are based on Hyperledger Indy distributed ledger technology. It is a community effort to establish a better way to find, issue, store and share trustworthy data about organizations. Von partners are using jointly developed software components to enable the digitization of organizations-issued public credentials—registrations, permits, and licenses. VON helps by making :

- applying for credentials faster and less error prone
- issuing (and reissuing) credentials simpler and more secure, and
- verifying credentials more standard, trustworthy, and transparent,anywhere in the world.

It is a useful mechanism for testing on sandbox Indy Networks.

We'll construct and launch a VON Network instance to explore what you can do with it.

1. To begin, open a bash command prompt and clone the von-network repository:

```
git clone https://github.com/bcgov/von-network
cd von-network
```

2. After cloning the source, we'll construct docker images for the VON Network and then launch your Indy network:

```
./manage build
./manage start --logs
```

As the nodes of the Indy network start up, keep an eye on the logs for any problem warnings. The $./\mathbf{manage}$ bash script streamlines the operation of the VON Network.

When the ledger is up and running, you may view it by connecting to the web server at port 9000. That implies going to http://localhost:9000 on localhost. With the VON-Network you can see states of nodes, examine genesis file and create a DID.

Additionaly you can walk through the three ledgers that make up an Indy network:

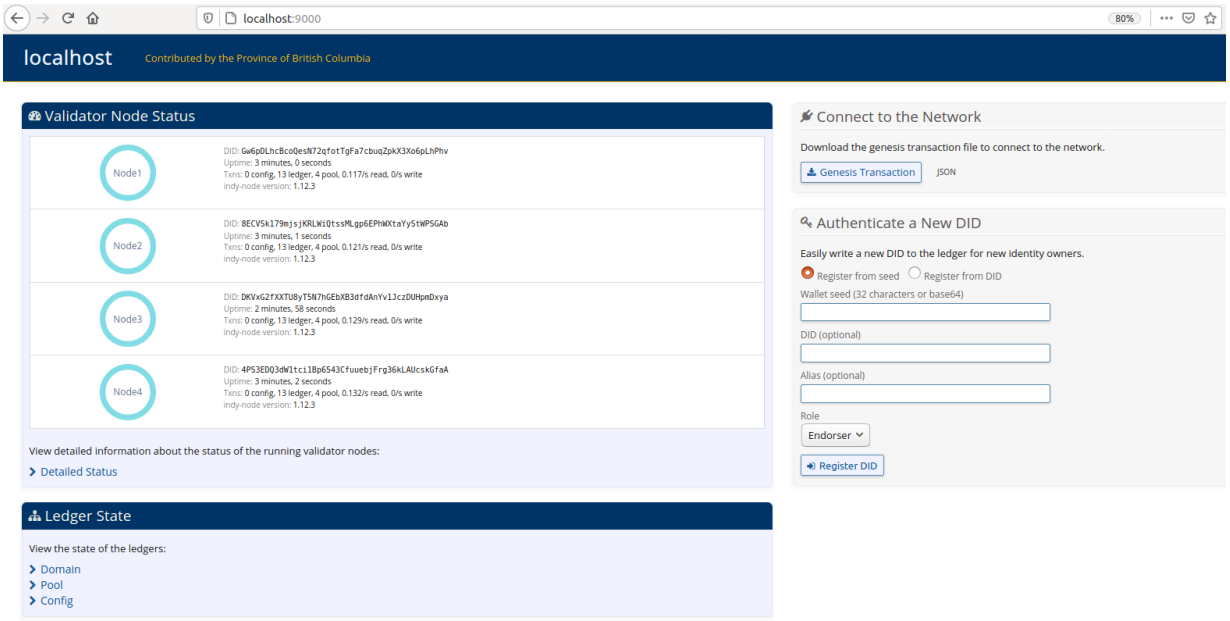

Figure 2.1: Running Instance of Aries Cloud Agent on local host

- The Domain ledger, where the DIDs, schema, etc. reside.
- The Pool ledger, where the set of nodes on the network are tracked.
- The Config ledger, where changes to the network configuration are tracked.

You can see all this from the Figure 2.1.

1. To halt and remove a running VON Network, open a command prompt (Ctrl-C if necessary), and then type the following command:

   ```
   ./manage down
   ```

   You'd usually do this after you've accomplished some work and testing and would like to start over.

2. Use the following command to terminate the network without destroying the data on the ledger:

   ```
   ./manage stop
   ```

3. You can relaunch the ledger later by issuing the standard network startup command:

   ```
   ./manage start
   ```

Inside von network we mentioned that its components are based on Hyperledger Indy distributed ledger technology. Lets dig deep on Hyperledger Indy distributed ledger technology.

2.3 Hyperledger Indy

Hyperledger Indy is a distributed ledger technology in which Plenum [2, 14] is the heart of the distributed ledger technology inside Hyperledger Indy. Plenum is special purposed for use in an identity system. The system maintains a replicated sequential transaction log called a ledger, and all communication between client and node and node and node takes place in CurveZMQ protocol [3], which we will explain the details of the protocol in the next chapter. Indy is all about Identity on the Internet. It is about being able to prove to people who you are and being assured of who they are. The problems with Identity on the Internet today come down to a single word - trust. When you are interacting online with someone, do you trust:

- Is the person you are connecting with online who they say they are?
- Are the claims they are making true?

The basic mechanism for knowing who you are on the Internet is the UserID/Password combination Figure 2.2. We all know the problems with User IDs and Passwords, we deal with them every day: We either have too many passwords to track or less that are used in many places. In the former case, we prefer guessable passwords to keep track of them. As a result password cracking mechanism provide avenues of attack by hackers. In the latter case, when we have few passwords that we use in many systems, this time when there is a data breach in one system, it exposes our data on other systems.

You register on a website and receive a UserID and create a secret password which you alone are aware of, and you use to access your account each time you return. Also need to note that while website users have IDs for the site, the opposite is not true - you don’t receive a "id and password" for the site that you can verify each time you logged in. This has enabled the "phishing" techniques. Indy solve above mention problem with DIDs-Decentralized Identifiers.

Figure 2.2: Example Real World Identity-Proofing Paper Documents (Licensed under CC By 4.0)

2.3.1 The Decentralized Identifiers (DIDs)

The Decentralized Identifiers (DIDs) [29] are a new type of globally unique identifier designed to enable individuals and organizations to generate our own identifiers using systems we trust, and to prove control of those identifiers (authenticate) using cryptographic proofs (for example, digital signatures). In other words, DIDs are URIs that link a DID subject to a DID document, permitting for trusted engagements with that subject. A DID specifies any subject that the DIDs' controller wishes to identify (e.g., people, organisation, thing, data schema, abstract entity, etc.). A DID is a simple text string consisting of three parts as shown in Figure 2.3.

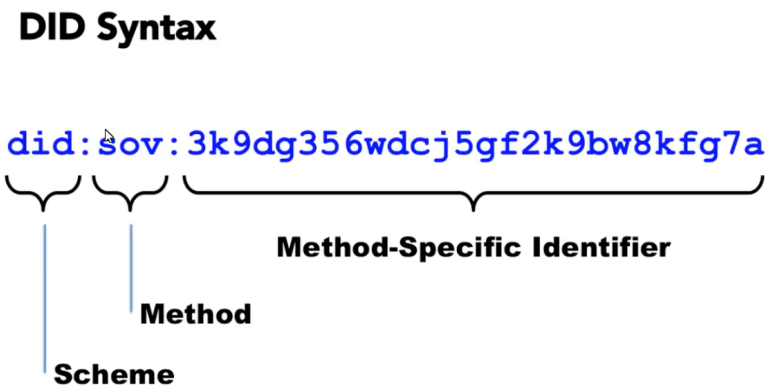

Figure 2.3: Structure of a DID (Licensed under CC By 4.0)

Each DID has associated with it one or more public keys created by the DID owner (and the owner holds the corresponding private keys), and one or more endpoints - addresses where messages can be delivered for that identity.

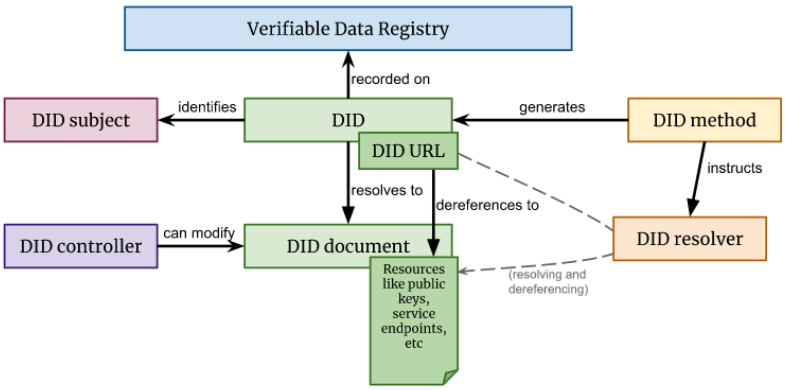

Figure 2.4: The basic components of DID architecture

We can see how a DID ecosystem looks like by looking at Figure2.4 and to understanding the Figure2.4 we need to define the components of it. After that, it will be self-explanatory.

DID Subject

The entity recognized by the DID and described by the DID document is the DID subject.

DID Document

Each DID document can express cryptographic material, verification techniques, or service endpoints, which offer a set of procedures that allow a DID controller to confirm DID control. Verifiable timestamps and signatures allow DID documents to be cryptographically verifiable. A verifiable timestamp allows a third party to validate that a data object existed at a certain point in time and has not been updated or damaged since that point in time.

DID Controller

The controller of a DID is the entity (person, organisation, or autonomous software) with the capacity to make modifications to a DID document as described by a DID method. Control of a set of cryptographic keys used by software working on behalf of the controller is generally used to establish this capacity.

DID Methods

DID methods are the mechanisms for creating, resolving, updating, and deactivating a certain type of DID and its associated DID document utilizing a specific verified data registry.

Verifiable Data Registry

A verifiable data registry is a system that allows for the recording of DIDs and the retrieval of data required to create DID documents.

DID URLs

A DID URL expands the syntax of a basic DID to include extra common URI components (path, query, fragment) in order to identify a specific resource, such as a public key within a DID document or a resource available outside of the DID document.

DID Resolvers

A DID resolver is a software or hardware element that accepts a DID (and related input metadata) as input and outputs a complying DID document (and associated information).

Indy uses DIDs to establish connections between two identities, such as a user and a service's website, so that they can securely communicate. Further, the expectation is that an entity - e.g. you - will have many DIDs - one for each relationship you have with another entity. The reason is that with many DIDs as one need to maintain intended separation of identities, personas, and contexts since you can control the production and assertion of these identifiers (in the everyday sense of these words). Think of each DID like a userID/password pair, but one that is backed with strong cryptography in the form of public/private keypairs. As well, note that both sides of a relationship provide a DID for the other to use to communicate with them. The Figure 2.5 show three entities, Alice, Bob and a Bank that both Alice and Bob use. For each entity, we see the various DIDs they have created for their relationships. We've also highlighted the DIDs that they have exchanged with each other - Alice's for Bob, Alice's for the Bank and so on.

