TRABALHO DE GRADUAÇÃO

A MECHANISM FOR GUARANTEEING THE
AUTHENTICITY OF DIGITAL IDENTITY DOCUMENTS USING
DIGITAL SIGNATURES, DNSSEC AND BLOCKCHAIN

Luiz Fernando Ribeiro Amaral

Brasília, Julho de 2018

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

A MECHANISM FOR GUARANTEEING THE AUTHENTICITY OF DIGITAL IDENTITY DOCUMENTS USING DIGITAL SIGNATURES, DNSSEC AND BLOCKCHAIN

Luiz Fernando Ribeiro Amaral

Trabalho de Graduação submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Georges Daniel Amvame Nze, Ph.D, ENE/UnB _____
Orientador

Joseph E. Gersch, Ph.D, CS/Colorado State University _____
Co-orientador

Ugo Silva Dias, Ph.D, ENE/UnB _____
Examinador Interno

FICHA CATALOGRÁFICA

AMARAL, LUIZ FERNANDO RIBEIRO
A MECHANISM FOR GUARANTEEING THE AUTHENTICITY OF DIGITAL IDENTITY DOCUMENTS USING DIGITAL SIGNATURES, DNSSEC AND BLOCKCHAIN [Distrito Federal] 2018.
xvi, 56 p., 210 x 297 mm (ENE/FT/UnB, Engenheiro, Engenharia Elétrica, 2018).
Trabalho de Graduação - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. DNSSEC	2. Blockchain
3. RSA	4. Authenticity
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

AMARAL, L. F. R. (2018). *A MECHANISM FOR GUARANTEEING THE AUTHENTICITY OF DIGITAL IDENTITY DOCUMENTS USING DIGITAL SIGNATURES, DNSSEC AND BLOCKCHAIN*. Trabalho de Graduação, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 56 p.

CESSÃO DE DIREITOS

AUTOR: Luiz Fernando Ribeiro Amaral
TÍTULO: A MECHANISM FOR GUARANTEEING THE AUTHENTICITY OF DIGITAL IDENTITY DOCUMENTS USING DIGITAL SIGNATURES, DNSSEC AND BLOCKCHAIN.
GRAU: Engenheiro de Redes de Comunicação ANO: 2018

É concedida à Universidade de Brasília permissão para reproduzir cópias desto Trabalho de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Os autores reservam outros direitos de publicação e nenhuma parte desse Trabalho de Graduação pode ser reproduzida sem autorização por escrito dos autores.

Luiz Fernando Ribeiro Amaral
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATION

To my grandfather Antônio, who unfortunately passed away before I could share this accomplishment with him.

Luiz Fernando Ribeiro Amaral

ACKNOWLEDGEMENTS

I was blessed by having the help and support of many great people during this journey. Naming all of them and forgetting one would be unfair. To each of you who have in some way contributed, I say thank you.

Luiz Fernando Ribeiro Amaral

ABSTRACT

This project proposes a mechanism for guaranteeing the authenticity of digital identity documents using Digital Signatures, DNSSEC and Blockchain. Although there are papers proposing similar implementations, none of them address the public key distribution using the DNS system and do not provide a way of implementing trusted timestamping. The goal of this project is to propose a mechanism design with the aforementioned characteristics, implement a proof of concept and evaluate it using crafted attack scenarios. In the end, a few improvements that were not implemented were presented as future work.

RESUMO

Este projeto propõe um mecanismo para garantia de autenticidade de documentos de identidade digitais utilizando Assinaturas Digitais, DNSSEC e *Blockchain*. Apesar de existirem trabalhos propondo implementações similares, nenhuma delas aborda a distribuição das chaves públicas utilizando o sistema DNS e não fornecem uma maneira de implementar *timestamping* confiável. O objetivo desse projeto é propor um mecanismo com as características mencionadas anteriormente, implementar uma prova de conceito e avaliá-la utilizando cenários de ataque construídos. Ao final, algumas melhorias que não foram implementadas foram apresentadas como trabalhos futuros.

CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
BTC	*Bitcoin*
CA	*Certificate Authority*
DNS	*Domain Name System*
DNSSEC	*Domain Name System Security Extensions*
DS	*Delegation Signer*
DSA	*Digital Signature Algorithm*
ECC	*Elliptic-curve cryptography*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
Hazmat	*Hazardous Materials Layer*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
KSK	*Key Signing Key*
NIST	*National Institute of Standards and Technology*
PKI	*Public Key Infrastructure*
PNG	*Portable Network Graphics*
POW	*Proof of Work*
PSS	*Probabilistic Signature Scheme*
QR Code	*Quick Response Code*
REST	*Representational State Transfer*
RR	*Resource Record*
TLD	*top Level Domain*
TXID	*Transaction Identifier*
UTXO	*Unspent Transaction Output*
WSGI	*Web Server Gateway Interface*
XML	*Extensible Markup Language*
ZSK	*Zone Signing Key*

1 INTRODUCTION

1.1 MOTIVATION

As sophisticated printing and scanning technologies become cheap, criminals are taking advantage of digital technology to produce high quality fraudulent documents. To avoid fraud and the use of falsified documents, society relies on experts for verifying the authenticity of these documents using special tools and document properties, an almost impossible job for entities dealing with thousands of documents daily (Garain and Halder 2008).

To avoid falsification and fraud, governmental agencies have started issuing digital versions of paper documents that can be carried by the citizens in their smartphones in the form of a QR Code. Although this new technology brings convenience, it also brings new problems related with the security of the information being carried in the code. For example, has it been tampered with or is the entity identified as issuer really who it claims to be?

Previous work rarely propose ways of distributing the public key for the purpose of Authentication of Identity Documents making the rollover of new keys more difficult, besides relying on a third-party Certificate Authority for guaranteeing the certificates authenticity, bringing additional costs to the operation or the challenge of managing a full Public Key Infrastructure.

The Brazilian Federal Data Processing Service (SERPRO) launched in 2017 a service called Lince, aiming to provide a similar solution to the one described in this work except that instead of using digital signatures, their system encrypts the information using a private key. While it provides authentication it does not fully guarantees integrity, a problem that is addressed in this work by using digital signatures instead of pure encryption of the data.

1.2 THE SOLUTION

This work addresses these problems by providing a mechanism to guarantee authenticity and integrity of the aforementioned data through the use of digital signatures. The distribution of the public keys used to validate the signatures is done by publishing the certificates and corresponding public keys on the DNS infrastructure. This mechanism enables any person to verify the information contained in the QR Code using a smartphone application, instead of relying on specialists or the physical security features of a document.

When it comes to guaranteeing authenticity and integrity, digital signatures are a standard nowadays. *A digital signature algorithm allows an entity to authenticate the integrity of signed data and the identity of the signatory. The recipient of a signed message can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory* (FIPS 2013). In the proposed scheme, a message is digitally signed and a digital signature block is stored in a QR Code which can be printed on the identity document, such as for instance a driver's license containing a QR Code with the message and its digital signature as shown in Figure 1.1 or that can be digitally loaded in a smartphone to be presented in a digital form.

Figure 1.1: Driver's license example. Adapted from (DMV 2013)

Compared to other proposed solutions (Warasart and Kuacharoen 2012), this system does not need to rely on a certificate authority to provide valid certificates, reducing the costs associated with certificate issuance, neither needs to have the certificates pre-loaded by the application used for signature validation. Through the use of DNSSEC (Domain Name System Security Extensions) enabled servers, the document issuer can publish the certificates in their DNS zone and the authenticity and integrity of the certificate can be guaranteed by the authentication chain provided by DNSSEC signatures (Rose et al. 2005).

1.3 OBJECTIVE

This work's objective is to design a mechanism that guarantees the integrity of the data contained in the QR Code and authenticates the issuer using digital signatures with the public keys distributed on the DNS and a trusted timestamping service based on the Bitcoin Blockchain. To evaluate the effectiveness and security of the system, a proof of concept will be implemented and the analyzed data will be collected from it.

1.3.1 Specific objectives

- Design a mechanism that addresses the aforementioned problems;

- Implement a proof of concept of the mechanism;
- Using the proof of concept, collect results and evaluate the performance and security of the mechanism against various attack scenarios.

1.4 ORGANIZATION OF THIS WORK

On chapter 2, the architectural components used to build our solution will be explained, followed by a proposal of a solution design on chapter 3. Chapter 4 will demonstrate how a proof of concept was implemented using the design from chapter 3, and the results obtained from the proof of concept will be presented and analyzed in chapter 5. Finally, chapter 6 will talk about the conclusions obtained with this work and some aspects that can be further developed in the future.

2 ARCHITECTURAL COMPONENTS

2.1 DIGITAL SIGNATURES

Digital signatures are a standard for guaranteeing authenticity and integrity nowadays. *A digital signature algorithm allows an entity to authenticate the integrity of signed data and the identity of the signatory. The recipient of a signed message can use a digital signature as evidence in demonstrating to a third party that the signature was, in fact, generated by the claimed signatory* (FIPS 2013).

On the other hand, digital signatures do not have timestamps by default, leaving it up to the user to include a timestamp with it using an additional mechanism.

2.1.1 The RSA Algorithm

RSA is a widely used public key cryptography algorithm. Invented in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman, the RSA provides encryption and digital signature capabilities, and its cryptosystem is based on the idea that factoring is a computationally intensive task (Jansma and Arrendondo 2004).

2.2 DOMAIN NAME SYSTEM - DNS

The Domain Name System (DNS) is standardized by a suite of Internet Engineering Task Force (IETF) Requests for Comments (RFC). The initial idea with its creation was to provide a translation between names and IP addresses, as a name is easier to remember than an IP address.

In order to make the DNS system robust and capable of load balancing, it is a distributed hierarchic system, divided in root name servers, top-level domains (TLDs) and authoritative name servers (Kurose and Ross 2010), as shown in Figure 2.1.

The DNS system stores information in standardized data structures called Resource Records (RRs). The RRs have predefined fields: owner name, type, class, time to live, length (RDLENGTH) and data (RDATA). The RDATA field contains the data stored in that record and its format is defined depending on the type and class of the record, even enabling advanced users to create their own custom records (Mockapetris 1987).

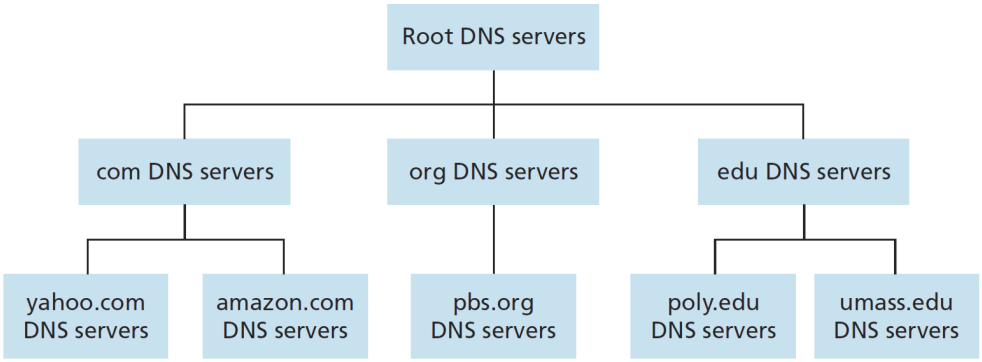


Figure 2.1: Portion of the DNS Server hierarchy. [Source: (Kurose and Ross 2010)]

2.2.1 DNSSEC

The DNS Security Extensions (DNSSEC) is a group of specifications for securing the DNS system, provided by the Internet Engineering Task Force (IETF). The DNSSEC provides origin authentication, authenticated denial of existence and data integrity, characteristics that were not present in the original DNS design. All that is added while still maintaining backwards compatibility (Infoblox 2018).

2.2.2 CERT Resource Record

The CERT Resource Record (RR) was created for storing certificates in the DNS. Public keys are usually published in the form of a certificate and by using the DNS, a public key can be made available with little to no human intervention.

When a certificate is retrieved from a secure DNS zone (DNSSEC enabled and valid chain), the key in the retrieved certificate may be trusted without having to verify the certificate chain. Same way, the non-existence of a CERT RR within the zone can only be asserted through the use of DNSSEC and NSEC/NSEC3 records (Josefsson 2006).

2.2.3 BIND DNS Server

BIND is the most used DNS software in the internet. It is open source and the software includes a domain name resolver, a domain name authority server and many additional tools (dig, dnssec-signzone, etc.), all fully compliant with the published DNS standards. By using the BIND authority server software, one can provide DNS services on the Internet for the domain names it is authoritative for (Consortium 2018).

2.3 BITCOIN

Bitcoin is the world's first completely decentralized digital currency (Brito and Castillo 2013). As of June 2018, the total market capitalization is estimated at more than $115 billion (CoinMarketCap 2018), the biggest market capitalization among all the digital currencies.