To better understand it let's look at the following example : A user registers for a service's website by creating and giving the service a new, never-used-before DID, and receives back from the service the same thing - a new, never-used-before DID created by the service. Each records the "relationship" DIDs so that when one wants to communicate with the other, they have an endpoint to send the message, and a public key to end-to-end encrypt the message. Later, when the user returns to the website to login, the user and the service exchange encrypted messages to confirm that each holds the private key to decrypt the messages. On completion, the service

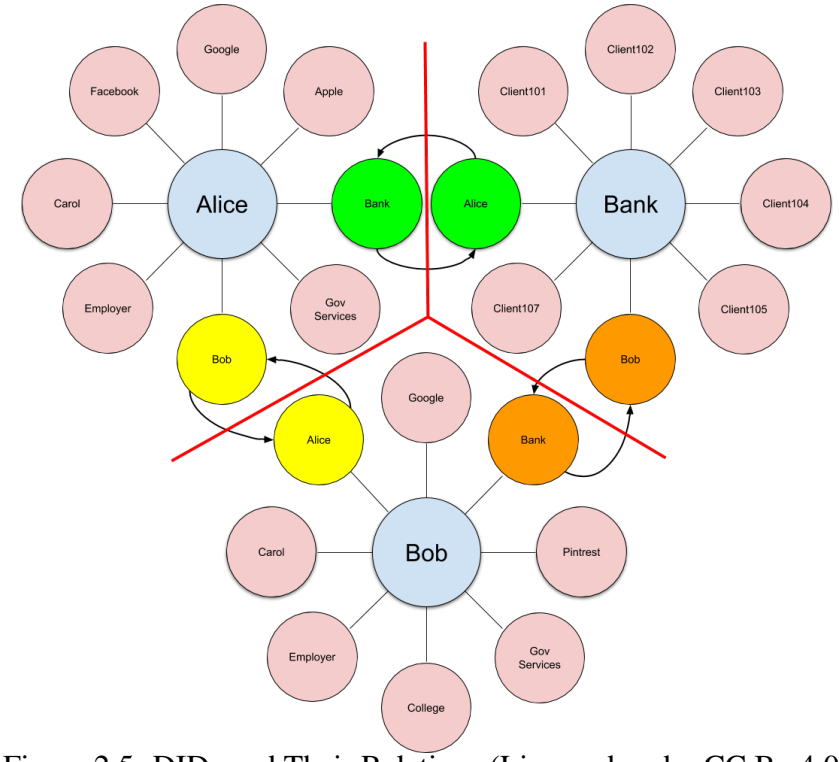

Figure 2.5: DIDs and Their Relations (Licensed under CC By 4.0)

knows that it's the user because the user used their DID, and the user knows it's the service because the service used its DID. With that we've already addressed one of the challenges raised with today's Internet Identity - two-way verification!

The term "decentralized" is a hint that Indy uses blockchain technology. The Figure 2.6 shows how the Agents send requests to the ledger to read and write DID (and other) information. The edge layer, which generates and stores the majority of private keys, and the agent layer, which exchanges and verifies DIDs, public keys, and verifiable claims via encrypted peer-to-peer interactions. An identity (e.g. a person, organization or thing) creating a DID can publish that DID to an Indy immutable public ledger. The "DID" (a globally unique string) can then be looked up ("resolved") on the public ledger, and the information associated with the DID (called the "DID Doc") returned - the public key(s) and endpoint(s) associated with the DID. The private keys associated with the public keys are held by the owner of the DID in their Wallet. As long as the private keys are protected (a non-trivial, but manageable challenge), the DID cannot be used by anyone else. Using a decentralized system based on blockchain technology empowers users to securely publish their DIDs without a Central Authority.

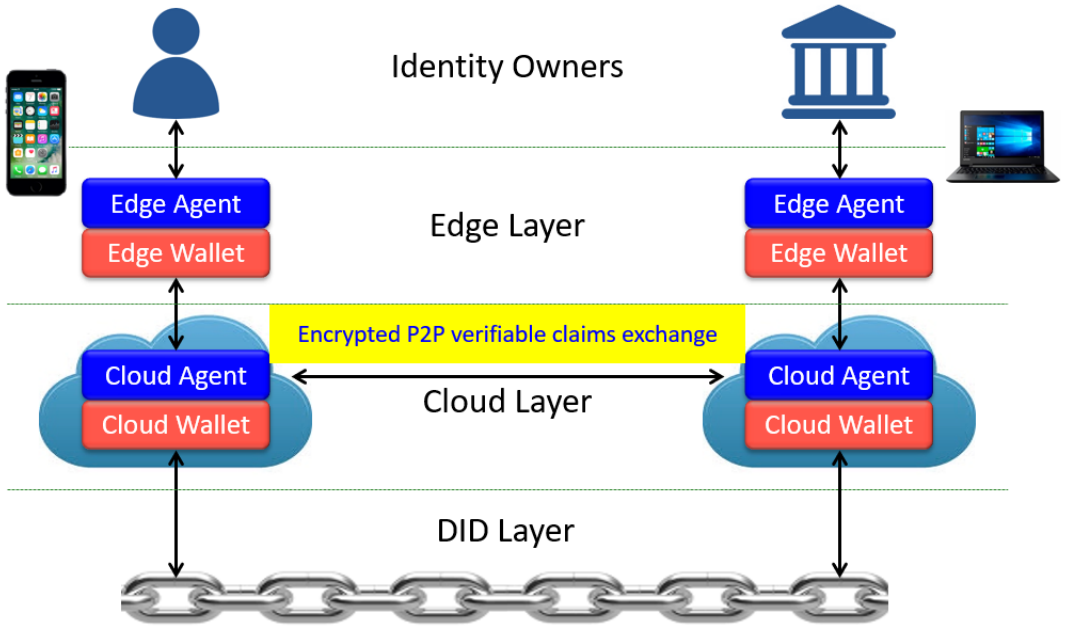

Figure 2.6: Example Of Some Communicating Indy Agents
(Licensed under CC By 4.0)

DIDs that are controlled by their owning entities and shared to establish relationships between entities. But how does each party know who it is that created and controls the DID? That's where the next part of the Hyperledger Indy story comes in - Verifiable Credentials.

2.3.2 Verifiable Credentials (VCs)

Verifiable credentials (VCs) [30] are trusted way to provide identity attributes about ourselves. Verifiable Credentials and their use closely mimics that of the real world. VCs take that model and put it online, in a trusted manner. The data flow for Verifiable Credentials is the same as with paper documents - Issuers give Verifiable Credentials to the Holder, and the Holder can prove them to Verifiers at any time Figure 2.7.

In the Figure 2.7

- holder is an entity that possesses one or more verifiable credentials
- issuer refers to an entity that acts by making claims about one or more subjects and generating a verifiable credential from these assertions.
- subject is an entity about which claims are made. Humans, animals, and objects are some examples of subjects.

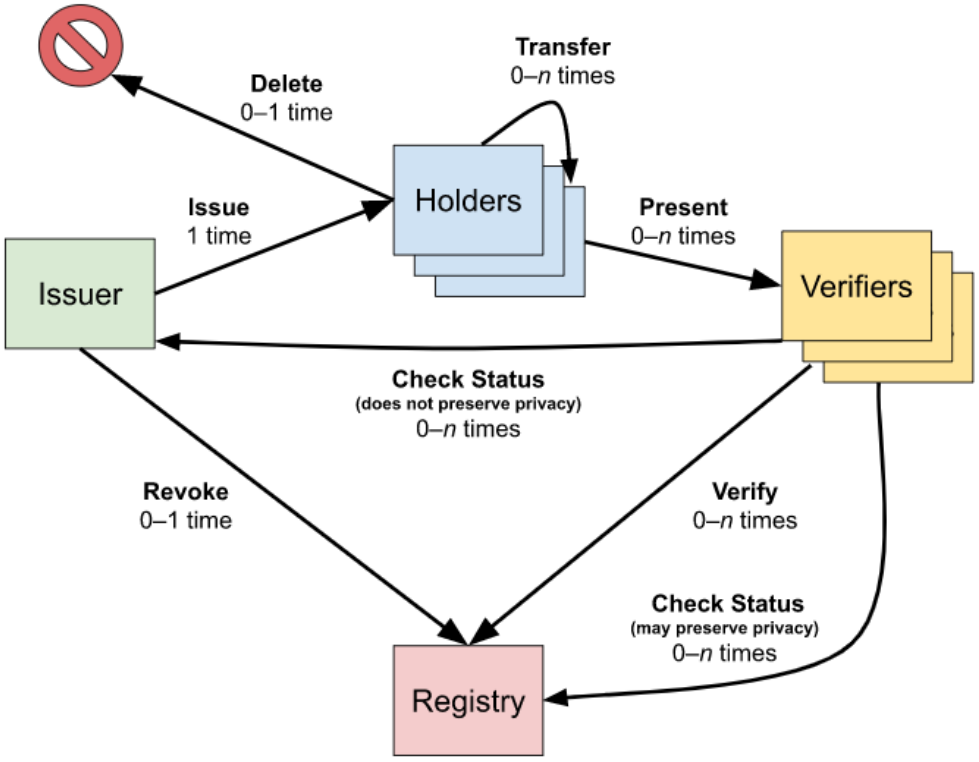

Figure 2.7: The roles and information flows in the verifiable credential ecosystem

- verifie is a role that an entity plays by getting and verifying one or more verifiable credentials. Employers, security staff, and websites are examples of verifiers.
- verifiable data registry is a function that a system may play by facilitating the generation and verification of identities, keys, and other necessary data, such as verifiable credential schemas, revocation registries, issuer public keys, and so on, that may be needed to utilize verifiable credentials. Trusted databases, decentralised databases, government ID datasets, and distributed ledgers are examples of verified data registries.

CHAPTER 3

CURVEZMQ PROTOCOL

CurveZMQ [21] is the protocol followed for client-to-node and node-to-node communication in distributed ledger technology used in identity systems such as Indy, which we mentioned in the previous section. In this section, the protocol details will be examined and it will be explained how the communication between the users in the network is provided. But first let's explain how CurveZMQ protocol come about and what is it compelling about.

CurveZMQ is an adaptation of the CurveCP protocol [8], which was designed by Daniel J. Bernstein. CurveCP is a more comprehensive, security-enhanced protocol for the Internet, which delivers improvements over TCP in a number of areas, and is designed to be an enhanced replacement for TCP. CurveZMQ uses the same high-speed elliptic curve cryptography that CurveCP uses and adopts the initial CurveCP "handshake" mechanism of initial key exchange. Combined with minor changes to adapt CurveCP for a connected and message-based use provides much of the security benefit of CurveCP in a more traditional flavor that is compatible with the ubiquitous TCP protocol.