One of the main characteristics of Bitcoin (BTC), is the decentralization, meaning that there is no central institution responsible for it, enabling users to send payments in a peer-to-peer manner, without having to rely on a third-party institution. By using digital signatures and a proof-of-work chain, Bitcoin solved the double-spending problem without the need of a third-party, achieving decentralization (Nakamoto 2008).

The blockchain can be considered a form of Distributed Ledger Technology, acting as a distributed database and enabling users to store arbitrary data within it, even tho this is not the main purpose. This feature is what enables Bitcoin to be used as a distributed trusted timestamp platform, as will be explained later.

2.3.1 Transactions

In Bitcoin, transactions are what enable users to spend Satoshis(BTC/10^8). In simple words, *a transaction is a transfer of Bitcoin value that is broadcast to the network and collected into blocks*, and on top of that, they are public and unencrypted, enabling any user of the network to effectively audit and be aware of all transactions. Although the transactions are binary, there are websites, called block chain browsers, that provides a view of the transactions in a human-readable way (Wiki 2018).

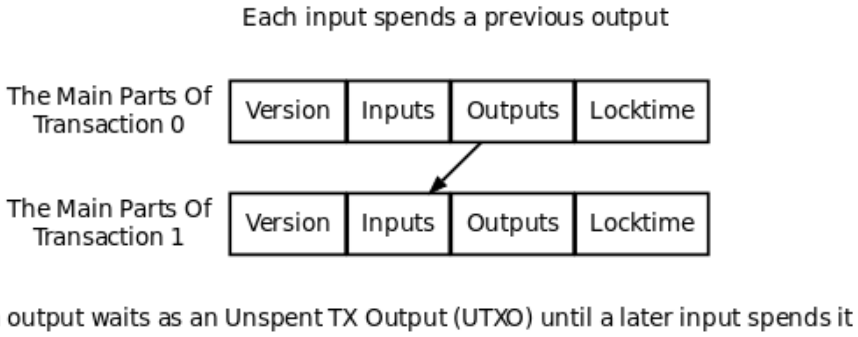


Figure 2.2: Main parts of a Bitcoin transaction. [Source: (Bitcoin Developer Guide 2018)]

The main parts of a transaction are the inputs and outputs, as can be seen in Figure 2.2. Each input is a reference to the output of a previous transaction, containing its transaction ID, the output index (vout) of that transaction and the ScriptSig, which contains a digital signature made with the private key and the corresponding public key. This digital signature is what proves to the network that someone is able to collect the input funds. An output contains the number of Satoshis being sent (how much the output is worth) as well as a ScriptPubKey, which is composed of the OP codes and the destination public key hash, which is effectively a Bitcoin wallet address (Wiki 2018). A transaction output that is not referenced by another transaction input, is called an Unspent Transaction Output (UTXO).

The outputs share the combined value of the inputs and it is important to note that any input value not included in the outputs is considered a transaction fee, which will be explained in Section 2.3.2. If Bob has an input transaction of 50 BTC and wants to send 25 BTC to Alice, he must include a second output that sends the remaining 25 BTC back to him, known as change, otherwise, the remaining value will be considered a transaction fee (Wiki 2018).

2.3.2 Blocks

The blocks are the structures that permanently record the transaction data. Each block in the network is linked to the previous block by including a 256-bit hash of the previous block header, creating what is called the Blockchain. The blocks also include a timestamp, denoting the approximate time when the block was created, and a list of transactions, with the first one on the list being a so called coinbase transaction, which enables the block miner to claim the block reward (Wiki 2018). A simplified example of the blockchain is shown in Figure 2.3

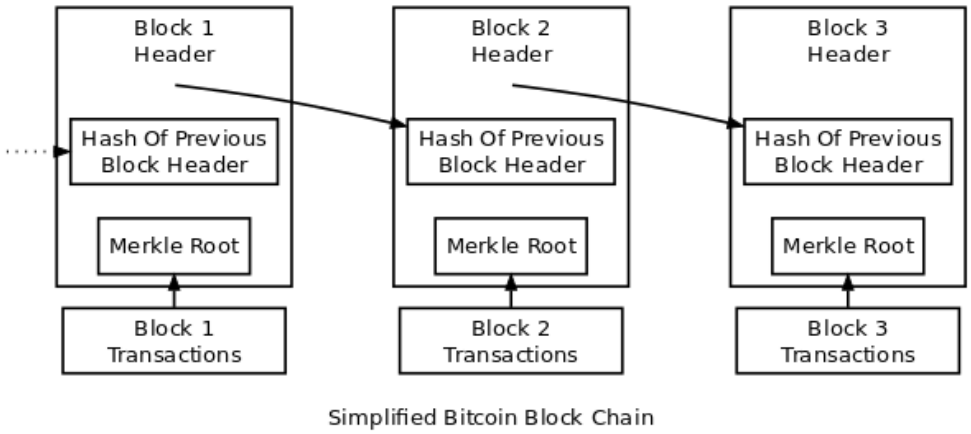


Figure 2.3: Simplified Bitcoin blockchain. [Source: (Bitcoin Developer Guide 2018)]

The blocks are mined using computational power and in order to incentivise people to provide computational power to the network, a predefined amount of Bitcoins is rewarded to the block finder, plus all the transaction fees. The Bitcoin protocol specified an initial value of 50 BTC as a reward, with a reduction of 50% every 210000 blocks, known as halving. Because of this characteristic, Bitcoin has a limited supply and one day will have a 0 BTC reward in mined blocks. When that happens, the reward for the miners will be comprised of only the transaction fees (Wiki 2018).

Blocks had an original limited size of 1MB (later expanded through the use of Segregated Witness), meaning that there is a limit of transactions that can be included in a block. Because the miners also get the transaction fees as a reward, transactions paying a bigger transaction fee will be included in a block faster than a transaction with a smaller or no fee, so when sending a transaction to the network, a user is effectively buying storage space in a block by paying a fee, which will be proportional to the size of the transaction.

2.3.3 Proof-of-Work

The proof-of-work (POW) is what gives the Bitcoin blockchain its append-only characteristic. The POW is a set of data that is costly to produce but easily verifiable by others. The POW production process is random, with a very low probability of success, meaning that on average, a lot of trial and error is necessary before a valid one is generated.

In Bitcoin, the POW is a double SHA-256 hash of the block header and to be valid, it must be smaller than the current network target. In order to generate different hashes for the same set of data, a nonce is included in the header and is incremented each time an invalid hash is calculated (Wiki 2018).

Because the POW requires a lot of computational effort, once a block is mined and appended to the blockchain, no modification can be done without having to redo all the work. The majority decision is made based on the amount of work, so it is determined by the longest chain, which has the biggest computational effort invested. That way in order to modify an existing block, an attacker would have to redo all the work, catch up with the work of the legit miners and surpass it in order for his modifications to be accepted by the network (Nakamoto 2008).

This append-only characteristic is what enables the use of the blockchain for a trusted timestamp, as *it is computationally infeasible to manipulate transaction records, meaning that timestamps are stored on a tamper-proof and persistently verifiable medium* (Breitinger and Gipp 2017).

2.3.4 Network

The Bitcoin network is a peer-to-peer network, comprised of nodes running a specific and agreed upon version of the Bitcoin protocol. Because of POW, the nodes in the network will always consider the longest chain as the valid one and will work on extending it. If two versions of an specific block are broadcast at the same time, a fork is created and the nodes work will be divided between the two branches. The first branch that becomes longer will take precedence and the smaller branch will be discarded (Nakamoto 2008).

When sending a transaction to the network, a node informs its peers of the new transaction. The peers will retrieve the full transaction data and if considered valid, they will inform their peers also, effectively broadcasting the transaction on the network. That way, a miner eventually receives the transaction and then include it in a block (Wiki 2018).

Every node in the network is capable of downloading the entire blockchain and the ones that do so are called full nodes. Full nodes have the entire blockchain downloaded and verified and thus are capable of searching the entire blockchain for an specific transaction or block (Bitcoin Developer Guide 2018).

2.4 QUICK RESPONSE CODE

A Quick Response Code (QR Code [1]) is one of the many kinds of 2D codes. It was invented in 1994 to be used in automotive industry and became an ISO standard in 2000 (WAVE 2018). QR Codes have become widespread nowadays, being used in patient identification in hospitals, bus tickets, betting tickets, electronic components labeling, aircraft and space industrial data-product identification and many other applications in all kinds of fields (Soon 2008).

The main advantages of the QR Code over other types of bar codes are the higher data-density, no predefined data structure as a requirement, meaning that it is able to store a higher amount of arbitrary data in the same area, ability to be read from any direction, resistance to distortion, data restoration, where the use of Reed-Solomon codes provide resistance to smudge or damage, and linking, making it possible for a QR Code to be split into up to 16 smaller QR Codes, enabling it to be printed even if the space is restricted, as shown in Figure 2.4 (Soon 2008).

The QR Code specification provides four types with different capacities and efficiencies, as shown on Table 2.1, at the cost of restricting the type of data that can be encoded (only numbers, only numbers and letters), meaning that when choosing an specific type of QR Code to be used, the type of data and the efficiency gain of the chosen method need to be taken into consideration, as the overhead of encoding binary data into alphanumeric might be bigger than the efficiency gain between the modes.

[1] QR Code is a registered trademark of DENSO WAVE INCORPORATED.

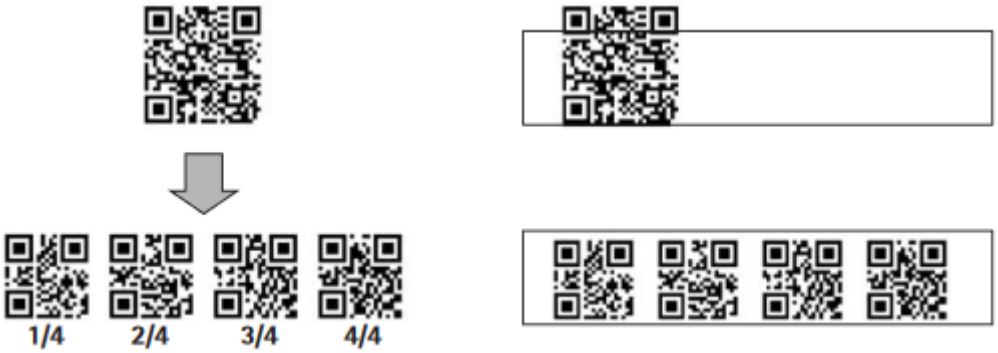


Figure 2.4: QR Code linking example. On the left, a QR Code splitted in 4 linked codes is shown. On the right, the use of linking to overcome a space restriction is demonstrated [Source: (Soon 2008)]

Table 2.1: QR Code modes characteristics. [Adapted from (Soon 2008)]

Mode	Capacity	Conversion Efficiency
Numerical characters	7,089 characters at maximum	3.3 cells/character
Alphanumerical	4,296 characters at maximum	5.5 cells/character
Binary (8 bit)	2,953 characters at maximum	8 cells/character
Kanji characters	1,817 characters at maximum	13 cells/character

QR Codes also have error correction capability, enabling them to be read even when part of it is missing or damaged. There are four error correction levels, chosen according to the operating environment, as listed in Table 2.2. The error correction capacity is measured in an approximate percentage of codewords that can be restored.

Table 2.2: QR Code error correction levels. [Adapted from (WAVE 2018)]

Level	Correction Capacity (approx.)
Level L	7% of codewords
Level M	15% of codewords
Level Q	25% of codewords
Level H	30% of codewords

2.5 PYTHON

Python is a powerful programming language capable of providing a simple but effective approach to object-oriented programming. Because of its interpreted nature, elegant syntax and dynamic typing, Python is widely used for rapid application development and scripting (Python 2018).

Python has a great amount of libraries and packages available, many developed by the open-source community. Because the interpreter can be easily extended with new functions implemented in C or C++, complex C libraries can be wrapped to be used by Python applications in a simpler manner.

2.5.1 Cryptography Package

Cryptography is a Python package that provides interfaces for cryptographic algorithms, including symmetric cyphers, message digests and key derivation functions. The package is divided in two levels: the recipes layer and the hazardous materials layer. The first, provides safe cryptographic recipes requiring little choices by a developer, while the later, provides low-level cryptographic primitives that required in-depth knowledge of cryptographic concept in order to be used safely (Python Cryptographic Authority 2013).

2.5.2 Tornado Framework

Tornado is a Python web framework and asynchronous network library (Tornado 2018) that unlike the other frameworks is not based on WSGI. The web framework provides its own interfaces and HTTP server, and through the use of non-blocking network I/O, has the capability of scaling to thousands of open connections.

The libraries provide many different components, including synchronous and asynchronous client and server-side implementations of HTTP and a co-routine library that makes the task of writing asynchronous code easier by avoiding callbacks among other components.

2.6 TELEGRAM

Telegram is an instant message platform that enables users to communicate securely by providing a service with end-to-end encryption. Besides the communication between users, the system also offers two APIs: The Telegram API, enabling users to develop customized Telegram clients and the Bot API, that provides a way for connecting bots to the system (Telegram 2018).

2.6.1 Telegram Bot

The official Telegram clients and the Bots API offer ways to enhance the user experience, by adding customized buttons to the screen and providing a command listing in the chat window, making it a very useful tool for building simple user interfaces.

Using the Telegram Bot API a code running on a third-party machine can be connected to the Telegram network using the HTTP protocol instead of using the complex Telegram protocol, enabling developers to provide customized applications to the Telegram users (Telegram 2018).

3 SOLUTION DESIGN

This chapter presents how each part of the system was implemented and the way they interact. The procedures are described in a way that makes them repeatable, enabling similar systems to be implemented.

First, a general explanation of the architecture will be given and later, the two main parts of the system will be explained separately in two sections.

3.1 SYSTEM ARCHITECTURE

The architecture is composed of signing entities, which can be governmental institutions or companies who issue identification documents, a phone application used for the signature validation and the DNS infrastructure. Figure 3.1 shows a diagram of the system components with a single signing entity for simplicity. Each entity can have a signing server, responsible for message signature and the QR Code generation and the DNS servers to provide the certificates for signature validation. A user, using his smartphone scans the QR Code using the camera, the phone decodes the data and queries the DNS server to retrieve the certificate and validate the DNSSEC authentication chain. It then validates the signature using the retrieved certificate and if valid, presents the data to the user.

It is important to note that the message signing and the signature verification parts are loosely coupled, meaning that one does not depend on the way the other is implemented, that way any DNS server software can be used, as long as it presents the data in the formats specified.

The software part responsible for generating signatures and serving the certificates would usually be under full control of the company implementing the system, while the signature verification part, would not be under control of the company, for example, a user verifying the data in the QR-Code using a software running on his phone. The main advantage of this implementation is that no updates are needed to the verifying software if the company rolls out new signing keys.

3.2 MESSAGE SIGNING

This section describes how the message signing process works on the proposed mechanism, including each of the required resources and operations and guidelines for their implementation.

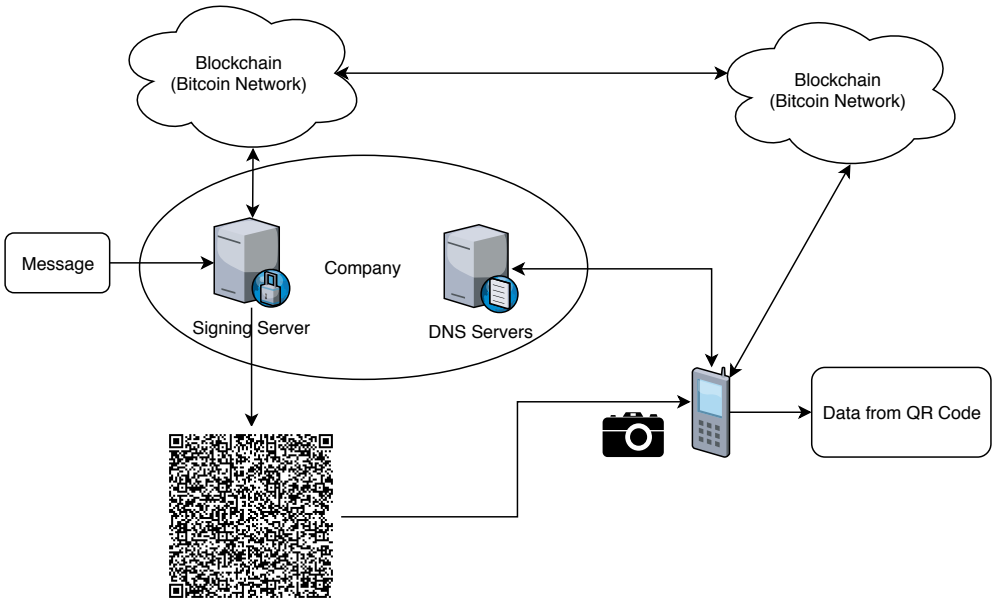


Figure 3.1: System Overview

3.2.1 Keys and Certificates

To assure the integrity of the message, the system relies on digital signatures generated using asymmetric cryptography. Any of the available digital signature algorithms can be used, as long as it is compatible with X.509 certificates containing the public key. To name a few examples, Elliptic Curve Cryptography (ECC) (Polk et al. 2009), Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) (Brown et al. 2010) and RSA (Housley, Schaad and Kaliski 2005).

The system supports certificates issued by a Certificate Authority (CA) or self-signed certificates. As the proposed mechanism publishes the certificates on the DNS, the system relies on the use of DNSSEC to guarantee the trustworthiness of the certificates published, meaning that a CA chain of trust is not required for a certificate to be trusted.

In fact, the certificate issuer can be ignored, as the system considers that a certificate published in a DNSSEC signed zone with a valid DNSSEC authentication chain is trustworthy.

3.2.2 Message Format

The main aspect of the design is that the data in the document, called message from now on, can be digitally signed to have its integrity and authenticity guaranteed. The main goal was to provide a flexible format that is machine and human readable at the same time, and still compact.

Two data formats were initially considered: Google Protocol Buffers (Buffers 2011) and JavaScript Object Notation (JSON).

Google states that *"protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data"*, similar to *"XML, but smaller, faster, and simpler"* (Buffers 2011). The problem with the protocol buffers is that changes to the data format might implies in a need to recompile application code. On top of that, protocol buffers are not human-readable, making it a bad fit for the design requirements.

On the other hand, JSON is human-readable data-interchange format that is easily parsed and generated by machines (Bray 2014). As JSON uses universal data structures, a message can be generated with one programming language and parsed with another, contributing to the uncoupling between the signing part and verifying part and if parsed in the right way, does not need to have all the fields previously specified, making it flexible.

Because of the facts exposed above, JSON was the chosen message format for the proposed mechanism.

The proposed format for the message is very flexible. It can contain a virtually unlimited amount of fields, with their names chosen by the entity implementing it. An example of how a message would look like can be seen in Figure 3.2.


```
{
    "_id": "1",
    "_root": "acme",
    "Name": "Motorist M. Michael",
    "DOB": "08/31/1978",
    "ID": "123 456 789",
    "Class": "D",
    "Issued": "08/31/2013",
    "Expires": "08/31/2021"
}
```


Figure 3.2: Example Message

The message object needs two mandatory values: *_root* and *_id*, representing the DNS root and the key identifier, respectively. These fields indicate where to find the certificate needed for signature validation meaning that the validating application still need a way to map the value in the *_root* to the corresponding DNS zone, to fetch the certificate, enabling the signing entity to roll out new keys without the need of updating the validating application to include the new keys.

The rest of the message fields should be named as they would appear for a user validating a message signature. Other hidden fields to be used by the validating application can be added by prepending an underscore (_) character to their names.

3.2.3 Trusted Timestamping

One of the problems found during the initial phase of this project, was the lack of a secure and efficient way to provide a timestamp to the signed message. It is important to have a timestamp because otherwise the signature will always be considered invalid if verified after the certificate expired, even if the signature was done before the expiration date.

RFC 3161 specifies a service (through a Time Stamping Authority) that proves that a certain data existed at a particular time, through the use of timestamp tokens (Zuccherato et al. 2001). The problem with this approach, is that the generated timestamp token is lengthy, thus making it hard to fit in a small QR Code, and has a complex to be obtained. Another point, is that the TSA system is centralized, making the the integrity of the timestamping process bound to the integrity of the TSA (Gipp, Meuschke and Gernandt 2015).

In order to overcome this obstacle, a system capable of providing a trusted timestamp with the minimal length possible was needed.

In (Clark and Essex 2012), the authors talk about a method to use cryptocurrencies as a decentralized trusted timestamping service. Based off of that work, (Gipp, Meuschke and Gernandt 2015) shows a method specific for the Bitcoin network, that generates an aggregated hash and can fit multiple message hashes into a single Bitcoin transaction.

For the sake of simplicity, the method used in this solution is a simpler version of the one mentioned above.

Before a new message signature is generated, the SHA-256 hash of the message is calculated and included in a transaction output containing an OP_RETURN opcode. After the transaction is created and broadcast to the network, the Transaction ID (TXID) is obtained and can be used to retrieve the transaction from the network, thus allowing the retrieval of the information about the block that contains the transaction, including its timestamp.

To be able to separate the TXID and message properly, a 0x1E ASCII character is inserted between them. This is a non-printable character, making it very easy to distinguish from the contents of the message or TXID, as they are both comprised of ASCII printable characters. The resulting set of data after this step, consists of message + 0x1E + TXID.

This set of data is now ready to be digitally signed, as described in Section 3.2.4.

3.2.4 Message Signature

After the timestamping process, the resulting data set (message + 0x1E + TXID) needs to be digitally signed. The private key used for this signature, is the one generated in the process explained in Section 3.2.1.

The signature algorithm needs to be the one provided by the type of certificate chosen before, adopting the best practices described in the standards, for example, the use of a Mask Generation Function and Padding for the RSA algorithm.

The signature is generated by taking the message + 0x1E + TXID as input data. The resulting binary signature is then Base64 encoded and appended to the input data using the 0x1F character as a separator.

The reason for encoding the signature in Base64 is to make sure that the signature only contains printable ASCII characters, enabling us to separate the data using the non-printable ASCII characters 0x1E and 0x1F. If we did not apply the Base64 encoding, the signature could contain a 0x1E or 0x1F byte and make it impossible to split the data correctly.

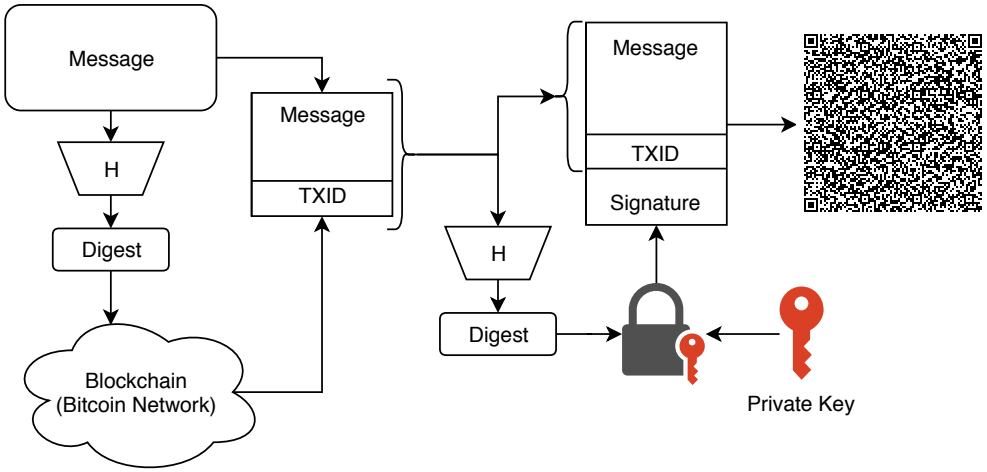


Figure 3.3: Message signing process.

The resulting data from the process shown in Figure 3.3 is now ready to be encoded in a QR Code.

3.2.5 QR-Code Generation

To enable the data to be machine readable, the final result (message + 0x1E + TXID + 0x1F + signature) is included in a QR-Code. The resulting size of the final data, is dependant on the size of the message and the digital signature algorithm, as the size of the Bitcoin TXID is fixed.

For example, when using RSA with a 2048-bit key, the produced signature is 256 bytes long and after the Base64 encoding, which has a 4:3 ratio (Josefsson 2006), will result in a 344 characters string. We then include the Bitcoin TXID, which has 64 characters and sums up to 408 characters of data, leaving enough space for up to 2545 bytes for the message, if we consider each character having 1 byte.

The QR Code mode chosen for storage, was the binary mode. The reason for choosing it over the alphanumerical mode is because as explained in Chapter 2, the alphanumerical mode has a gain in data capacity of approximately 45% in relation to binary mode, but it only supports part of the ASCII character table, which does not include the brackets and colon used in the JSON message object. One possible solution for this problem, is to encode all the data using Base32, which meets the character requirements of the alphanumerical mode but has an overhead of about 60% (Josefsson 2006), making the final liquid gain around -15%.

Table 3.1: QR Code modes characteristics. [Adapted from (Soon 2008)]

Mode	Capacity
Alphanumerical	4,296 characters at maximum
Binary (8 bit)	2,953 characters at maximum

Table 3.2: Encoding characteristics.

Encoding	Ratio	Overhead
Base 32	8:3	62.5%
Base 64	4:3	25%

It is worth mentioning that the availability of the total theoretical capacity depends strongly on the physical size of the QR-Code. For example, if we end up with a QR-Code that is the size of a Letter page, it will be impossible to print it in a driver's license.

3.3 SIGNATURE VERIFICATION

This section describes how the signature verification process works on the proposed mechanism, including each of the required resources and operations and guidelines for their implementation.

3.3.1 DNS

The main advantage of the system is being able to roll out new keys for signature without the need of updating the verifying application. To achieve this characteristic, the public keys used for signature validation are made publicly available using the DNS system. This has some special requirements, like the use of DNSSEC and the preparation of the certificates to be published in a specific format.