3.1 Introduction

Curve ZMQ is a protocol created to establish secure communication on the web based on Curve PC security handshake. CurveZMQ employs the Curve25519 elliptic curve to provide high performance with small key sizes (256 bits). To ensure complete forward security, the protocol creates short-term session keys for each connection.

Session keys are stored in memory and are deleted when the connection is terminated.

CurveZMQ has two primary applications: To secure a single hop between client and server, and end-to-end security for a client and server over distrusted peers when TLS is insufficient. To send data from one peer to another using out-of-band transports like as email or file transfer, providing that both peers first conduct the initial handshake using a compatible transport.

The goal of technology design is to avoid:

- Eavesdropping, where an attacker watches a conversation without authorization. Every packets are sent into boxes that can only be opened by the genuine receiver having the secret key required.
- Fraudulent data, where an attacker creates a data packet that supposedly comes from one of the peer. Only when the actual transmitter knows the required secret key can create a valid box, thus creating a valid data packet.
- Change the input, where adversary located among the user and the service distorts the data packet in a particular way. When a package is altered in any way, the box it carries will not be opened, alerting the receiver to the fact that an assault has happened and trash the package.
- Data replay, where the adversary records data packets and forwards those later to mimic a legitimate pair. Each box is encrypted with a unique random number. Two types of random numbers used in CurveZMQ. The long random number protects the permanent key and has a good 16-byte random number generator. The short random number protects the temporary key, which is an 8-byte serial number. The attacker cannot reproduce any packets other than the hello packet, which has no effect on the server or the client. Your receiver will discard any other played packets.
- Amplification attack, where the evil party sends several short unauthenticated requests, yet utilizes a bogus source address and thus causes a large number of responses to be sent innocent parties, overwhelming them. Only Hello packages are not authenticated and they are padded to be larger than Cookie packets.
- MITM attack, that allows an evil party between a user and a service to play a "server" to the user and a "client" to the service to open all boxes by replacing

the key. Since the permanent key is known in advance, if the server authenticates the client, the attacker cannot successfully imitate the server or the client.

- A key theft attack, wherein the adversary captures encrypted content and subsequently obtains the private key, may be a physical attack on the peers. The peers use temporary keys that they discard when they end up the communication.
- Client identification, where the attacker traces the client's identity by obtaining its persistent public key from the data packet. The user only transmits this in an Initiate packet, which is secured by the temporary keys.
- DOS attack is an evil party's depletion of a server by causing it to perform expensive tasks before authentication. Before client send the Initiate packet he server does not allocate memory until a client sends the Initiate packet.

3.2 CurveZMQ's Main Functioning

Users and services have long-lasting persistent keys, and for each session, they generate and safely exchange short-lasting temporal keys. Each key is a public/private key pair, which follows the Curve25519 elliptic curve design to generate a fixed-length 256 bit key pair. These key pairs are used both as long-term and short-term keys in the elliptic curve security model. Also, since CurveZMQ closely follows CurveCP, as a result keys are 32 octets (256 bits) as previously indicated, and nonces are 24 octets. An encrypted box is 16 bytes bigger than the unencrypted data. The underlying cryptographical library enforces these sizes and acts as universal constants for CurveZMQ deploys.

The client requires the permanent public key of the server to establish a secure connection. Then a temporary key pair is generated and a HELLO instruction is sent to the server with its short-term public key.

When the server receives a HELLO command, it produces its own short-term key pair and encodes this new private key in a "cookie," which it delivers back to the client as a WELCOME command. The cookie is encrypted with server's minute key. This key only maintained by server. It also provides its short-term public key, which is encrypted and can only be read by the client. This short-term key pair is then discarded.

The server SHOULD maintain minimum state with the client until the client answers with an appropriate INITIATE command. In this manner, an adversary cannot compel the server to execute costly tasks prior to authentication, therefore depleting the server, which prevents Denial-of-Service attacks.

The client responds with an INITIATE instruction, which gives the server its cookie back as well as the client's permanent public key, which is encrypted as a "vouch" so that only the server can read it. The voucher is just client short-term public key is encrypted with server permanent public key. As we observed here, pre-distributed long-term keys serve as an authentication voucher for short-term keys.The server is now authenticated in the client's eyes, so it may send back metadata in the command.

The server reads the INITIATE and persisting client public key may now be authenticated. It also unwraps the cookie and retrieves the connection's short-term key pair. The client is now authenticated as regards the server, so that the server may securely provide its information. Both parties can then transmit messages and orders.

Both sides have own short-term keys, are mutually authenticated, and can freely exchange encrypted and authenticated data such as symmetric keys for data encryption. To describe this process with the Figure 3.1, the CurveCP notation being used "Box [X](C->S)" indicates a cryptographic box which encrypts X "from C to S," implying that only C can build it and only S can open it. Simply put, CurveCP protocol provides authenticated encryption for the message payload. Data in the Box is encrypted by the counterpart's short-term public key and authenticated by the peer's short-term private keys (or peer's long-term private key in case of message 1). Short-term session key pairs are generated only after successful authentication and are used to establish shared secret employing the ECDHE key agreement protocol. Shared secret is then used by xSalsa20 [12] stream cipher to ensure data confidentiality. Lastly, algorithm poly1305 [10] is used as a Message Authentication Code (MAC) confirming data integrity.

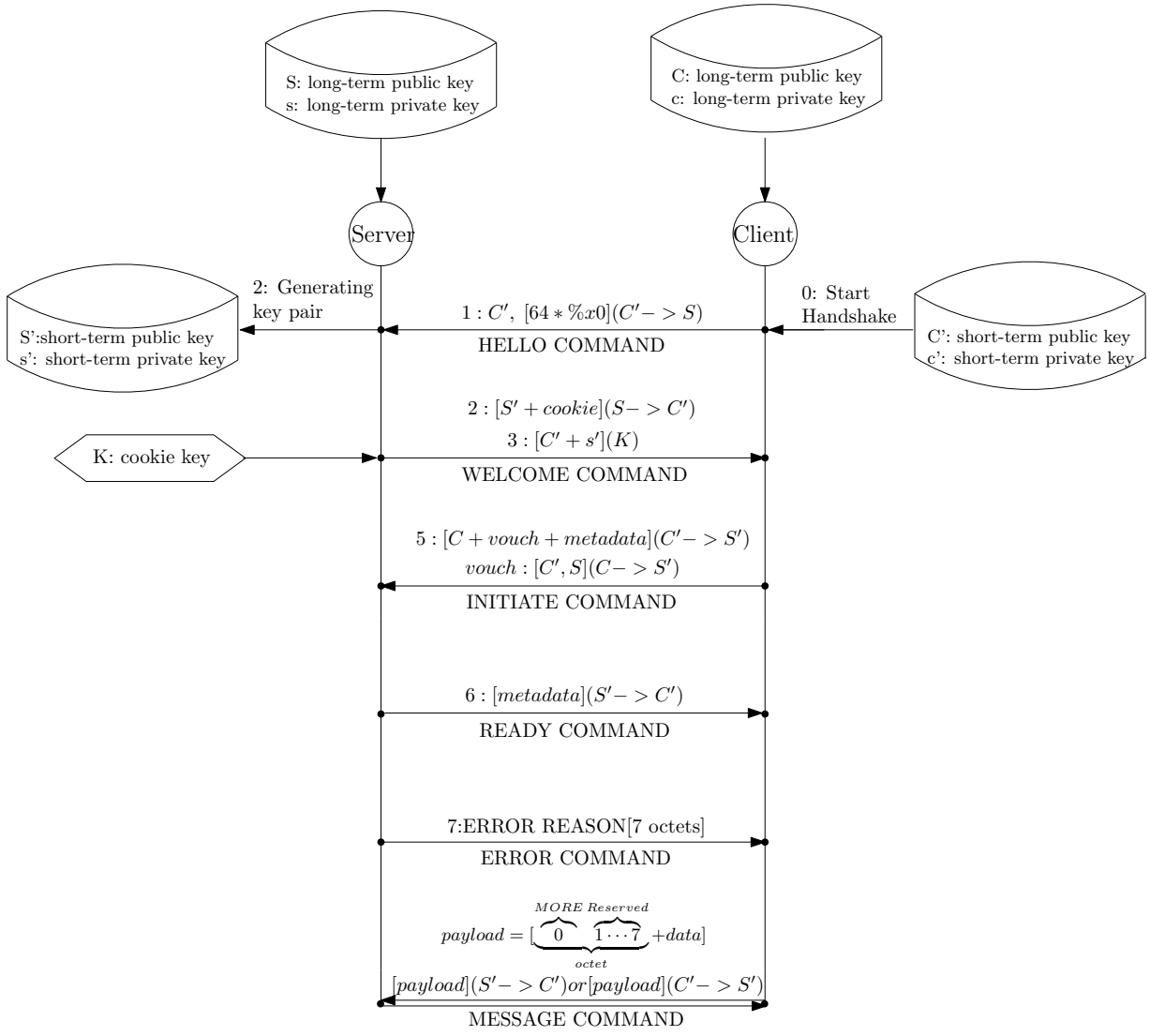

Figure 3.1: CurveZMQ Handshake

CHAPTER 4

HYPERLEDGER URSA

4.1 Motivation

Each blockchain has different requirements from the others that affect their choice in crypto libraries. As certain features in a blockchain gain popularity, other blockchains want to adopt these features in order to remain competitive. Different projects on Hyperledger want to get crypto features in another. Having parts of the code in different places is a maintenance nightmare [5] because fixing security issues found in a crypto application requires developers to review all projects and make sure every copy is updated. The solution to this problem is to collect all crypto applications and put them in one library that everyone uses. That's why the Ursa library was created.

On the other hand motivation for using the Ursa project is the Rust programming language which was used in the development of the Ursa project. Rust is specifically designed for writing secure code. The language and compiler support features that eliminate all of the most common mistakes programmers make. This makes it an ideal language for Ursa because it offers maximum security expected from a cryptographic components. Thus, Ursa adds extra security measures by wrapping crypto apps with Rust and can ensure that memory and data are always handled correctly and program execution is always predictable. Also Rust comes with the powerful software creation tool namely Cargo. With Cargo, you can provide an easy-to-use and therefore hard-to-confuse system for determining exactly which crypto applications should be included when compiling Ursa.
Moreover, Ursa uses openssl, libsodium and libsecp256k1 external dependencies.

There is two large libraries on Ursa namely: libursa [4, 6] and libzmix. The main pillar of our study is the library of libursa.

4.2 Libursa

Libursa library is designed for fundamental cryptography such as digital signatures, encryption methods and key exchange. Cryptographic primitives in the Libursa library are shown in Figure 4.1.