An entity wishing to implement this system could use its existing DNS infrastructure, including the zone, as long as it meets the listed requirements, but a separate child zone helps organize the information and helps with security by using different keys from the main parent zone.

3.3.1.1 DNSSEC

In order to serve the zone containing the certificates in a secure manner, DNSSEC must be enabled and correctly setup. That means generating the DNSSEC Zone Signing Key (ZSK) and Key Signing Key (KSK), including the DS records in the parent zone to establish the trust relationship and include the public keys in the zone file, in the form of DNSKEY resource records, matching the DS records in the parent zone.

After generating the keys, the private keys are used to sign the zone. This operation generates the digital signatures for each set of Resource Records (RRs) and include them as RRSIG records, making the zone ready to be served in a secure manner.

3.3.1.2 Certificates on the DNS

Now that the data in the zone is served in a secure manner, the entity can proceed to include the RRs containing the certificates used for signature validation.

The best choice to make the certificates publicly available was through the use of the CERT RRs. Although DANE could be used to meet this requirement, the TLSA and SMIMEA RRs have specific purposes that differ from what is intended here. That way, they were dropped in favor of CERT, that has a more generic purpose.

(Josefsson 2006) discusses appropriate owner names for the certificates but only as a recommendation. In order to make the JSON message the smallest possible, a small integer number is used as an ID for the certificate. This same owner name is found in the *_id* field of the JSON message.

The use of small numbers in the *_id* field helps to keep a key history in an environment where an entity would roll out new keys every two days in order to minimize losses in case of key compromise (only two days worth of signatures are lost if a certificate gets revoked).

The creation of the CERT RR involves defining a certificate type, in this case type 1, PKIX, calculating the key tag as documented in (Rose et al. 2005) Appendix B, setting the algorithm type and finally encoding the certificate contents in Base64. An example entry is shown in Listing 3

After including all CERT RRs in the zone and generating the DNSSEC signatures, the zone is ready to be published.

3.3.2 Signature Validation

The signature verification process consists in standard steps. Decode the generated QR Code, query the DNS servers for the certificate, validate the signature, query the Bitcoin blockchain for the timestamp and finally check if the timestamp correspond to a day in between the certificate validity period.

That means that the verification system can be implemented in a variety of ways, be it a desktop application, a mobile phone app or even an online webpage.

An overview of the validation process can be seen on Figurue 3.4.

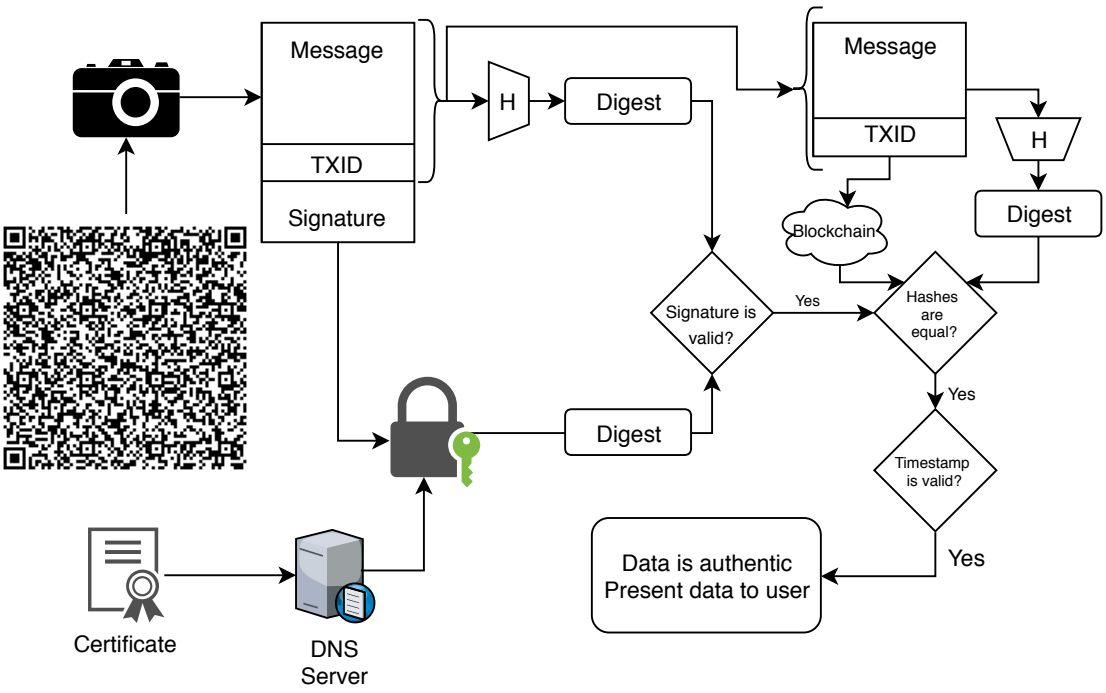


Figure 3.4: Signature Validation Process.

3.3.2.1 QR Code Decoding

The verification process starts by decoding the QR Code provided to the system and splitting the data using the 0x1E and 0x1F delimiters. With the data split, the system proceeds to retrieve the certificates from the DNS.

3.3.2.2 Fetching Certificates from the DNS

To fetch the certificates from the DNS, the system needs to know the owner name and zone associated with it. The owner name comes from the *_id* of the message, while the zone, is mapped to previously known values using the *_root* field, meaning that the only information that the validating party needs to know beforehand is how to map the value contained in the *_root* field to a DNS zone.

The mapped DNS zone is prepended with the information from the *_id* field and then used to query the DNS infrastructure for the certificate used.

The reason for using a dictionary mapping for the *_root* field instead of simply including the full domain straight away, is that an attacker could simply reproduce the same mechanism structure designed here and craft tampered messages that contains his domain in the *_root* field, essentially making the system recognize that information as valid, when it should not be.

During the process of querying for the certificate, the system must validate the DNSSEC authentication chain, either by validating it on each DNS level or by using a validating resolver.

If the DNSSEC authentication chain is deemed valid, the application proceeds to the next step, otherwise it ends the process and informs the user that the signature could not be verified. The whole DNSSEC authentication chain must be valid to guarantee that the returned certificates were not tampered with or forged by an attacker.

3.3.2.3 Validating the Signature

The signature validation process start by splitting the message + TXID from the signature by looking for the 0x1F character. The signature is then decoded from Base64 to binary format and verified against the message + TXID by using the procedure specified by the chosen algorithm.

Even if the signature is properly validated at this point, the application still needs to check if the signature was generated during the certificate validity period. If the signature cannot be validated, the user is informed and the process is terminated.

3.3.3 Timestamp Validation

To validate the timestamp, the application gets the message + TXID and split it using the 0x1E character, as explained in 3.2.4. The result of this operation is the message and the ID of the Bitcoin transaction (TXID) containing the message digest in the output that has the OP_RETURN opcode.

Using the TXID, the application queries the Bitcoin network for the transaction, with the most important data retrieved being the number of confirmations, outputs, timestamp and hash of the first block that included the transaction. If for any reason the transaction cannot be retrieved from the network or it does not have enough confirmations, the user is informed and the process is terminated.

The aforementioned block timestamp denotes the moment that the first block containing this transaction was broadcast to the network, proving that the data that has the SHA-256 hash stored through the OP_RETURN opcode existed at that time. Using that timestamp, the validating party can verify if the message was signed in between the certificate validity period (after the *notBefore* date and before the *notAfter* date) (Boeyen et al. 2008) and finally make a decision about the validity of the signature.

If the signature was generated outside of the certificate validity period, it is considered invalid and the user should be informed.

3.3.4 Presenting the Data to the User

The final step after completing the signature and timestamp validation is to present the data to the user. If the final decision about the signature considers it invalid, an error message should be presented to the user informing that the signature could not be validated and no data should be presented to him.

If all the validation steps are successful and the signature is considered valid, a success message is presented to the user, together with the data contained in the JSON message. Because of the way the message was designed, to present the data to the user the validating application can simply output the key-value pairs, using the key as the field name, followed by the value except for the fields that have keys starting with an underscore. These are the supporting fields, like *_root* and *_id* and do not have to be shown.

4 PROOF OF CONCEPT

4.1 OVERVIEW

This chapter will present how each part of the system is implemented in the proof of concept and explain how they interact with each other. The diagram in Figure 4.1 shows a general overview of the components and their interactions, which will be detailed in the following sections.

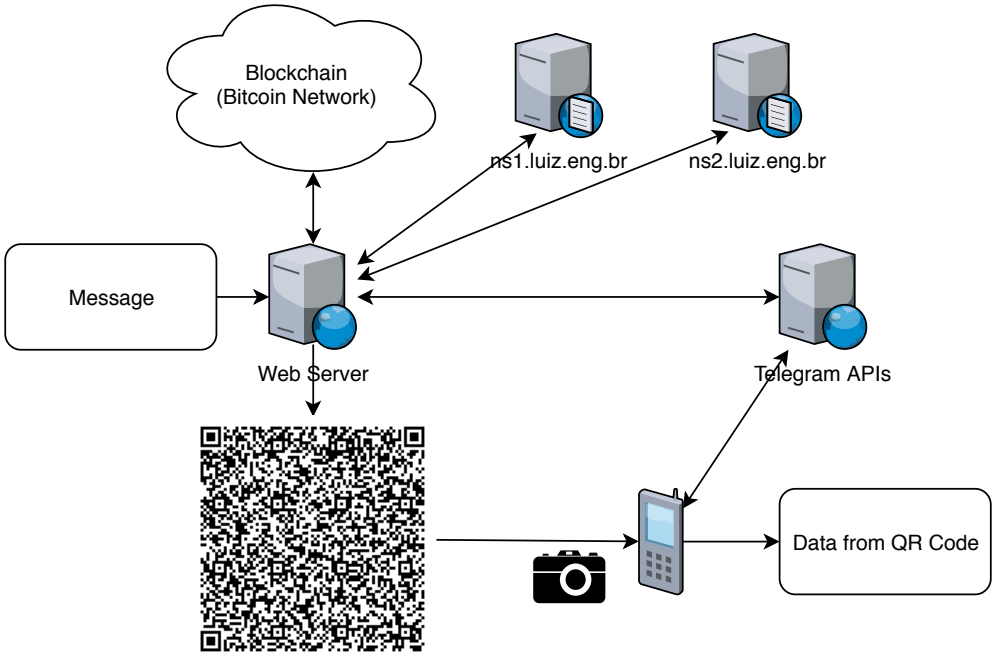


Figure 4.1: System components overview [Source: Author]

In order to validate the system features, two example companies will be used throughout the proof of concept: Acme and Globex.

4.2 DNS SYSTEM

The certificates containing the public keys used for signature validation are published using the DNS system. The details about how this proof of concept implements the DNS system are given in the following subsections.

4.2.1 DNS Servers

The proof of concept uses two DNS servers in a master-slave configuration, named ns1.luiz.eng.br and ns2.luiz.eng.br. The servers are two virtual machines running Ubuntu 16.04 and the open source DNS software BIND version 9.10.

For each example company a zone was created and published in both servers: acme.luiz.eng.br and globex.luiz.eng.br, having the certificates for each company published under the corresponding zone.

Although in this proof of concept both companies share the same DNS infrastructure, in a real world scenario each company would use its own DNS infrastructure.

4.2.2 DNSSEC

In order to serve the zones in a secure manner, DNSSEC was implemented in both zones using RSA keys. First a Key Signing Key (KSK) 4096 bits long was created and then a Zone Signing Key (ZSK) 2048 bits long, using the commands shown in Listing 4.1, where *$ZONE* was replaced with each of the zone names. The chosen algorithm for the keys was the RSASHA256, meaning that the signatures will use RSA with a SHA-256 hash function and support NSEC3. SHA-256 was chosen as it is recommended by the NIST in (FIPS 2015) and the use of NSEC3 prevents the ability to perfom DNSSEC zone walking, as recommended by NIST in (NIST 2013).

Listing 4.1: Commands to generate the DNSSEC keys [Source: Author]

```
1 dnssec-keygen -f KSK -a RSASHA256 -b 4096 -n ZONE $ZONE
2 dnssec-keygen -a RSASHA256 -b 2048 -n ZONE $ZONE
```

The execution of the above commands for key creation results in 4 files: 2 for the private keys, 2 for the public keys. The private keys are contained in files with the .private extension, while the public keys are contained in files with the .key extension.

The files containing the public keys, which are simply DNSKEY RRs matching the DS records in the parent zone, were included in the child zone through the use of the "$INCLUDE <file>" directive. Finally, the private keys were used to sign the zone, generating the digital signatures for each set of Resource Records (RRs) and include them as RRSIG records.

The signature process can be done automatically using the command shown in Listing 4.2. To make the process even easier when building the proof of concept, a more elaborated script that incremented the zone serial and generated the signatures was used. The script can be seen in Listing 2.