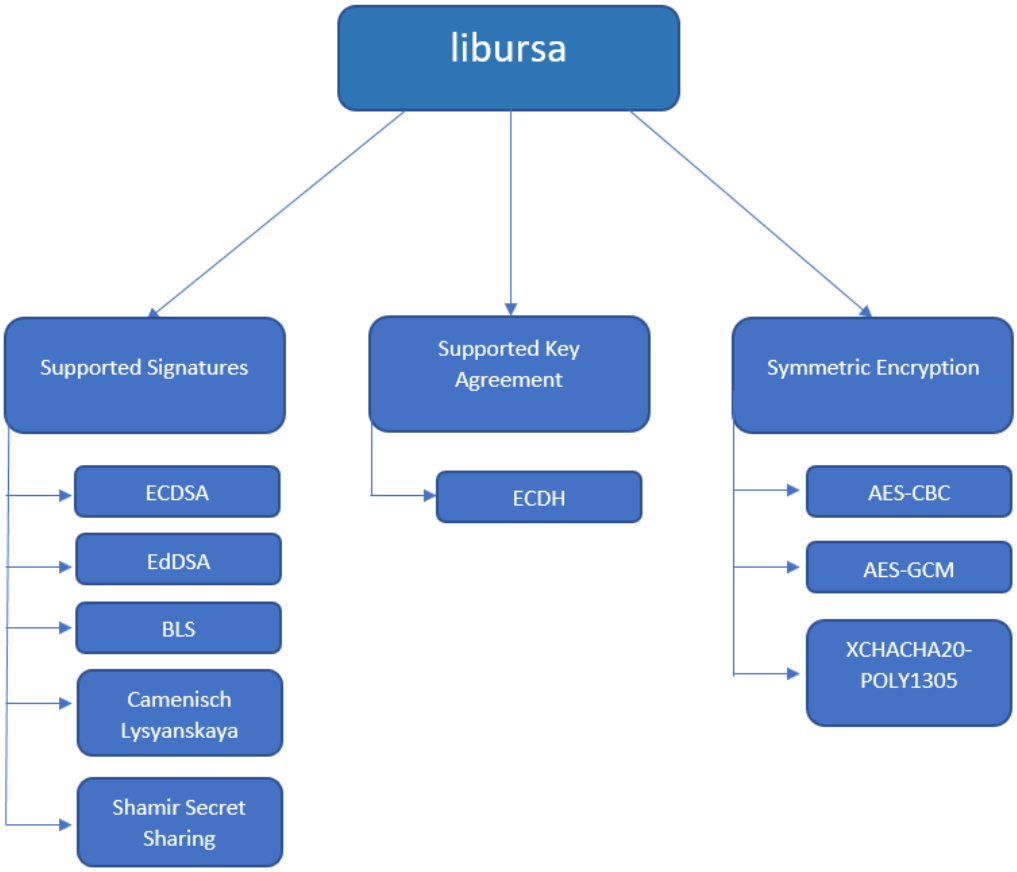

Figure 4.1: Cryptographic primitives inside Libursa Library Module

AES-CBC is supported for historical reason. It's older which means more compatible and as a plus, CBC doesn't fail as catastrophically if the IV is reused compare to GCM, and it can be faster if implemented on basic hardware. As for AES-GCM, is not a silver bullet for symmetric encryption but it's fast and secure if used correctly, and very versatile, hence its popularity. Finally AES CBC with HMAC is also supported because of MAC and it does not catastrophically fail for IV resued. On the supported

signature schemes Boneh-Lynn-Shacham signatures particular of interest. Imagine we have a block with 1000 transactions and every transaction contains a signature S_i, a public key P_i and a message that is signed m_i. Insted of storing all the signatures with BLS we can combine them. After all, we only care if all signatures in the block are valid. Aggregated signature will be just a sum of all signatures in the block and it's the primary reason of support.

4.2.1 Benchmarks

Benchmark tests for implementation of libursa library is performed respectively on an Ubuntu 16.04.7 LTS with Linux Kernel 4.15.0 machine running on VMware Workstation 16 Pro with host operating system Windows 10 running on Intel Xeon Silver 4116 Processor running at 2100 MHz, Arch Linux with Linux Kernel 5.13.8 running on Intel Xeon Silver 4116 Processor running at 2100 MHz and Fedora 34 workstation edition running on Intel Xenon E-2136 Processor at 3300 MHz and software is compiled respectively with rustc-1.51.1 and rustc-1.54.0. The benchmark results for cryptographic operations are given in Table 4.1. Although there is no special reason for using different compiler versions here, it is aimed to use the facilities [21] provided to the developers.

As you can see on Table 4.1, encryption/decryption and signing processes takes micro seconds whereas verification, which is highlighted with orange colour, takes milliseconds. Operation such as addition, rotation, xor which are ChaCha20-Poly1305 cipher suites based on are CPU friendly instructions as a result ChaCha is working fast as expected. As is the case with ChaCha, AES operations works fast enough but not as quite as ChaCha, but Intel processors have AES hardware support that makes AES operations cheap and tons of cryptanalysis on AES can make AES preferred choice over ChaCha. This results shown with green colour in the Table 4.1. Moreover, proof verification, highlighted with red colour, stands as the slowest operation among them. Because in the CKS schema proving knowledge of a valid credential requires exponentiation and pairing operations. At this point we move forward to look at how this crypto implementations perform on a anonymous credential protocol that put ahead as an integration test on libursa library.

Table 4.1: Benchmark Results of Cryptographic Operations

Cryptographic Operations Performance Results			
group	ubuntu	arch	fedora
Create small bls key pair	1318.6±31.48µs	1174.2±24.90µs	815.1±6.34µs
Create usual bls key pair	676.7±15.96µs	608.5±12.15µs	417.9±3.59µs
Encrypt/Decrypt for Aes128CbcHmac256 for 1024bytes	55.9±0.18µs	54.2±0.78µs	38.6±0.23µs
Encrypt/Decrypt for Aes128CbcHmac256 for 1048576 bytes	48.5±0.43ms	51.6±0.19ms	34.3±0.17ms
Encrypt/Decrypt for Aes128CbcHmac256 for 128 bytes	13.3±0.25µs	13.0±0.16µs	9.7±0.09µs
Encrypt/Decrypt for Aes128CbcHmac256 for 16384 bytes	742.8±5.24µs	781.8±2.96µs	525.9±3.23µs
Encrypt/Decrypt for Aes128Gcm for 1024 bytes	22.4±0.23µs	21.0±0.08µs	15.2±0.11µs
Encrypt/Decrypt for Aes128Gcm for 1048576 bytes	18.5±0.31ms	17.5±0.14ms	11.7±0.05ms
Encrypt/Decrypt for Aes128Gcm for 128 bytes	6.5±0.08µs	6.1±0.06µs	5.2±0.04µs
Encrypt/Decrypt for Aes128Gcm for 16384 bytes	291.1±3.84µs	272.1±1.94µs	185.0±2.13µs
Encrypt/Decrypt for Aes256CbcHmac512 for 1024 bytes	61.9±1.62µs	61.1±0.32µs	42.0±0.41µs
Encrypt/Decrypt for Aes256CbcHmac512 for 1048576 bytes	56.4±0.56ms	56.3±0.20ms	37.6±0.21ms
Encrypt/Decrypt for Aes256CbcHmac512 for 128 bytes	15.4±0.14µs	14.1±0.08µs	10.6±0.11µs
Encrypt/Decrypt for Aes256CbcHmac512 for 16384 bytes	864.3±6.15µs	856.8±3.35µs	576.2±5.12µs
Encrypt/Decrypt for Aes256Gcm for 1024 bytes	27.8±0.34µs	26.4±0.10µs	18.7±0.12µs
Encrypt/Decrypt for Aes256Gcm for 1048576 bytes	23.9±0.56ms	22.2±0.12ms	14.7±0.07ms
Encrypt/Decrypt for Aes256Gcm for 128 bytes	8.0±0.07µs	7.6±0.07µs	6.1±0.04µs
Encrypt/Decrypt for Aes256Gcm for 16384 bytes	364.1±3.41µs	344.4±1.17µs	232.0±1.26µs
Encrypt/Decrypt for XChaCha20Poly1305 for 1024 bytes	8.9±0.23µs	9.6±0.05µs	7.5±0.04µs
Encrypt/Decrypt for XChaCha20Poly1305 for 1048576 bytes	7.1±0.11ms	5.1±0.04ms	3.3±0.02ms
Encrypt/Decrypt for XChaCha20Poly1305 for 128 bytes	3.1±0.06µs	5.4±0.05µs	4.7±0.04µs
Encrypt/Decrypt for XChaCha20Poly1305 for 16384 bytes	111.3±1.36µs	77.8±0.71µs	53.3±0.38µs
Sign small bls	550.7±8.17µs	487.3±1.43µs	335.7±1.48µs
Sign usual bls	1954.1±33.06µs	1745.9±15.31µs	1218.0±9.03µs
Small bls aggregate signatures no rogue key protection	17.9±0.24µs	15.3±0.32µs	10.5±0.16µs
Small bls aggregate signatures no rogue key protection verify	3.6±0.11ms	3.1±0.01ms	2.2±0.01ms
Small bls aggregate signatures rogue key protection verify	3.5±0.03ms	3.0±0.01ms	2.2±0.01ms
Small bls multisignature verify	16.1±0.08ms	14.0±0.03ms	10.0±0.05ms
Small bls sign with rogue key protection	1532.2±23.70µs	1389.5±6.45µs	962.5±11.13µs
Usual bls aggregate signatures no rogue key protection	45.2±0.55µs	40.0±0.12µs	28.0±0.22µs
Usual bls aggregate signatures no rogue key protection verify	4.4±0.03ms	3.8±0.01ms	2.6±0.02ms
Usual bls aggregate signatures rogue key protection verify	4.4±0.03ms	3.9±0.01ms	2.7±0.01ms
Usual bls multisignature verify	24.0±0.31ms	21.0±0.11ms	14.6±0.14ms
Usual bls sign with rogue key protection	3.8±0.02ms	3.3±0.01ms	2.3±0.03ms
Verify small bls	3.7±0.14ms	3.1±0.01ms	2.2±0.02ms
Verify usual bls	4.2±0.04ms	3.7±0.02ms	2.6±0.02ms
cks revocation proof generation with on demand issuance/(10, 2)	124.7±3.36ms	104.2±0.22ms	71.9±0.32ms
cks revocation proof generation with on demand issuance/(100, 2)	124.6±0.69ms	104.6±0.36ms	71.8±0.34ms
cks revocation verify proof with on demand issuance/(10, 2)	129.1±1.30ms	109.9±0.77ms	75.4±0.43ms
cks revocation verify proof with on demand issuance/(100, 2)	130.2±1.42ms	109.1±0.27ms	73.3±0.16ms

4.3 Anonymous Credentials

The anonymous credentials [23] idea enables individuals to demonstrate that their identity fulfills specific criteria in an unrelated manner without exposing other identifying data. Before moving on to the performance results of the anonymous credential protocol, we explain the implementation steps of the protocol explaining how the credentials granted by different publishers (issuers) to numerous rights holders are created to be presented to the various affiliated parties (verifier).

- Prover chooses a master secret to prove the attributes owned by himself/herself in all credentials [26].
- A credential schema [10, 31], which is a BTree set to add the attributes, is chosen by issuer.
- Credential definition is created by issuer. The issuer's public-private key pair,

the revocation public-private key pair and proof of correctness of issuer is defined.