Listing 4.2: Command used to generate zone signatures [Source: (DigitalOcean 2016)]

```
1 /usr/local/sbin/dnssec-signzone -A -3 $(head -c 1000 /dev/urandom | sha1sum | cut -b 1-16) -N
      ↪ increment -o $ZONE -t $ZONEFILE
```

The *dnssec-signzone* command generates the DNSSEC signatures and the Delegation Signer (DS) records in a file called *dsset-<zonename>*. These records were added to the parent zone, in order to establish the DNSSEC authentication chain.

4.2.3 Publishing the Certificates

With the DNSSEC in place and the zone being served in a secure manner, the RRs containing the certificates used for signature validation can be published in the zone.

To make the certificates publicly available, they were published in the zones under CERT RRs. The owner names are formed using the specification described in Subsection 3.3.1.2.

In this proof of concept, the generated certificates were published as described in Section 4.3, following the guidelines previously mentioned.

The generation of the CERT RR involves obtaining the key tag, algorithm and certificate contents. To make the task easier, a Python script (1) was created to automate the process. The script is executed and receives the certificate file and owner name as arguments and returns a CERT RR in the correct format to be published in the BIND zone file.

When executed, the script loads the certificate file and serializes the public key in the PEM and DER format. It then calculates the key tag using the dnspython module, obtains the certificate algorithm identifier using the Hazmat layer functions (as specified in (IANA 2017)), strips off the certificate delimiters from the PEM encoded data and finally, it prints the resulting CERT RR.

An example containing one of the certificates published in the acme.luiz.eng.br zone can be seen in Listing 3.

4.3 KEYS AND CERTIFICATES GENERATION

As explained in Chapter 3, the system can support many digital signature algorithms. The chosen algorithm for the proof of concept was RSA, because of its widely available documentation.

The chosen private key size was 2048-bit the minimum required key size established by NIST (NIST 2015). All of the used certificates are self-signed and were created using OpenSSL. The use of self-signed certificates is possible if used in conjunction with DNSSEC. It guarantees the security and trustworthiness of these certificates, enabling the entity implementing it to replace the use of trusted CAs (Dukhovni and Hardaker 2015).

An example OpenSSL command used to generate some of the certificates is shown in Listing 4.3. This command will generate a self-signed certificate with a 2048-bit private key and the certificate will be valid for 730 days. The details for the certificate, like common name, state, etc. will be asked to be filled during the process.

Listing 4.3: OpenSSL command to generate self-signed certificates [Source: Author]

```
1 openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 730
```

The above command will produce two files: cert.pem, containing the certificate and key.pem, containing the private key.

For the Acme company, one certificate with 1 year of validity was issued and published in the DNS zone under the record 1.acme.luiz.eng.br, another certificate with 5 days of validity was issued but not published, followed by a certificate yet to be valid published under 3.acme.luiz.eng.br and finally an expired certificate published under 4.acme.luiz.eng.br.

For the Globex company, a single certificate with 365 days of validity was issued and published in the DNS zone under the record 1.globex.luiz.eng.br.

The certificates were issued with big validity periods in order to make the evaluation of the system as presented in Chapter 5 easier. In a real world scenario, certificates with only a few days of validity should be used, so that in case of key revocation, the entity does not lose a year worth of signed documents, for example.

4.4 WEB APIS

To make it easier to sign information, generate QR codes and retrieve timestamps with a Transaction ID (TXID), two Web APIs were implemented under the address https://tcc.luiz.eng.br/api/. The APIs handle all the complicated operations on the background, exposing endpoints that provides a simpler interaction using HTTP GET and POST requests.

Many API frameworks written in different programming languages are available. The Tornado framework was the chosen one because it is easy to implement and the author had previous experience with it. On top of that, Python provides a nice package, called Cryptography, to implement all the cryptographic operations that were needed.

4.5 MESSAGE SIGNING

This section describes in detail all the components involved in the process of signing a message. Each subsection will describe the details behind the components implementations.

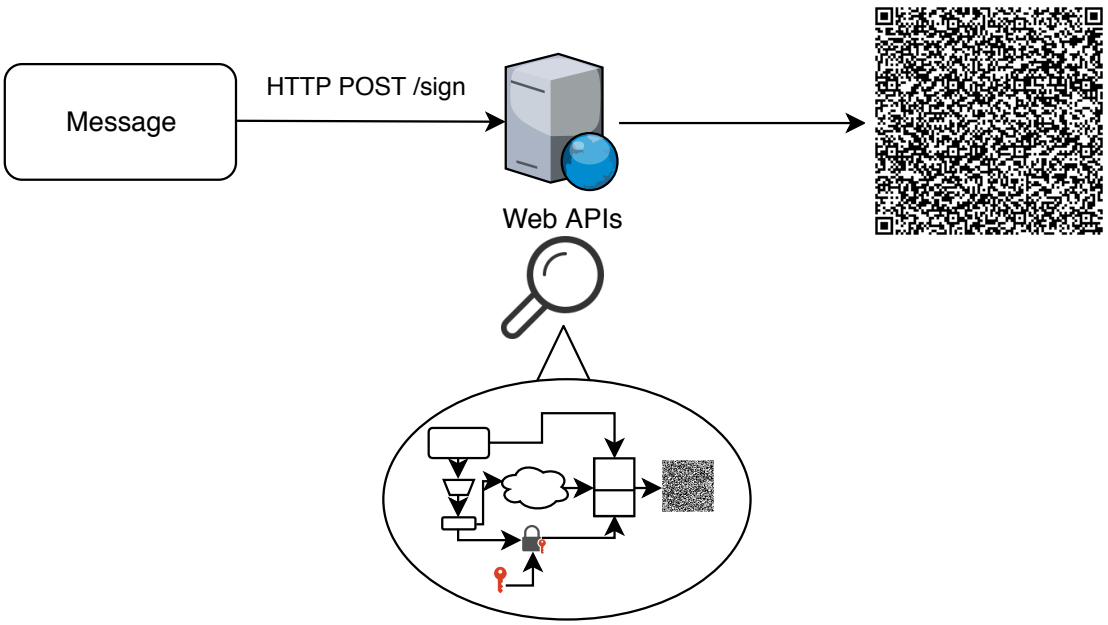


Figure 4.2: Message Signature Generation Process

Figure 4.2 shows the steps to generate a message signature. The diagram contained in the circle is the one shown in Figure 3.3.

4.5.1 Signing API

The Signing API has a single endpoint, called */sign*. This endpoint is responsible for receiving a message and returning a QR Code containing the message, transaction ID and digital signature.

When a message needs to be signed, an HTTP POST request is issued to the */sign* endpoint with the message complying with the format specified in 3.2.2 contained in the request body. When the server receives the request, it starts the signature process, performing all the required operations: checking the message for required fields, obtaining a timestamp, generating the signature and encoding the data in the QR Code. The resulting QR Code is returned to the client in the response body.

The operations performed by this API will be explained in details in the following subsections and a working example is available in the appendix under Section I.3.

4.5.2 Message Format

The implemented proof of concept uses the message format explained in Subsection 3.2.2. No changes were made to it.

4.5.3 Trusted Timestamping

When the Signing API receives a request containing a message to be signed, the first step is to calculate the SHA-256 hash of the received message. After, a Bitcoin transaction is created and an output containing the OP_RETURN opcode and the encoded hash is included. The transaction is then signed using the *signrawtransaction* Bitcoin JSON-RPC command and then sent to the network using the *sendrawtransaction* command.

The *sendrawtransaction* command sends the transaction to the network and returns the Transaction ID (TXID) that can be later used to retrieve the transaction from the network, allowing the retrieval of the information about the block that contains the transaction, including its timestamp.

To be able to separate the TXID and message properly when verifying the signature, a 0x1E ASCII character is inserted between them. This is a non-printable ASCII character, making it very easy to distinguish from the contents of the message or the TXID. The resulting set of data after this step, consists of message + 0x1E + TXID, as shown in Listing 4.4, and is ready to be signed.

Although the "+" signs are shown on the Listing 4.4, they are not included in the data. They are only used in the example to clearly show the separator used.

Listing 4.4: Data before signature

```
1 {"_id": 1, "_root": "acme", "Name": "Paul"} + 0x1E +
2 10eda5106d6c6ea3ba1514c5692070c6780193f8de0cd87d97fc5571fb0c966c
```

If for any reason the mentioned steps cannot be performed with success, the API will return an HTTP 400 error and a JSON message in the body containing the key *status* with the value *error* and the key *message* with the returned error message from the Bitcoin JSON-RPC interface as the value.

4.5.4 Digital Signature

The Python Cryptography module (Python Cryptographic Authority 2013) is used as a tool to deal with all the cryptographic operations, specially its Hazardous Materials Layer (Hazmat). The Hazmat layer of the module consists of functions specially designed to deal directly with cryptographic operations involving encryption and signatures. In this implementation, the RSA functions were used to import the private keys and generate the signatures.

When generating the signatures, the system uses MGF1 (Kaliski and Staddon 1998) as a Mask Generating Function, Probabilistic Signature Scheme (PSS) (Moriarty et al. 2016) as a padding algorithm and SHA-512 as a hash function. The reason for choosing these options is because they are the ones used in the module examples (Python Cryptographic Authority 2013) and also meet some of the recommendations established by NIST in the SP800-57 Part 3 Rev. 1 (NIST 2015).

With the data (message + 0x1E + TXID) ready to be signed, the API loads the appropriate private key, instantiates a signer object and generates the signature. The resulting signature is then encoded in Base64 and appended to the data using the 0x1F separator. This results in a set of data comprised of message + 0x1E + TXID + 0x1F + Base64 encoded signature.

4.5.5 QR Code generation

After receiving the final data (message + 0x1E + TXID + 0x1F + Encoded Signature) from the previous steps, the application generates the QR Code using the parameters specified in Section 3.2.5. The solution design, did not specify an error correction level. To have the maximum amount of space available for data and because there is no reason for the QR Codes in this proof of concept to be damaged or dirty, the level L was chosen.

To avoid rework, the QR Codes in this proof of concept were generated using the python-qrcode module (Lincolnloop 2018). This module provides a pure Python implementation based on the Pillow imaging library for Python (Pillow: the friendly PIL fork 2018).

In this proof of concept implementation a specific function, shown in Listing 4.5, was created, receiving the final data as an argument and returning a PNG image of the QR Code.

Listing 4.5: QR Code generation function

```
1  def generate_signature_qrcode(data):
2      outfile = BytesIO()
3      try:
4          qr = qrcode.QRCode(
5              version=None,
6              error_correction=qrcode.constants.ERROR_CORRECT_L,
7              box_size=2,
8              border=4,
9          )
10         qr.add_data(data)
11         qr.make(fit=True)
12         im = qr.make_image()
13         im.save(outfile, "PNG")
14     except Exception as e:
15         logging.warning('Error when generating the QR-Code')
16         logging.exception('%s', e)
17
18     return outfile.getvalue()
```

The resulting PNG image is then returned by the API in the body of the HTTP response.

4.6 SIGNATURE VERIFICATION

The signature verification process consists of simple steps. The system decodes the generated QR Code, splits the data, queries the DNS servers for the certificate, validates the RSA signature, queries the Bitcoin network for the timestamp and finally checks if the timestamp correspond to a day in between the certificate validity period.

A diagram showing the complete process can be seen in Figure 4.3.

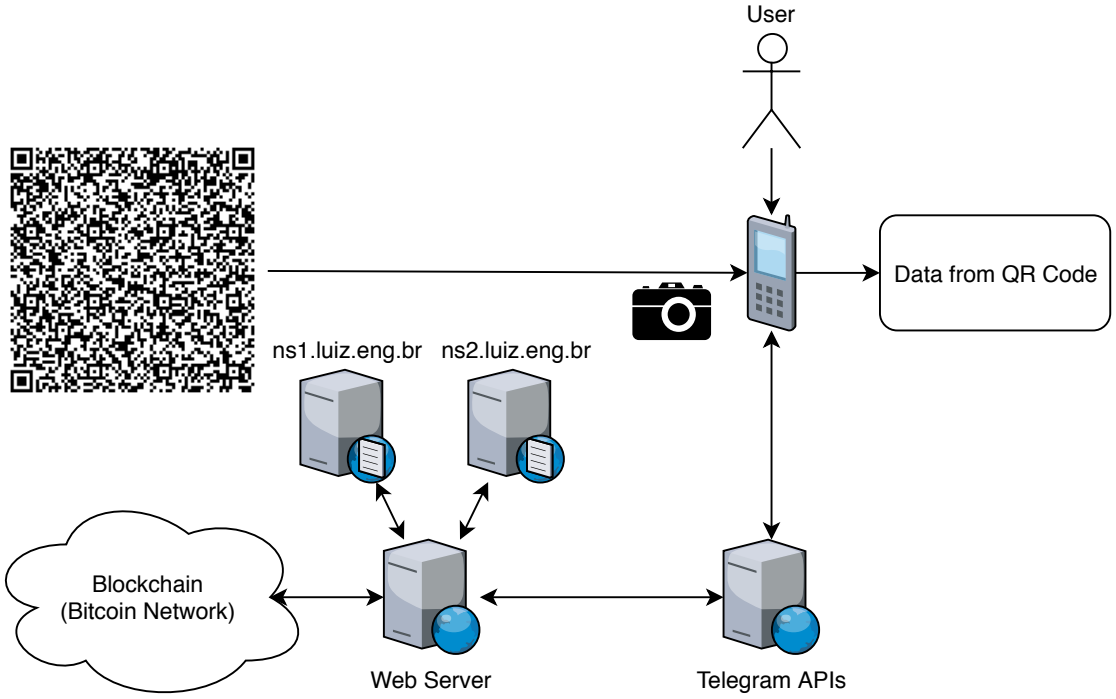


Figure 4.3: Signature verification process.

4.6.1 Telegram Bot

To make the proof of concept implementation simpler and easier, the user interface was implemented using a Telegram Bot instead of a native Android application, as the Telegram Bot already offers a nice UI and no effort needs to be spent towards UI implementation. The first step was to create a new bot on the platform and obtain the API keys.

That was done by starting a conversation with the BotFather (@BotFather) and using the command */newbot*. The BotFather will ask a name and username for the bot and will return the API key. An example of the conversation is shown in Figure 4.4.

The bot backend was implemented in Python using the pyTelegramBotAPI (Eternnoir 2018) module and the behavior is defined by the finite state machine shown in Figure 4.5.

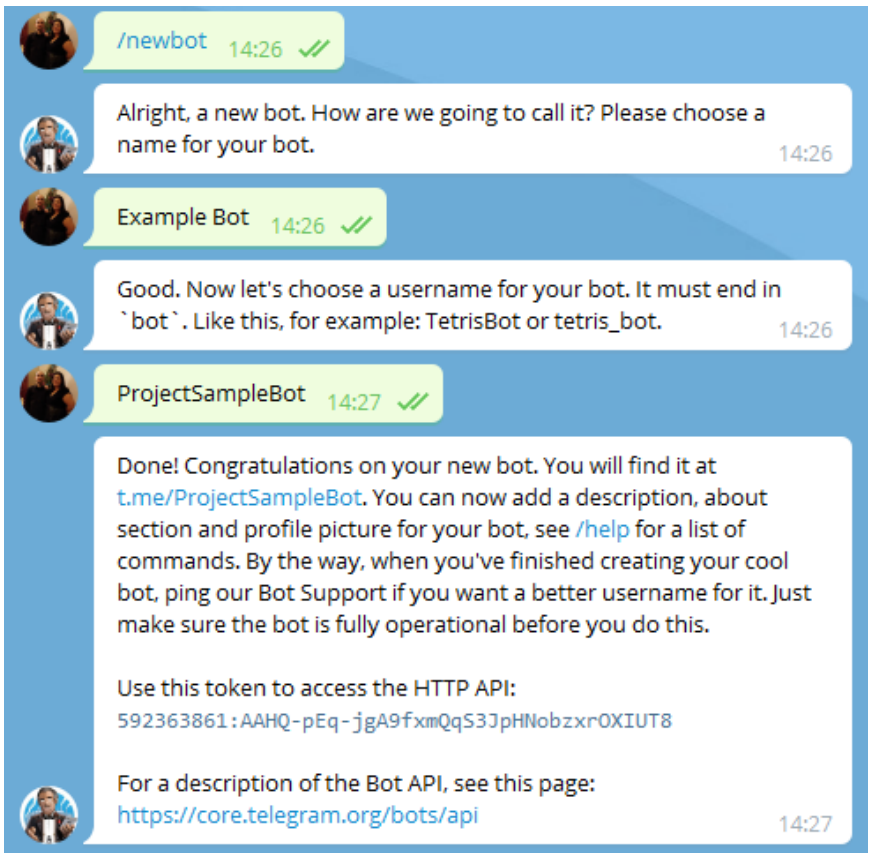


Figure 4.4: BotFather bot creation demo.

When a user needs to validate a signature, it starts a conversation with the bot by sending a message containing the command "/start". The bot will ask for a picture containing the QR Code to be validated.

4.6.1.1 Decoding the QR Code

When the user sends a picture of the QR-Code, the backend uses the ZBarlight module (Polyconseil 2018), a wrapper for the ZBar library (ZBar 2010), to decode the QR-Code and return the data contained in it.

It then splits the data as explained in 3.2.4 and proceeds to fetch the certificate from the DNS to validate the signature.

4.6.1.2 Fetching the Certificate from the DNS

To fetch the certificate from the DNS, the bot obtains the *_id* and *_root* fields from the message. To build the complete name for the DNS record, the field *_root* is mapped to the zone name using an existing table by the bot and the field *_id* is prepended to the zone, forming the complete name.

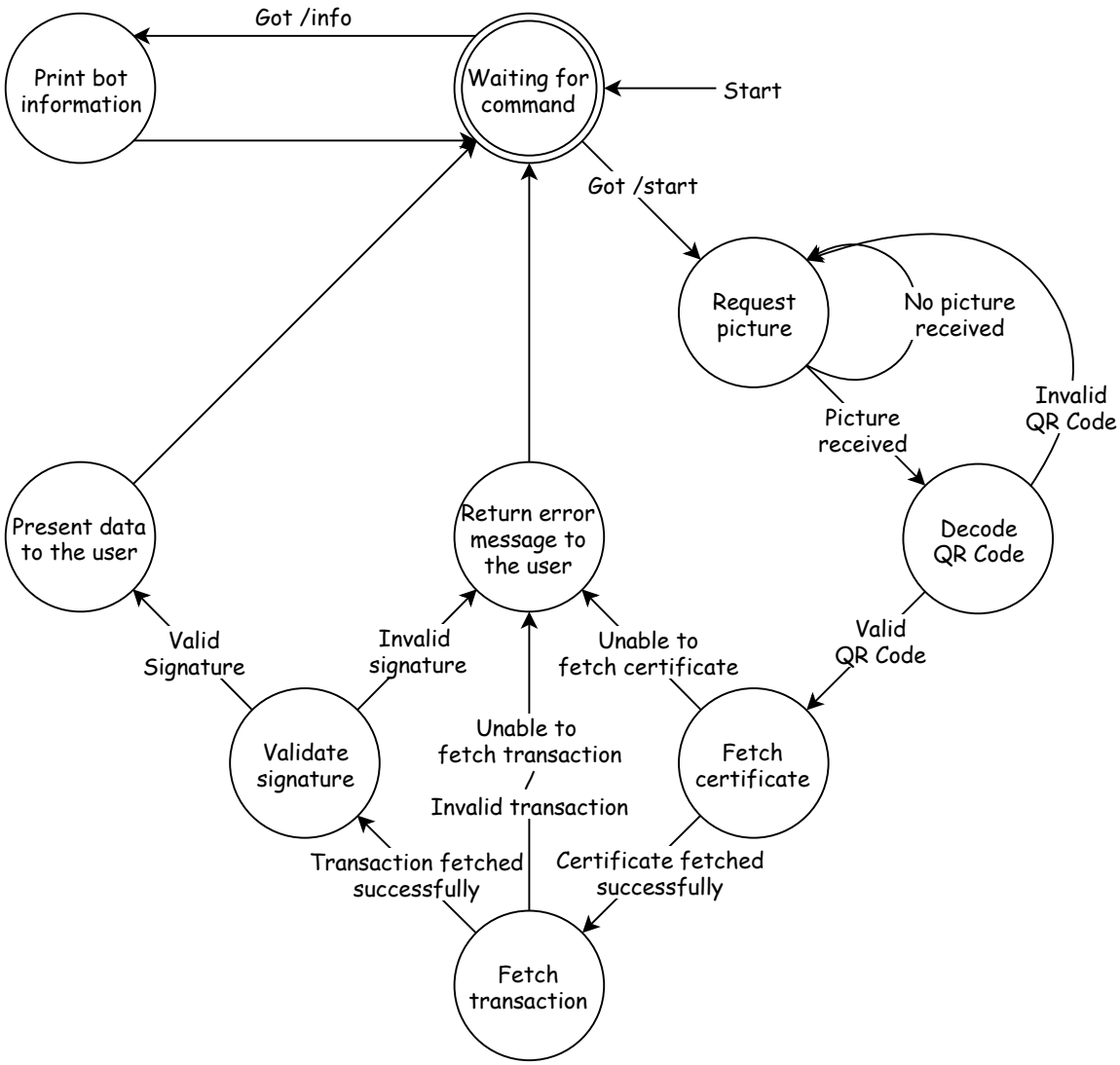


Figure 4.5: Telegram Bot finite state machine.

With the name formed, the bot issues a query for a CERT record to the DNS and after successful retrieval of the DNS record containing the certificate, it proceeds to validate the DNSSEC authentication chain by querying the DNSSEC related records (RRSIG, DNSKEY and DS) for each of the parent zones all the way up to the root. It then uses that data to check the DNSSEC signatures and establish the validity of the authentication chain.

4.6.1.3 Validating the Signature

With the corresponding certificate retrieved, the next step is the signature validation. The signature is decoded to binary and the Hazmat layer functions are used to load the certificate and instantiate a verifier object. The signature and message + 0x1E + TXID are provided to the verifier and validated.

If the signature is deemed valid at this point, the timestamp still needs to be fetch and verified to ensure that the message was signed in between the certificate validity period (notBefore and notAfter) (Boeyen et al. 2008) and a final decision be made.

This process will be described in the next section.

4.6.1.4 Validating the Timestamp

Using the TXID obtained in the first step, the bot sends a request to the Timestamp API and gets a response with the required information, as will be explained in Subsection 4.6.2. The response contains the block timestamp that denotes the moment that this transaction was included in a block and thus, was recognized by the network, proving that the data that has the SHA-256 hash contained in the OP_RETURN opcode existed at that time.

The bot calculates the SHA-256 hash of the message and compares to the one contained in the OP_RETURN opcode. If the hashes match, it means that the transaction information applies to the message and validation can proceed. Using the mentioned timestamp, the bot can verify if the message was signed in between the certificate validity period (after the *notBefore* date and before the *notAfter* date) and finally make a decision about the validity of the signature.

4.6.1.5 Presenting the Data to the User

The final step is to present the result to the user. If any of the steps deems the signature invalid or for any reason the required components for validation cannot be obtained, no data from the message is returned to the user. The only information the user will receive is that the bot could not validate the signature. An example is shown in Figure 4.6.

Figure 4.6: Bot message after invalid signature.

If the signature is verified and deemed valid by the bot, the fields starting with an underscore are stripped from the JSON message and all other fields and their respective data are returned to the user in a proper format, as show in Figure 4.7.

4.6.2 Timestamp API

To make the retrieval of timestamps from the Bitcoin Network easier, a specific API was implemented to provide a simple HTTP interface to make the timestamp retrieval operation, instead of exposing the full JSON-RPC interface of the Bitcoin Core client.

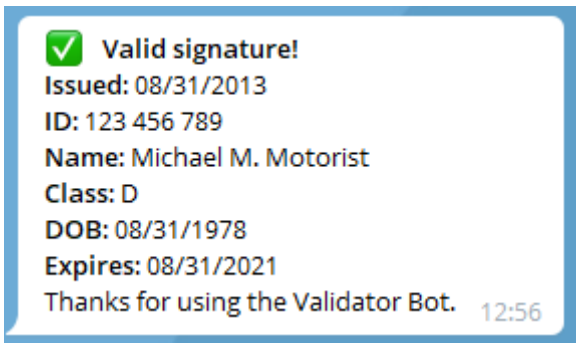


Figure 4.7: Bot message after valid signature.

When a user makes an HTTP GET request to the */stamp* endpoint of the API including the *txid* parameter containing a valid transaction ID, the Tornado process issues a *getrawtransaction* JSON-RPC command to the Bitcoin Core client and parses the response, similar to the example shown in Listing 4.6.