- Issuer creates nonce used by Prover to create correctness proof for blinded secrets.
- Credential values is created by issuer. Issuer figures out how to map the attributes in the schema to a flat array of integer values and add them to credential schema BTree.
- Hidden attributes are blinded by prover. Prover checks the correctness proof of issuer, generates blinded primary credential factors, blinded revocation credential secrets and correctness proof of blinded credential secrets [24].
- Credential values are signed by issuer. First, issuer verifies the inputs sent by prover and then generates and signs credential context.
- Prover processes credential signature. Prover checks the signature of issuer and stores the credential and related non-revocation credential.
- Prover creates nonce used by Issuer to create correctness proof for signature.
- In the system, there exist three types of credential: known credentials, hidden credentials and predicates. For predicates, there are commitments created by prover. In the process of sub proof request creation by verifier, BTree sets for revealed credentials and predicates are created and then verifier adds the proof requests for revealed attributes and predicates
- Prover checks the proof requests sent by verifier. Then creates and initializes the proofs and sent them to verifier [25].
- Verifier verifies the proofs sent by prover by using relevant variables.

The mathematical background under the application steps are given in the following sections.

4.3.1 Schema Attributes

Issuer defines the primary credential schema $\mathcal{S}$ with l attributes $m_1, m_2, \ldots, m_l$ and the set of hidden attributes $A_h \subset \{1, 2, \ldots, l\}$. In Sovrin, m_1 is reserved for the link secret of the holder, m_2 is reserved for the context – the enumerator for the holders, m_3 is reserved for the policy address I. By default, $\{1, 3\} \subset A_h$ whereas $2 \notin A_h$.

Issuer defines the non-revocation credential with 2 attributes m_1, m_2. In Sovrin, $A_h = \{1\}$ and m_1 is reserved for the link secret of the holder, m_2 is reserved for the context – the enumerator for the holders.

4.3.2 Schema Primary Credential Cryptographic Setup

In Sovrin, issuers use CL-signatures for primary credentials, although other signature types will be supported too.

For the CL-signatures issuer generates:

1. Random 1536-bit primes p', q' such that $p \leftarrow 2p'+1$ and $q \leftarrow 2q'+1$ are primes too. Then compute $n \leftarrow pq$.
2. A random quadratic residue S modulo n;
3. Random $x_Z, x_{R_1}, \ldots, x_{R_l} \in [2; p'q' - 1]$

Issuer computes

$$Z \leftarrow S^{x_Z} \pmod{n}; \quad \{R_i \leftarrow S^{x_{R_i}} \pmod{n}\}_{1 \leq i \leq l}; \tag{4.1}$$

The issuer's public key is $P_k = (n, S, Z, \{R_i\}_{1 \leq i \leq l})$ and the private key is $s_k = (p, q)$.

4.3.3 Schema Optional: Setup Correctness Proof

1. Issuer generates random $\widetilde{x_Z}, \widetilde{x_{R_1}}, \ldots, \widetilde{x_{R_l}} \in [2; p'q' - 1]$;
2. Computes

$$\widetilde{Z} \leftarrow S^{\widetilde{x_Z}} \pmod{n}; \qquad \{\widetilde{R_i} \leftarrow S^{\widetilde{x_{R_i}}} \pmod{n}\}_{1 \leq i \leq l}; \tag{4.2}$$

$$c \leftarrow H_I(Z||\widetilde{Z}||\{R_i, \widetilde{R_i}\}_{i \leq i \leq l}); \tag{4.3}$$

$$\widehat{x_Z} \leftarrow \widetilde{x_Z} + cx_Z \pmod{p'q'}; \quad \{\widehat{x_{R_i}} \leftarrow \widetilde{x_{R_i}} + cx_{R_i} \pmod{p'q'}\}_{1 \leq i \leq l}; \tag{4.4}$$

Here H_I is the issuer-defined hash function, by default SHA-256.

3. Proof $\mathcal{P}_I$ of correctness is $(c, \widehat{x_Z}, \{\widehat{x_{R_i}}\}_{1 \leq i \leq l})$

4.3.4 Schema Non-revocation Credential Cryptographic Setup

In Sovrin, issuers use CKS accumulator and signatures to track revocation status of primary credentials, although other signature types will be supported too. Each primary credential is given an index from 1 to L.

The CKS accumulator is used to track revoked primary credentials, or equivalently, their indices. The accumulator contains up to L indices of credentials. If issuer has to issue more credentials, another accumulator is prepared, and so on. Each accumulator A has an identifier I_A.

Issuer chooses

- Groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q;
- Type-3 pairing operation $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.
- Generators: g for $\mathbb{G}_1$, g' for $\mathbb{G}_2$.

Issuer:

1. Generates
   1.1. Random $h, h_0, h_1, h_2, \widetilde{h} \in \mathbb{G}_1$;
   1.2. Random $u, \widehat{h} \in \mathbb{G}_2$;
   1.3. Random $sk, x \pmod q$.
2. Computes

$$pk \leftarrow g^{sk}; \qquad\qquad y \leftarrow \widehat{h}^x.$$

The revocation public key is $P_r = (h, h_0, h_1, h_2, \widetilde{h}, \widehat{h}, u, pk, y)$ and the secret key is (x, sk).

4.3.4.1 New Accumulator Setup

To create a new accumulator A, issuer:

1. Generates random $\gamma \pmod q$.
2. Computes
   2.1. $g_1, g_2, \ldots, g_L, g_{L+2}, \ldots, g_{2L}$ where $g_i = g^{\gamma^i}$.
   2.2. $g'_1, g'_2, \ldots, g'_L, g'_{L+2}, \ldots, g'_{2L}$ where $g'_i = g'^{\gamma^i}$.
   2.3. $z = (e(g, g'))^{\gamma^{L+1}}$.
3. Set $V \leftarrow \emptyset$, $\mathrm{acc} \leftarrow 1$.

The accumulator public key is $P_a = (z)$ and secret key is (γ).

Issuer publishes (P_a, V) on the ledger. The accumulator identifier is $ID_a = z$.

4.3.5 Issuance Holder Setup

Holder:

- Loads credential schema $\mathcal{S}$.
- Sets hidden attributes $\{m_i\}_{i \in A_h}$.
- Establishes a connection with issuer and gets nonce n_0 either from issuer or as a precomputed value. Holder is known to issuer with identifier $\mathcal{H}$.

Holder prepares data for primary credential:

1. Generate random 3152-bit v'.
2. Generate random 593-bit $\{\widetilde{m_i}\}_{i \in A_h}$, and random 3488-bit $\widetilde{v'}$.
3. Compute taking S, Z, R_i from P_k:

$$U \leftarrow (S^{v'}) \prod_{i \in A_h} R_i^{m_i} \pmod n; \tag{4.5}$$

4. For proving correctness of U, compute

$$\widetilde{U} \leftarrow (S^{\widetilde{v'}}) \prod_{i \in A_h} R_i^{\widetilde{m_i}} \pmod{n}; \tag{4.6}$$

$$c \leftarrow H(U||\widetilde{U}||n_0); \qquad \widehat{v'} \leftarrow \widetilde{v'} + cv'; \tag{4.7}$$

$$\{\widehat{m_i} \leftarrow \widetilde{m_i} + cm_i\}_{i \in A_h}; \tag{4.8}$$

5. Generate random 80-bit nonce n_1
6. Send $\{U, c, \widehat{v'}, \{\widehat{m_i}\}_{i \in A_h}, n_1\}$ to the issuer.

Holder prepares for non-revocation credential:

1. Load issuer's revocation key P_R and generate random $s'_R \bmod q$.
2. Compute $U_R \leftarrow h_2^{s'_R}$ taking h_2 from P_R.
3. Send U_R to the issuer.
4. For proving correctness of U_R
   - generate random $\widetilde{s'_R} \bmod q$ and compute $\widetilde{U_R} \leftarrow h_2{}^{\widetilde{s'_R}}$
   - Compute above challenge c as $c \leftarrow H(U||\widetilde{U}||U_R||\widetilde{U_R}||n_0)$ instead of $c \leftarrow H(U||\widetilde{U}||n_0)$
   - Compute $\widehat{s'_R} \leftarrow \widetilde{s'_R} + cs'_R$
   - Send c and $\widehat{s'_R}$ to issuer

4.3.5.1 Optional: Issuer Proof of Setup Correctness

To verify the proof $\mathcal{P}_i$ of correctness, holder computes

$$\widehat{Z} \leftarrow Z^{-c} S^{\widehat{x_Z}} \pmod{n}; \quad \{\widehat{R_i} \leftarrow R_i^{-c} S^{\widehat{x_{R_i}}} \pmod{n}\}_{1 \le i \le l};$$

and verifies

$$c = H_I(Z||\widehat{Z}||\{R_i, \widehat{R_i}\}_{1 \le i \le l})$$

.

4.3.6 Primary Credential Issuance

Issuer verifies the correctness of holder's input:

1. Compute

$$\widehat{U} \leftarrow (U^{-c}) \prod_{i \in A_h} R_i^{\widehat{m_i}} (S^{\widehat{v'}}) \pmod{n}; \tag{4.9}$$

2. Verify $c = H(U||\widehat{U}||n_0)$

3. Verify that $\widehat{v'}$ is a 673-bit number, $\{\widehat{m_i}, \widehat{r_i}\}_{i \in \mathcal{A}_c}$ are 594-bit numbers.

4. If a revocable credential is requested

   - Compute $\widehat{U_R} = {U_R}^{-c} {h_2}^{\widehat{s'_R}}$
   - Verify that c equals $H(U||\widehat{U}||U_R||\widehat{U_R}||n_0)$ instead of $H(U||\widehat{U}||n_0)$

Issuer prepare the credential:

1. Assigns index $i < L$ to holder, which is one of not yet taken indices for the issuer's current accumulator A. Compute $m_2 \leftarrow H(i||\mathcal{H})$ and store information about holder and the value i in a local database.

2. Set, possibly in agreement with holder, the values of disclosed attributes, i.e. with indices from A_k.

3. Generate random 2724-bit number v'' with most significant bit equal 1 and random prime e such that

$$2^{596} \leq e \leq 2^{596} + 2^{119}. \tag{4.10}$$

4. Compute

$$Q \leftarrow \frac{Z}{US^{v''} \prod_{i \in \mathcal{A}_k} R_i^{m_i} \pmod{n}}; \tag{4.11}$$

$$A \leftarrow Q^{e^{-1} \pmod{p'q'}} \pmod{n}; \tag{4.12}$$

5. Generate random $r < p'q'$;

6. Compute

$$\widehat{A} \leftarrow Q^r \pmod{n}; \tag{4.13}$$

$$c' \leftarrow H(Q||A||\widehat{A}||n_1); \tag{4.14}$$

$$s_e \leftarrow r - c'e^{-1} \pmod{p'q'}; \tag{4.15}$$

7. Send the primary pre-credential $(\{m_i\}_{i \in A_k}, A, e, v'', s_e, c')$ to the holder.

4.3.7 Non-revocation Credential Issuance

Issuer:

1. Generate random numbers $s'', c \bmod q$.

2. Take m_2 from the primary credential he is preparing for holder.

3. Take A as the accumulator value for which index i was taken. Retrieve current set of non-revoked indices V.

4. Compute:

$$\sigma \leftarrow \left(h_0 h_1^{m_2} \cdot U_R \cdot g_i \cdot h_2^{s''}\right)^{\frac{1}{x+c}}; \qquad w \leftarrow \prod_{j \in V} g'_{L+1-j+i}; \tag{4.16}$$

$$\sigma_i \leftarrow g'^{1/(sk+\gamma^i)}; \qquad u_i \leftarrow u^{\gamma^i}; \tag{4.17}$$

$$A \leftarrow A \cdot g'_{L+1-i}; \qquad V \leftarrow V \cup \{i\}; \tag{4.18}$$

$$\text{wit}_i \leftarrow \{\sigma_i, u_i, g_i, w, V\}. \tag{4.19}$$

5. Send the non-revocation pre-credential $(I_A, \sigma, c, s'', \text{wit}_i, g_i, g'_i, i)$ to holder.

6. Publish updated V, A on the ledger.

4.3.8 Issuance Storing Credentials

Holder works with the primary pre-credential :

1. Compute $v \leftarrow v' + v''$.

2. Verify e is prime and satisfies Eq. (4.10).

3. Compute

$$Q \leftarrow \frac{Z}{S^v \prod_{i \in C_s} R_i^{m_i}} \pmod{n}; \tag{4.20}$$

4. Verify $Q = A^e \pmod{n}$

5. Compute [1]

$$\widehat{A} \leftarrow A^{c' + s_e \cdot e} \pmod{n}. \tag{4.21}$$

6. Verify $c' = H(Q||A||\widehat{A}||n_2)$.

7. Store *primary credential* $C_p = (\{m_i\}_{i \in C_s}, A, e, v)$.

Holder takes the non-revocation pre-credential $(I_A, \sigma, c, s'', \mathrm{wit}_i, g_i, g_i', i)$ computes $s_R \leftarrow s' + s''$ and stores the non-revocation credential $C_{NR} \leftarrow (I_A, \sigma, c, s, \mathrm{wit}_i, g_i, g_i', i)$.

4.3.9 Issuance Non revocation proof of correctness

Holder computes

$$\frac{e(g_i, acc_V)}{e(g, w)} \stackrel{?}{=} z; \tag{4.22}$$

$$e(pk \cdot g_i, \sigma_i) \stackrel{?}{=} e(g, g'); \tag{4.23}$$

$$e(\sigma, y \cdot \widehat{h}^c) \stackrel{?}{=} e(h_0 \cdot h_1^{m_2} h_2^s g_i, \widehat{h}). \tag{4.24}$$

4.3.10 Revocation

Issuer identifies a credential to be revoked in the database and retrieves its index i, the accumulator value A, and valid index set V. Then he proceeds:

[1] We have removed factor $S^{v' s_e}$ here from computing of $\widehat{A}$ as it seems to be a typo in the Idemix spec.

1. Set $V \leftarrow V \setminus \{i\}$;
2. Compute $A \leftarrow A/g'_{L+1-i}$.
3. Publish $\{V, A\}$.

4.3.11 Presentation Proof Request

Verifier sends a proof request, where it specifies the ordered set of d credential schemas $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_d\}$, so that the holder should provide a set of d credential pairs (C_p, C_{NR}) that correspond to these schemas.

Let credentials in these schemas contain X attributes in total. Suppose that the request makes to open x_1 attributes, makes to prove x_2 equalities $m_i = m_j$ (from possibly distinct schemas) and makes to prove x_3 predicates of form $m_i >\leq\geq< z$. Then effectively $X - x_1$ attributes are unknown (denote them A_h), which form $x_4 = (X - x_1 - x_2)$ equivalence classes. Let ϕ map A_h to $\{1, 2, \ldots, x_4\}$ according to this equivalence. Let A_v denote the set of indices of x_1 attributes that are disclosed.

The proof request also specifies A_h, ϕ, A_v and the set $\mathcal{D}$ of predicates. Along with a proof request, Verifier also generates and sends 80-bit nonce n_1.

4.3.12 Presentation Proof Preparation

Holder prepares all credential pairs (C_p, C_{NR}) to submit:

1. Generates x_4 random 592-bit values $\widetilde{y_1}, \widetilde{y_2}, \ldots, \widetilde{y_{x_4}}$ and set $\widetilde{m_j} \leftarrow \widetilde{y_{\phi(j)}}$ for $j \in \mathcal{A}_h$.
2. Create empty sets $\mathcal{T}$ and $\mathcal{C}$.
3. For all credential pairs (C_p, C_{NR}) executes Section 4.3.12.
4. Executes Section 4.3.12.1 once.
5. For all credential pairs (C_p, C_{NR}) executes Section 4.3.12.2.
6. Executes Section 4.3.12.2 once.

Verifier:

1. For all credential pairs (C_p, C_{NR}) executes Section 4.3.13.3.
2. Executes Section 4.3.13.4 once.

Non-revocation proof Holder:

1. Load issuer's public revocation key $p = (h, h_1, h_2, \widetilde{h}, \widehat{h}, u, pk, y)$.
2. Load the non-revocation credential $C_{NR} \leftarrow (I_A, \sigma, c, s, \mathrm{wit}_i, g_i, g'_i, i)$;
3. Obtain recent V, acc (from Verifier, Sovrin link, or elsewhere).
4. Update C_{NR}:

$$w \leftarrow w \cdot \frac{\prod_{j \in V \setminus V_{old}} g'_{L+1-j+i}}{\prod_{j \in V_{old} \setminus V} g'_{L+1-j+i}};$$
$$V_{old} \leftarrow V.$$

   Here V_{old} is taken from wit_i and updated there.

5. Select random $\rho, \rho', r, r', r'', r''', o, o' \bmod q$;
6. Compute

$$E \leftarrow h^{\rho}\widetilde{h}^{o} \qquad D \leftarrow g^{r}\widetilde{h}^{o'}; \tag{4.25}$$
$$A \leftarrow \sigma\widetilde{h}^{\rho} \qquad \mathcal{G} \leftarrow g_i\widetilde{h}^{r}; \tag{4.26}$$
$$\mathcal{W} \leftarrow w\widehat{h}^{r'} \qquad \mathcal{S} \leftarrow \sigma_i\widehat{h}^{r''} \tag{4.27}$$
$$\mathcal{U} \leftarrow u_i\widehat{h}^{r'''} \tag{4.28}$$

   and adds these values to $\mathcal{C}$.

7. Compute

$$m \leftarrow \rho \cdot c \mod q; \qquad t \leftarrow o \cdot c \mod q; \tag{4.29}$$
$$m' \leftarrow r \cdot r'' \mod q; \qquad t' \leftarrow o' \cdot r'' \mod q; \tag{4.30}$$

   and adds these values to $\mathcal{C}$.

8. Generate random $\widetilde{\rho}, \widetilde{o}, \widetilde{o'}, \widetilde{c}, \widetilde{m}, \widetilde{m'}, \widetilde{t}, \widetilde{t'}, \widetilde{m_2}, \widetilde{s}, \widetilde{r}, \widetilde{r'}, \widetilde{r''}, \widetilde{r'''}, \mathrm{mod} q$.

9. Compute

$$\overline{T_1} \leftarrow h^{\widetilde{\rho}}\widetilde{h}^{\widetilde{o}} \qquad \overline{T_2} \leftarrow E^{\widetilde{c}}h^{-\widetilde{m}}\widetilde{h}^{-\widetilde{t}} \tag{4.31}$$

$$\overline{T_3} \leftarrow e(A,\widehat{h})^{\widetilde{c}} \cdot e(\widetilde{h},\widehat{h})^{\widetilde{r}} \cdot e(\widetilde{h},y)^{-\widetilde{\rho}} \cdot e(\widetilde{h},\widehat{h})^{-\widetilde{m}} \cdot e(h_1,\widehat{h})^{-\widetilde{m_2}} \cdot e(h_2,\widehat{h})^{-\widetilde{s}} \tag{4.32}$$

$$\overline{T_4} \leftarrow e(\widetilde{h},\mathrm{acc})^{\widetilde{r}} \cdot e(1/g,\widehat{h})^{\widetilde{r'}} \quad \overline{T_5} \leftarrow g^{\widetilde{r}}\widetilde{h}^{\widetilde{o'}} \tag{4.33}$$

$$\overline{T_6} \leftarrow D^{\widetilde{r''}}g^{-\widetilde{m'}}\widetilde{h}^{-\widetilde{t'}} \qquad \overline{T_7} \leftarrow e(pk\cdot\mathcal{G},\widehat{h})^{\widetilde{r''}} \cdot e(\widetilde{h},\widehat{h})^{-\widetilde{m'}} \cdot e(\widetilde{h},\mathcal{S})^{\widetilde{r}} \tag{4.34}$$

$$\overline{T_8} \leftarrow e(\widetilde{h},u)^{\widetilde{r}} \cdot e(1/g,\widehat{h})^{\widetilde{r'''}} \tag{4.35}$$

and add these values to $\mathcal{T}$.

Validity proof

Holder:

1. Generate a random 592-bit number $\widetilde{m_j}$ for each $j \in \mathcal{A}_{\overline{r}}$.

2. For each credential $C_p = (\{m_j\}, A, e, v)$ and issuer's public key pk_I:

   2.1. Choose random 3152-bit r.

   2.2. Take n, S from pk_I compute

$$A' \leftarrow AS^r \pmod{n} \text{ and } v' \leftarrow v - e\cdot r \text{ as integers;} \tag{4.36}$$

   and add to $\mathcal{C}$.

   2.3. Compute $e' \leftarrow e - 2^{596}$.

   2.4. Generate random 456-bit number $\widetilde{e}$.

   2.5. Generate random 3748-bit number $\widetilde{v}$.

2.6. Compute

$$T \leftarrow (A')^{\widetilde{e}} \left(\prod_{j \in \mathcal{A}_{\overline{r}}} R_j^{\widetilde{m_j}} \right) (S^{\widetilde{v}}) \pmod{n} \tag{4.37}$$

and add to $\mathcal{T}$.