Listing 4.6: Bitcoin Core JSON-RPC response for a published transaction

```
{
    "confirmations": 2687,
    "vin": [{
        "txinwitness": ["30440220248ad6d087cae918cbb062173999ca90a289b14fedd92edb892c8a3b019c
            ↪ 303102205bca1cc54ea6d83352f2405f949506fd519b8e1dc25d6faa73c5bcfe9c7c81a101", "
            ↪ 03d84418a33cd13e473f616b9c6a42037aa3a0cf9d09e06533ac2f7c6f0b1beee5"],
        "scriptSig": {
            "asm": "0014c0c49f665d0f9c140d248e80abb941ada836b59b",
            "hex": "160014c0c49f665d0f9c140d248e80abb941ada836b59b"
        },
        "sequence": 4294967295,
        "vout": 0,
        "txid": "64f1dedaed3c0d90f10e052a38fe50705637e5cb73ab1d5a6c95bc707271db1c"
    }],
    "blockhash": "00000000000002aa4cf25115e3d933d0d57952f1129a6a53ac2e1bd70537dfcc",
    "vsize": 209,
    "version": 2,
    "blocktime": 1524365641,
    "size": 290,
    "vout": [{
        "scriptPubKey": {
            "type": "scripthash",
            "addresses": ["2MtPyBqH8rtER6ppWzdWnGyT2kJgWpqE989"],
            "asm": "OP_HASH160 0c9f83a89d181247534e1f121ea0d9be2f94dc64 OP_EQUAL",
            "reqSigs": 1,
            "hex": "a9140c9f83a89d181247534e1f121ea0d9be2f94dc6487"
        },
        "n": 0,
        "value": 0.0
    }, {
        "scriptPubKey": {
            "type": "scripthash",
            "addresses": ["2N4uWHiXdcRs7CLhne3v9Ymi7qD7LYrL8sa"],
            "asm": "OP_HASH160 7fe6cc15202b00ce6b2a92862eea85ddb72d0ca9 OP_EQUAL",
            "reqSigs": 1,
```

```
34              "hex": "a9147fe6cc15202b00ce6b2a92862eea85ddb72d0ca987"
35          },
36          "n": 1,
37          "value": 0.8999
38      }, {
39          "scriptPubKey": {
40              "type": "nulldata",
41              "asm": "OP_RETURN 6ad2ada45e682fc1836d299e12ffd7b229c1b01f71afe617c73e3372d54ecf9
                    ↪ c",
42              "hex": "6a206ad2ada45e682fc1836d299e12ffd7b229c1b01f71afe617c73e3372d54ecf9c"
43          },
44          "n": 2,
45          "value": 0.0
46      }],
47      "txid": "10eda5106d6c6ea3ba1514c5692070c6780193f8de0cd87d97fc5571fb0c966c",
48      "hash": "8650794e4614f308c9071d827ac063ebe2cb4732b28b5251fe9929ad2507470e",
49      "locktime": 0,
50      "time": 1524365641,
51      "hex": "020000000001011cdb717270bc956c5a1dab73cbe537567050fe382a050ef1900d3ceddadef164000
            ↪ 0000017160014c0c49f665d0f9c140d248e80abb941ada836b59bfffffffff03000000000000000017a
            ↪ 9140c9f83a89d181247534e1f121ea0d9be2f94dc648770235d050000000017a9147fe6cc15202b00
            ↪ ce6b2a92862eea85ddb72d0ca9870000000000000000226a206ad2ada45e682fc1836d299e12ffd7b2
            ↪ 29c1b01f71afe617c73e3372d54ecf9c024730440220248ad6d087cae918cbb062173999ca90a289b1
            ↪ 4fedd92edb892c8a3b019c303102205bca1cc54ea6d83352f2405f949506fd519b8e1dc25d6faa73c5
            ↪ bcfe9c7c81a1012103d84418a33cd13e473f616b9c6a42037aa3a0cf9d09e06533ac2f7c6f0b1beee5
            ↪ 00000000"
52  }
```

The API parses this response and if it has more than 6 confirmations, the OP_RETURN opcode with a valid SHA-256 hash included in the output, returns only the relevant fields, as shown in Listing 4.7.

Listing 4.7: API JSON response for a valid transaction

```
1  {
2      "txid": "10eda5106d6c6ea3ba1514c5692070c6780193f8de0cd87d97fc5571fb0c966c",
3      "hash": "6ad2ada45e682fc1836d299e12ffd7b229c1b01f71afe617c73e3372d54ecf9c",
4      "blocktime": 1524365641,
5      "blockhash": "00000000000002aa4cf25115e3d933d0d57952f1129a6a53ac2e1bd70537dfcc",
6      "status": "OK",
7      "confirmations": 2687
8  }
```

The fields are, the transaction ID on line 2, the hash of the message that was embedded using the OP_RETURN opcode on line 3, the timestamp denoting the instant that the block was mined on line 4, the hash of the block online 5, the API reponse status on line 6 and the number of confirmations for the transaction on line 7.

One possible case, is if an existing transaction is provided but containing no OP_RETURN opcode in the outputs. In that case, the API returns an HTTP 400 error containing the message in Listing 4.8:

Listing 4.8: API JSON response for a transaction without the OP_RETURN opcode

```
{
    "info": "tx has no valid OP_RETURN",
    "code": "-4",
    "status": "error"
}
```

Another case, is to have a valid unconfirmed transaction (less than 6 confirmations). In that case, the API returns the message shown in Listing 4.9.

Listing 4.9: API JSON response for an unconfirmed transaction

```
{
    "info": "tx has not been confirmed yet",
    "code": "-3",
    "status": "error"
}
```

If the transaction ID is not found on the network, the Bitcoin Core client will return an HTTP 500 error with the JSON message shown in Listing 4.10. The API endpoint will then return an HTTP 404 error with the JSON response shown in Listing 4.11.

Listing 4.10: Bitcoin Core JSON response for an inexistent transaction

```
{
    "result": null,
    "error": {
        "code": -5,
        "message": "No such mempool or blockchain transaction. Use gettransaction for wallet
            ↪ transactions."
    },
    "id": "1525385300.7972455-124878"
}
```

Listing 4.11: JSON response for transaction not published

```
{
    "info": "tx not found",
    "code": -2
    "status": "error"
}
```

5 RESULTS

This chapter will present all the steps taken to validate if the implemented proof of concept meets all the expectations established for the mechanism described in Chapter 3.

5.1 DNSSEC AUTHENTICATION CHAIN

In Section 4.2.2 steps were taken to deploy DNSSEC in each of the DNS zones created in the Proof of Concept. In order to correctly validate the features of the mechanism, one of the zones (the globex.luiz.eng.br) did not have the DS RRs published in the parent zone, effectively breaking the DNSSEC authentication chain. In the other hand, the acme.luiz.eng.br zone was setup a full authentication chain.

To properly evaluate the DNSSEC configuration of the zones, the DNSSEC Analyzer tool (Verisign Inc. 2011) from Verisign Labs was used. The tool validates the DNSSEC authentication chain for a given zone and lists all the records used to establish the completeness of it, as shown in Figure 5.1.

in Figure 5.1, we can verify that each of the zones from the root (.) to the acme.luiz.eng.br had proper DNSSEC signatures and corresponding DS RRs on their respective parent zones.

.	Found 3 DNSKEY records for . DS=20326/SHA-256 verifies DNSKEY=20326/SEP DS=19036/SHA-256 verifies DNSKEY=19036/SEP Found 1 RRSIGs over DNSKEY RRset RRSIG=19036 and DNSKEY=19036/SEP verifies the DNSKEY RRset
br	Found 1 DS records for br in the . zone DS=45673/SHA-256 has algorithm RSASHA1 Found 1 RRSIGs over DS RRset RRSIG=39570 and DNSKEY=39570 verifies the DS RRset Found 2 DNSKEY records for br DS=45673/SHA-256 verifies DNSKEY=45673/SEP Found 1 RRSIGs over DNSKEY RRset RRSIG=45673 and DNSKEY=45673/SEP verifies the DNSKEY RRset
eng.br	Found 1 DS records for eng.br in the br zone DS=47872/SHA-1 has algorithm RSASHA1 Found 1 RRSIGs over DS RRset RRSIG=7320 and DNSKEY=7320 verifies the DS RRset Found 1 DNSKEY records for eng.br DS=47872/SHA-1 verifies DNSKEY=47872/SEP Found 1 RRSIGs over DNSKEY RRset RRSIG=47872 and DNSKEY=47872/SEP verifies the DNSKEY RRset
luiz.eng.br	Found 2 DS records for luiz.eng.br in the eng.br zone DS=31389/SHA-1 has algorithm RSASHA1-NSEC3-SHA1 DS=31389/SHA-256 has algorithm RSASHA1-NSEC3-SHA1 Found 1 RRSIGs over DS RRset RRSIG=47872 and DNSKEY=47872/SEP verifies the DS RRset Found 2 DNSKEY records for luiz.eng.br DS=31389/SHA-1 verifies DNSKEY=31389/SEP Found 2 RRSIGs over DNSKEY RRset RRSIG=6488 and DNSKEY=6488 verifies the DNSKEY RRset
acme.luiz.eng.br	Found 2 DS records for acme.luiz.eng.br in the luiz.eng.br zone DS=57049/SHA-256 has algorithm RSASHA256 DS=57049/SHA-1 has algorithm RSASHA256 Found 1 RRSIGs over DS RRset RRSIG=6488 and DNSKEY=6488 verifies the DS RRset Found 2 DNSKEY records for acme.luiz.eng.br DS=57049/SHA-256 verifies DNSKEY=57049/SEP Found 2 RRSIGs over DNSKEY RRset RRSIG=40278 and DNSKEY=40278 verifies the DNSKEY RRset acme.luiz.eng.br A RR has value 198.46.149.130 Found 1 RRSIGs over A RRset RRSIG=40278 and DNSKEY=40278 verifies the A RRset

Figure 5.1: DNSSEC authentication chain for the acme.luiz.eng.br zone.

Figure 5.2 shows that the globex.luiz.eng.br zone itself contained DNSSEC signatures, but the parent zone did not have the DS RRs making it impossible to establish the complete authentication chain.

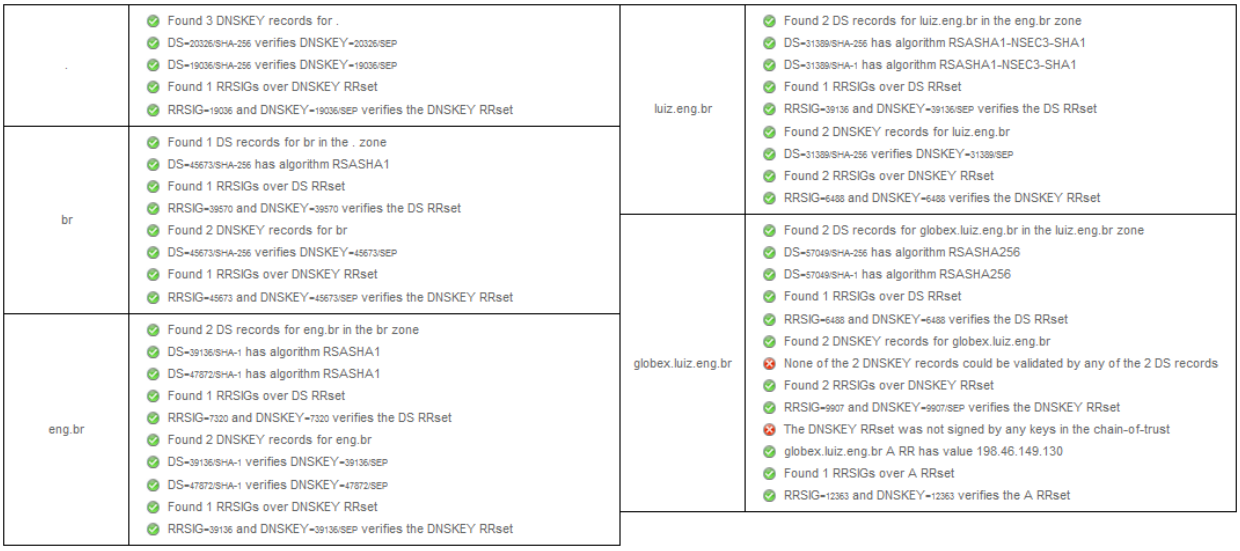


Figure 5.2: DNSSEC authentication chain for the globex.luiz.eng.br zone.

5.2 MECHANISM VALIDATION

With the DNSSEC in place and properly validated, the next step was to issue QR Codes to be later used to validate the mechanism robustness.

During this step, many different QR Codes were generated, with specific characteristics used to evaluate the resilience of the mechanism against different attacks, each aimed at testing the security features of the mechanism in ways that an attacker would possibly do.

The example QR Codes shown in Figure 5.3, containing data from an example driver's license, were produced using the Web API implemented in Section 4.5.1 and the Python 3 interpreter in interactive mode, along with some reused code from the Web API. After the QR Codes were produced, they were used to test if the Telegram Bot would properly validate the untampered one and return the data and alert the user when supplied with the tampered ones.