3. Load Z, S from issuer's public key.

4. For each predicate p where the operator $*$ is one of $>, \geq, <, \leq$.

4.1. Calculate Δ such that:

$$\Delta \leftarrow \begin{cases} z_j - m_j; & \text{if } * \equiv \leq \\ z_j - m_j - 1; & \text{if } * \equiv < \\ m_j - z_j; & \text{if } * \equiv \geq \\ m_j - z_j - 1; & \text{if } * \equiv > \end{cases}$$

4.2. Calculate a such that:

$$a \leftarrow \begin{cases} -1 & \text{if } * \equiv \leq or < \\ 1 & \text{if } * \equiv \geq or > \end{cases}$$

4.3. Find (possibly by exhaustive search) u_1, u_2, u_3, u_4 such that:

$$\Delta = (u_1)^2 + (u_2)^2 + (u_3)^2 + (u_4)^2 \tag{4.38}$$

4.4. Generate random 2128-bit numbers $r_1, r_2, r_3, r_4, r_\Delta$.

4.5. Compute

$$\{T_i \leftarrow Z^{u_i} S^{r_i} \pmod{n}\}_{1 \leq i \leq 4}; \tag{4.39}$$

$$T_\Delta \leftarrow Z^{\Delta} S^{r_\Delta} \pmod{n}; \tag{4.40}$$

and add these values to $\mathcal{C}$ in the order $T_1, T_2, T_3, T_4, T_\Delta$.

4.6. Generate random 592-bit numbers $\widetilde{u_1}, \widetilde{u_2}, \widetilde{u_3}, \widetilde{u_4}$.

4.7. Generate random 672-bit numbers $\widetilde{r_1}, \widetilde{r_2}, \widetilde{r_3}, \widetilde{r_4}, \widetilde{r_\Delta}$.

4.8. Generate random 2787-bit number $\widetilde{\alpha}$

4.9. Compute

$$\{\overline{T_i} \leftarrow Z^{\widetilde{u_i}} S^{\widetilde{r_i}} \pmod{n}\}_{1 \leq i \leq 4}; \tag{4.41}$$

$$\overline{T_\Delta} \leftarrow Z^{\widetilde{m_j}} S^{\widetilde{ar_\Delta}} \pmod{n}; \tag{4.42}$$

$$Q \leftarrow (S^{\widetilde{\alpha}}) \prod_{i=1}^{4} T_i^{\widetilde{u_i}} \pmod{n}; \tag{4.43}$$

and add these values to $\mathcal{T}$ in the order $\overline{T_1}, \overline{T_2}, \overline{T_3}, \overline{T_4}, \overline{T_\Delta}, Q$.

4.3.12.1 Hashing

Holder computes challenge hash

$$c_H \leftarrow H(\mathcal{T}, \mathcal{C}, n_1); \tag{4.44}$$

and sends c_H to Verifier.

4.3.12.2 Final preparation

Holder:

1. For non-revocation credential C_{NR} compute:

$$\widehat{\rho} \leftarrow \widetilde{\rho} - c_H \rho \bmod q \qquad \widehat{o} \leftarrow \widetilde{o} - c_H \cdot o \bmod q$$

$$\widehat{c} \leftarrow \widetilde{c} - c_H \cdot c \bmod q \qquad \widehat{o'} \leftarrow \widetilde{o'} - c_H \cdot o' \bmod q$$

$$\widehat{m} \leftarrow \widetilde{m} - c_H m \bmod q \qquad \widehat{m'} \leftarrow \widetilde{m'} - c_H m' \bmod q$$

$$\widehat{t} \leftarrow \widetilde{t} - c_H t \bmod q \qquad \widehat{t'} \leftarrow \widetilde{t'} - c_H t' \bmod q$$

$$\widehat{m_2} \leftarrow \widetilde{m_2} - c_H m_2 \bmod q \qquad \widehat{s} \leftarrow \widetilde{s} - c_H s \bmod q$$

$$\widehat{r} \leftarrow \widetilde{r} - c_H r \bmod q \qquad \widehat{r'} \leftarrow \widetilde{r'} - c_H r' \bmod q$$

$$\widehat{r''} \leftarrow \widetilde{r''} - c_H r'' \bmod q \qquad \widehat{r'''} \leftarrow \widetilde{r'''} - c_H r''' \bmod q.$$

and add them to $\mathcal{X}$.

2. For primary credential C_p compute:

$$\widehat{e} \leftarrow \widetilde{e} + c_H e'; \tag{4.45}$$

$$\widehat{v} \leftarrow \widetilde{v} + c_H v'; \tag{4.46}$$

$$\{\widehat{m}_j \leftarrow \widetilde{m_j} + c_H m_j\}_{j \in \mathcal{A}_{\overline{r}}}; \tag{4.47}$$

The values $Pr_C = (\widehat{e}, \widehat{v}, \{\widehat{m_j}\}_{j \in \mathcal{A}_{\overline{r}}}, A')$ are the ***sub-proof*** for credential C_p.

3. For each predicate p compute:

$$\{\widehat{u_i} \leftarrow \widetilde{u_i} + c_H u_i\}_{1 \le i \le 4}; \tag{4.48}$$

$$\{\widehat{r_i} \leftarrow \widetilde{r_i} + c_H r_i\}_{1 \le i \le 4}; \tag{4.49}$$

$$\widehat{r_\Delta} \leftarrow \widetilde{r_\Delta} + c_H r_\Delta; \tag{4.50}$$

$$\widehat{\alpha} \leftarrow \widetilde{\alpha} + c_H(r_\Delta - u_1 r_1 - u_2 r_2 - u_3 r_3 - u_4 r_4); \tag{4.51}$$

The values $Pr_p = (\{\widehat{u_i}\}, \{\widehat{r_i}\}, \widehat{r_\Delta}, \widehat{\alpha}, \widehat{m_j})$ are the sub-proof for predicate p.

4.3.12.3 Sending

Holder sends $(c_H, \mathcal{X}, \{Pr_C\}, \{Pr_p\}, \mathcal{C})$ to the Verifier.

4.3.13 Presentation Verification

For the credential pair (C_p, C_{NR}), Verifier retrieves relevant variables from $\mathcal{X}, \{Pr_C\}, \{Pr_p\}, \mathcal{C}$.

4.3.13.1 Non-revocation check

Verifier computes

$$\widehat{T_1} \leftarrow E^{c_H} \cdot h^{\widehat{\rho}} \cdot \widetilde{h}^{\widehat{o}} \qquad\qquad \widehat{T_2} \leftarrow E^{\widehat{c}} \cdot h^{-\widehat{m}} \cdot \widetilde{h}^{-\widehat{t}} \tag{4.52}$$

$$\widehat{T_3} \leftarrow \left(\frac{e(h_0\mathcal{G}, \widehat{h})}{e(A, y)}\right)^{c_H} \cdot e(A, \widehat{h})^{\widehat{c}} \cdot e(\widetilde{h}, \widehat{h})^{\widehat{r}} \cdot e(\widetilde{h}, y)^{-\widehat{\rho}} \cdot e(\widetilde{h}, \widehat{h})^{-\widehat{m}} \cdot e(h_1, \widehat{h})^{-\widehat{m_2}} \cdot e(h_2, \widehat{h})^{-\widehat{s}} \tag{4.53}$$

$$\widehat{T_4} \leftarrow \left(\frac{e(\mathcal{G}, \text{acc})}{e(g, \mathcal{W})z}\right)^{c_H} \cdot e(\widetilde{h}, \text{acc})^{\widehat{r}} \cdot e(1/g, \widehat{h})^{\widehat{r'}} \quad \widehat{T_5} \leftarrow D^{c_H} \cdot g^{\widehat{r}}\widetilde{h}^{\widehat{o'}} \tag{4.54}$$

$$\widehat{T_6} \leftarrow D^{\widehat{r''}} \cdot g^{-\widehat{m'}}\widetilde{h}^{-\widehat{t'}} \qquad \widehat{T_7} \leftarrow \left(\frac{e(pk \cdot \mathcal{G}, \mathcal{S})}{e(g, g')}\right)^{c_H} \cdot e(pk \cdot \mathcal{G}, \widehat{h})^{\widehat{r''}} \cdot e(\widetilde{h}, \widehat{h})^{-\widehat{m}} \tag{4.55}$$

$$\widehat{T_8} \leftarrow \left(\frac{e(\mathcal{G}, u)}{e(g, \mathcal{U})}\right)^{c_H} \cdot e(\widetilde{h}, u)^{\widehat{r}} \cdot e(1/g, \widehat{h})^{\widehat{r'''}} \tag{4.56}$$

and adds these values to $\widehat{T}$.

4.3.13.2 Validity

Verifier uses all issuer public key pk_I involved into the credential generation and the received $(c, \widehat{e}, \widehat{v}, \{\widehat{m_j}\}, A')$. He also uses revealed $\{m_j\}_{j \in \mathcal{A}_r}$. He initiates $\widehat{\mathcal{T}}$ as empty set.

1. For each credential C_p, take each sub-proof Pr_C and compute

$$\widehat{T} \leftarrow \left(\frac{Z}{\left(\prod_{j \in \mathcal{A}_r} {R_j}^{m_j}\right)(A')^{2^{596}}}\right)^{-c} (A')^{\widehat{e}} \left(\prod_{j \in (\mathcal{A}_{\tilde{r}})} {R_j}^{\widehat{m_j}}\right) (S^{\widehat{v}}) \pmod{n}. \tag{4.57}$$

   Add $\widehat{T}$ to $\widehat{\mathcal{T}}$.

2. For each predicate p:

$$\Delta' \leftarrow \begin{cases} z_j; & \text{if } * \equiv \leq \\ z_j - 1; & \text{if } * \equiv < \\ z_j; & \text{if } * \equiv \geq \\ z_j + 1; & \text{if } * \equiv > \end{cases}$$

$$a \leftarrow \begin{cases} -1 & \text{if } * \equiv \leq \text{ or } < \\ 1 & \text{if } * \equiv \geq \text{ or } > \end{cases}$$

2.1. Using Pr_p and $\mathcal{C}$ compute

$$\{\widehat{T_i} \leftarrow T_i^{-c} Z^{\widehat{u_i}} S^{\widehat{r_i}} \pmod n\}_{1 \leq i \leq 4}; \tag{4.58}$$

$$\widehat{T_\Delta} \leftarrow \left(T_\Delta^a Z^{\Delta'}\right)^{-c} Z^{\widehat{m_j}} S^{\widehat{ar_\Delta}} \pmod n; \tag{4.59}$$

$$\widehat{Q} \leftarrow (T_\Delta^{-c}) \prod_{i=1}^{4} T_i^{\widehat{u_i}} (S^{\widehat{\alpha}}) \pmod n, \tag{4.60}$$

and add these values to $\widehat{\mathcal{T}}$ in the order $\widehat{T_1}, \widehat{T_2}, \widehat{T_3}, \widehat{T_4}, \widehat{T_\Delta}, \widehat{Q}$.

4.3.13.3 Verification

For the credential pair (C_p, C_{NR}), Verifier retrieves relevant variables from $\mathcal{X}, \{Pr_C\}, \{Pr_p\}, \mathcal{C}$.

4.3.13.4 Final hashing

1. Verifier computes
$$\widehat{c_H} \leftarrow H(\widehat{\mathcal{T}}, \mathcal{C}, n_1).$$

2. If $c = \widehat{c}$ output VERIFIED else FAIL.

4.3.14 Performance Analysis

Performance tests for implementation is performed respectively on an Ubuntu 16.04.7 LTS with Linux Kernel 4.15.0 machine running on VMware Workstation 16 Pro with host operating system Windows 10 running on Intel Xeon Silver 4116 Processor running at 2100 MHz. To obtain more consistent results, we reported the minimum, maximum and average time of the respective implementation over 100 and 1000 executions. The results for anonymous credential setup are shown in Table 4.2

Table 4.2: Anonymus Credential Test Implementation Speed

	master secret gen. time		issuer GVT credential schema gen time		issuer GVT credential definition gen time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	17.47µs	15.17µs	18.5µs	18.2µs	1.00557315s	1.057872s
MAX	50.49µs	117.27µs	107.04µs	167.42µs	947.2851s	870.46875s
AVERAGE	20.7583µs	19.62601µs	24.0495µs	23.7571µs	30.78273532s	15.80892103s
	issuer GVT revocation registry gen time		issuer GVT credential values gen time		prover hidden attribute blinding time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	337.23855ms	331.57356ms	41µs	40.78µs	73.42475ms	72.89835ms
MAX	408.36219ms	534.09137ms	329.33µs	215.88µs	103.44737ms	138.99228ms
AVERAGE	345.3957446ms	339.4934018ms	56.6513µs	51.84878µs	76.3689156ms	75.43458866ms
	issuer sign GVT credential values gen time		prover GVT witness creation time		prover GVT credential processing time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	125.47951ms	122.93533ms	49.2µs	48.63µs	469.30908ms	461.60831ms
MAX	322.58375ms	386.90856ms	156.59µs	179.2µs	585.23201ms	583.67048ms
AVERAGE	165.966149ms	163.1786705ms	63.4952µs	58.22852µs	482.4763579ms	470.9423873ms
	issuer XYZ credential schema gen time		issuer XYZ credential definition gen time		issuer XYZ revocation registry gen time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	20.67µs	19.08µs	1.58330112s	1.05539338s	448.60208ms	440.92805ms
MAX	145.05µs	126.54µs	804.4612s	989.43063s	561.5501ms	566.64639ms
AVERAGE	39.1393µs	29.14326µs	20.07649803s	22.50077315s	462.8122666ms	450.868284ms
	issuer XYZ credential values gen time		prover blind hidden attribute gen time		issuer XYZ credential values gen time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	34.15µs	34.48µs	57.37726ms	57.07111ms	124.69843ms	122.71689ms
MAX	127.3µs	218.98µs	72.73212ms	86.46947ms	367.32748ms	378.17915ms
AVERAGE	50.9141µs	48.39224µs	59.7042193ms	59.04681154ms	167.3723781ms	162.7702631ms
	prover XYZ witness create time		prover XYZ credential signature processing time		subproof request creation time related to GVT credential	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	1.09492ms	1.07825ms	469.26361ms	462.4839ms	26.59µs	1.16507µs
MAX	3.67564ms	4.6593ms	554.53307ms	591.3504ms	124.39µs	138.52µs
AVERAGE	1.2317891ms	1.18287572ms	482.1433752ms	471.0339915ms	45.0879µs	38.63973507µs
	subproof request creation time related to XYZ credential		prover proof creation fime for two subproof creation request		verifier verify proof gen time	
	n_{100}	n_{1000}	n_{100}	n_{1000}	n_{100}	n_{1000}
MIN	3.91µs	4µs	3.55959207s	3.49669162s	4.45898388s	4.3867859s
MAX	20.85µs	68.05µs	4.14176742s	4.40915196s	5.06590458s	5.00973582s
AVERAGE	5.0608µs	5.70737µs	3.637155797s	3.563194193s	4.559505885s	4.46603348s

As can be seen in the Table 4.2 highlighted with red colour, it takes seconds for the protocol validators to complete the verification period for anonymous identity information. It is not surprising to us that this part is one of the slowest parts of the protocol. Because the study in [27] confirms that the CKS scheme is slow in proving a valid credential for the verifier to check the user's revocation status with the help of zero-knowledge proof. On the other hand, before each zero-knowledge proof call, an update witness protocol must be found as described in the CKS article [9]. As a result, the effectiveness of witness updates is important. Looking at the results obtained from the article in [27], it is seen at the left bottom corner of the Table 4.2 with green highlight that the witness update performance of the CKS schema is good. However, with regard to witness update, in small-scale environments with limited number of revocations or additions, the effectiveness of the user's revocation status where the time spend for it is shown with green colour on the right side of the Table 4.2, with the help of the zero-knowledge proof protocol may be more important than the efficiency of the update witness protocol [27]. Therefore, when you decide to use the library for anonymous credential revocation, you should consider the scale of your business which schema best suits your situation.

CHAPTER 5

CONCLUSION

In this thesis, Aries, Indy projects, which are part of the Hyperledger umbrella project, and the Ursa cryptographic library, which is the focus of our thesis, where the cryptographic tools of this project are brought together, are examined. To elaborate further, firstly, we briefly talked about why we studied the Aries and Indy projects, examined the theoretical details of these projects, explained how the ACA-Py framework works and how you can run the framework, and then we presented a working example of it. Secondly, we explained the general operation and design intent of the CurveZMQ protocol. Finally, we explained why Ursa lib was created, what the concept of anonymous identity means, and gave the performance test results of both anonymous identity creation/revocation and cryptographic primitives such as digital signatures and key exchange algorithm used in the libursa library. With the performance values we showed in this study, it has been shown that processes such as revocation verification, proof verification and credential definition parts which took more time, needs to be studied in depth to increase overall performance of the library.

As a future work, some algorithms can be replaced and their effectiveness may be investigated. For example, anonymous credential systems was using CL signature of which it's security is based on factoring of two large prime numbers. To achieve sufficient security, CL signature-based Anoncred systems require long keys and signatures, resulting in slow cryptographic operations. For that reason, the BBS+ signature proposed which compared to CL signatures has much shorter keys and signatures for a comparable level of security but it's performance on anoncreds yet unknown although it is documented extensively on Anonymous Credential 2.0 paper [28]. Also, there are ongoing discussions for the application of post quantum algorithms to be used in

blockchain projects. Implementation of cryptographic primitives such as signature, key exchange/basic encryption with Post-quantum algorithms and the effects of these studies on system security and performance remain an important field of study.

REFERENCES

[1] Hyperledger indy, https://hyperledger-indy.readthedocs.io/en/latest/index.html.

[2] Hyperledger indy-plenum, https://hyperledger-indy.readthedocs.io/projects/plenum/en/latest/index.html.

[3] An introduction to hyperledger [white paper], https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf.

[4] Libursa, Github repository, https://github.com/hyperledger/ursa/tree/main/libursa, December 2018.

[5] Ursa library motivation, https://wiki.hyperledger.org/display/ursa/Ursa+Library+Motivation, February 2019.

[6] Ursa for tokyo meetup presentation, Jira Software, https://wiki.hyperledger.org/display/ursa/Presentations, July 2019.

[7] Von network, Github repository, https://https://github.com/bcgov/von-network, November 2017.

[8] D. J. Bernstein, High-speed high-security cryptography: encrypting and authenticating the whole internet, 27th Chaos Communication Congress, https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4295.en.html, December 28 2010.

[9] J. Camenisch, M. Kohlweiss, and C. Soriente, *An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials*, Springer, Berlin, Heidelberg, 2009.

[10] J. Camenisch and A. Lysyanskaya, A signature scheme with efficient protocols, Security in Communication Networks, Third International Conference, 2567 of Lecture Notes in Computer Science, pp. 268–289, 2002.

[11] L. Foundation, Hyperledger, https://www.hyperledger.org.

[12] S. Foundation, Self-sovirin identity, https://sovrin.org/, 2021.

[13] S. Foundation, Sovrin: A protocol and token for self-sovereign identity and decentralized trust, https://sovrin.org/wp-content/uploads/

2018/03/Sovrin-Protocol-and-Token-White-Paper.pdf, January 2018.

[14] L. Harchandani, S. Khoroshavin, Toktar, Ashcherbakov, A. Nikitin, A. Obruchnikov, V. Muzychenko, and A. Kononykhin, Hyperledger indy-plenum, GitHub repository, https://github.com/hyperledger/indy-plenum, 2016.

[15] D. Hardman, Aries rfc 0003: Protocols, https://github.com/hyperledger/aries-rfcs/blob/master/concepts/0003-protocols/README.md, 2019.

[16] D. Hardman, Aries rfc 0004: Agents, https://github.com/hyperledger/aries-rfcs/blob/master/concepts/0004-agents/README.md, 2019.

[17] D. Hardman, Aries rfc 0005: Did communication, https://github.com/hyperledger/aries-rfcs/blob/master/concepts/0005-didcomm/README.md, 2019.

[18] D. Hardman, Aries rfc 0051: Decentralized key management, https://github.com/hyperledger/aries-rfcs/blob/master/concepts/0051-dkms/README.md, 2019.

[19] D. Hardman, R. Jones, S. Curran, S. Curran, T. Ronda, R. Esplin, and D. Bluhm, Hyperledger aries, GitHub repository, https://github.com/hyperledger/aries, May 2019.

[20] D. Huseby, Hyperledger ursa, Github repository, https://www.hyperledger.org/use/ursa, November 2018.

[21] iMatrix Corporation, Curvezmq - security for zeromq, http://curvezmq.org/page:read-the-docs, [Accessed: 19.02.2021], 2013.

[22] iMatrix Corporation, Frequently asked questions - zeromq, http://wiki.zeromq.org/area:faq, [Accessed: 19.02.2021], 2013.

[23] D. Khovratovich and M. Lodder, Anonymous credentials with type-3 revocation, 19 June 2019, version 0.5.

[24] D. Khovratovich and M. Lodder, Issuance of credentials, Anonymous credentials with type-3 revocation, pp. 4–6, 19 June 2019, version 0.5.

[25] D. Khovratovich and M. Lodder, Presentation, Anonymous credentials with type-3 revocation, pp. 6–7, 19 June 2019, version 0.5.

[26] D. Khovratovich and M. Lodder, Schema preparation, credentials with type-3 revocation, pp. 2–3, 19 June 2019, version 0.5.

[27] J. Lapon, M. Kohlweiss, B. De Decker, and V. Naessens, *Performance Analysis of Accumulator-Based Revocation Mechanisms*, Springer, Berlin, Heidelberg, 2010.

[28] M. Lodder, B. Zundel, and D. Khovratovich, Pairings-based anonymous credentials with circuit-based revocation and permission policies, 17 June 2019, version 0.7.

[29] D. Reed, M. Sporny, D. Longley, C. Allen, R. Grant, and M. Sabadello, Decentralized identifiers (dids) v1.0, `https://www.w3.org/TR/2021/WD-did-core-20210103/#introduction`, November 2019.

[30] M. Sporny, D. Longley, and D. Chadwick, Verifiable credentials data model 1.0, `https://www.w3.org/TR/vc-data-model/#refreshing`, January 2021.

[31] R. Tosirisuk, Anonymous credential part2: Selective disclosure and cl signature, Medium repository, `https://medium.com/finema`, Feb 4, 2018.