5.2.1 QR Codes Generation

Each of the QR Codes shown in Figure 5.3 contain different data, simulating different scenarios, as shown on the list below:

(a) Untampered message, untampered TXID and untampered signature;

(b) Untampered message, tampered TXID and untampered signature;

(c) Untampered message, untampered TXID and tampered signature;

(d) Untampered message, untampered TXID and signature generated before certificate validity period;

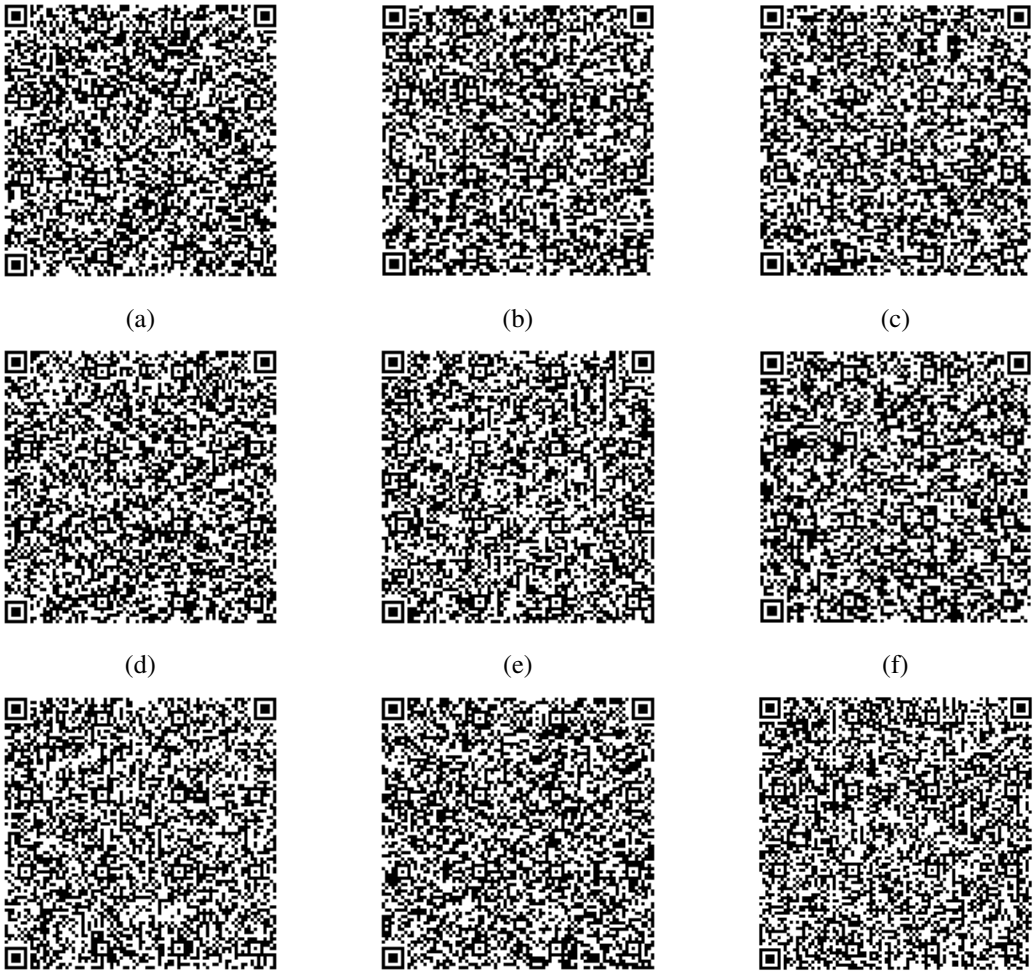


Figure 5.3: QR Codes used for system evaluation

(e) Untampered message, untampered TXID and signature generated after certificate validity period;

(f) Tampered message, untampered TXID and untampered signature;

(g) Untampered message, untampered TXID, untampered signature but DNS zone with DNSSEC problems;

(h) Untampered message, untampered TXID, untampered signature using an unpublished certificate;

(i) Untampered message, untampered TXID, untampered signature and an unknown zone;

(j) Untampered message, untampered TXID and untampered signature verified before transaction confirmation;

Obtaining a digitally signed QR Code using the Web API consists of a simple HTTP POST request containing the JSON formatted message in the body of the request. To issue the requests in a simpler way, the Insomnia REST Client (Software 2018) was used and the request made to obtain the QR Code of Figure 5.3 (a) is shown in Figure 5.4.

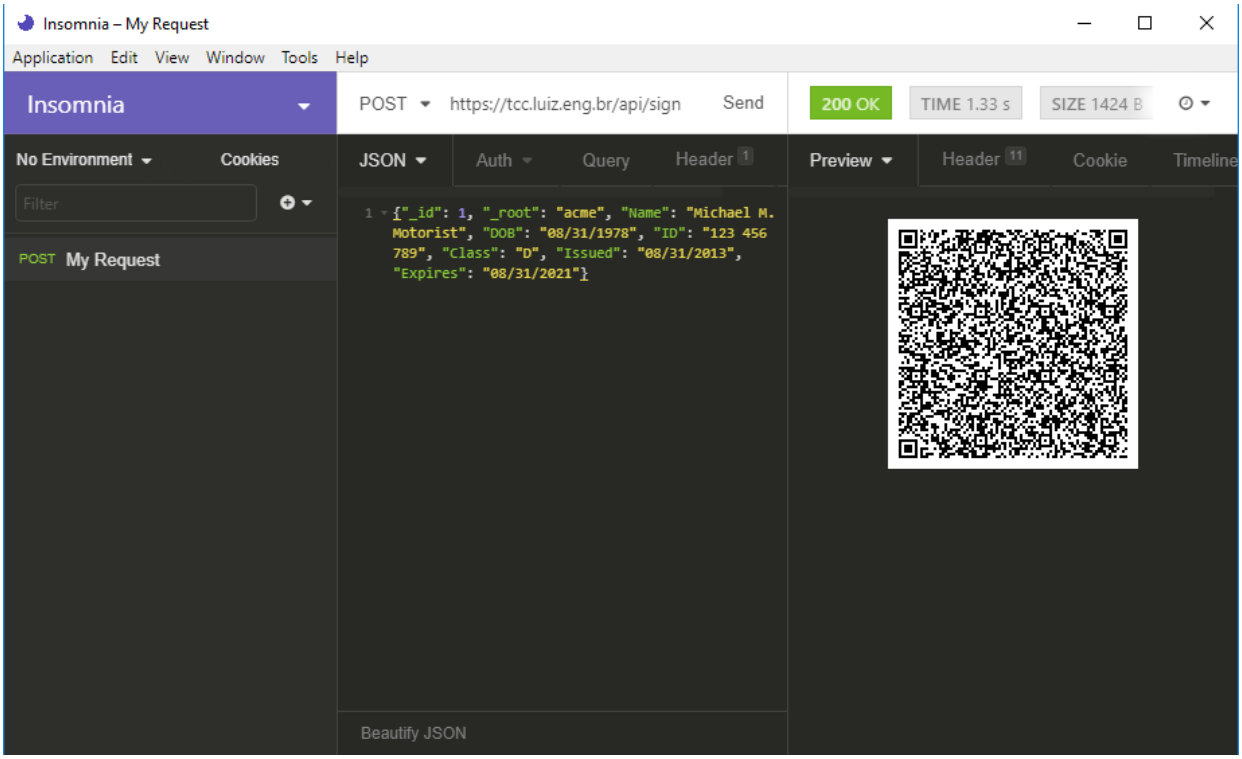


Figure 5.4: HTTP POST request to /sign endpoint using Insomnia.

5.2.2 Validation with the Telegram Bot

In order to test the bot behavior, each of the above QR Codes were validated through the bot. The resulting messages returned by the bot were very descriptive and specific, to demonstrate the bot resilience against each kind of attack. In a real world implementation, it would be done with a simpler message informing that the signature could not be validated.

5.2.2.1 DNSSEC Authentication Chain validation

Part of the mechanism security relies on the use of DNSSEC. Section 4.6.1.2 talked about the validation of the authentication chain in order to establish trustness on the data returned by the DNS server.

Figure 5.5 contains a Wireshark (Wireshark 2018) packet capture of the network traffic between the bot and a DNS resolver, showing queries and responses for DNS records related to DNSSEC used to validate the authentication chain.

```
DNS        84 Standard query 0x357f DNSKEY <Root> OPT
DNS      1195 Standard query response 0x357f DNSKEY <Root> DNSKEY DNSKEY DNSKEY RRSIG OPT
DNS        84 Standard query 0x36c0 DNSKEY <Root> OPT
DNS      1195 Standard query response 0x36c0 DNSKEY <Root> DNSKEY DNSKEY DNSKEY RRSIG OPT
DNS        76 Standard query 0xc838 NS br
DNS       176 Standard query response 0xc838 NS br NS a.dns.br NS b.dns.br NS c.dns.br NS d.dns.br NS e.dns.br NS f.dns.br
DNS        87 Standard query 0xec5b DNSKEY br OPT
DNS       705 Standard query response 0xec5b DNSKEY br DNSKEY DNSKEY RRSIG OPT
DNS        87 Standard query 0xcf12 DS br OPT
DNS       422 Standard query response 0xcf12 DS br DS RRSIG OPT
DNS        80 Standard query 0x8af5 NS eng.br
DNS       180 Standard query response 0x8af5 NS eng.br NS a.dns.br NS b.dns.br NS c.dns.br NS d.dns.br NS e.dns.br NS f.dns.br
DNS        91 Standard query 0x3455 DNSKEY eng.br OPT
DNS       469 Standard query response 0x3455 DNSKEY eng.br DNSKEY RRSIG OPT
DNS        91 Standard query 0x7c28 DS eng.br OPT
DNS       321 Standard query response 0x7c28 DS eng.br DS RRSIG OPT
DNS        85 Standard query 0x13ec NS luiz.eng.br
DNS       121 Standard query response 0x13ec NS luiz.eng.br NS ns1.luiz.eng.br NS ns2.luiz.eng.br
DNS        96 Standard query 0xc180 DNSKEY luiz.eng.br OPT
DNS       298 Standard query response 0xc180 DNSKEY luiz.eng.br DNSKEY DNSKEY RRSIG RRSIG OPT
DNS        96 Standard query 0x2b0b DS luiz.eng.br OPT
DNS       378 Standard query response 0x2b0b DS luiz.eng.br DS DS RRSIG OPT
DNS        92 Standard query 0xe5b0 NS globex.luiz.eng.br
DNS       128 Standard query response 0xe5b0 NS globex.luiz.eng.br NS ns1.luiz.eng.br NS ns2.luiz.eng.br
DNS       103 Standard query 0x3edf DNSKEY globex.luiz.eng.br OPT
DNS       319 Standard query response 0x3edf DNSKEY globex.luiz.eng.br DNSKEY DNSKEY RRSIG RRSIG OPT
DNS       103 Standard query 0x7af7 DS globex.luiz.eng.br OPT
DNS      1225 Standard query response 0x7af7 DS globex.luiz.eng.br NSEC3 RRSIG SOA ns1.luiz.eng.br RRSIG NSEC3 RRSIG OPT
```

Figure 5.5: Wireshark packet capture of the bot validating the DNSSEC authentication chain.

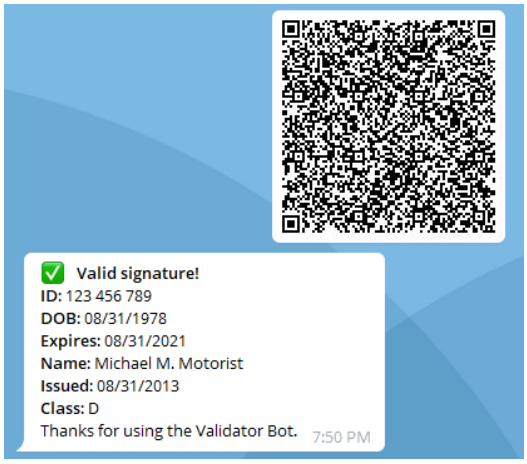
The implementation of the authentication chain validation used in the bot was heavily based on the Electrum Bitcoin Client DNSSEC library (Electrum 2018), with modifications to make the methods simpler and to add the DNS Checking Disabled (CD) flag to the queries.

5.2.2.2 Validating QR Codes with the bot

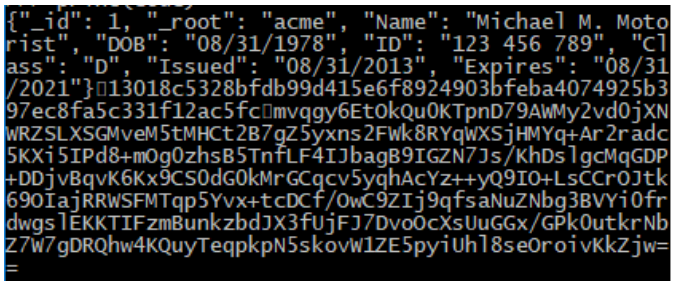
In order to test if the bot could properly validate the QR Code in all the proposed scenarios, each QR Code was sent to the bot and the presented behavior was registered and compared with what was expected.

Figure 5.6a shows the validation result of scenario (a) by the bot. The data displayed by the bot is the data obtained from the JSON message encoded in the QR Code, as shown in Figure 5.6b. The output shown was obtained by manually decoding the code using the Python Interactive Mode and shows the JSON message, followed by a small square that represents the 0x1E character (the square is a way to represent non-printable ASCII characters), the Bitcoin transaction ID, another small square representing the 0x1F character and finally the digital signature encoded in Base64.

The QR Code in Figure 5.6a represented a scenario of a perfect use case, where no tampering was tried.

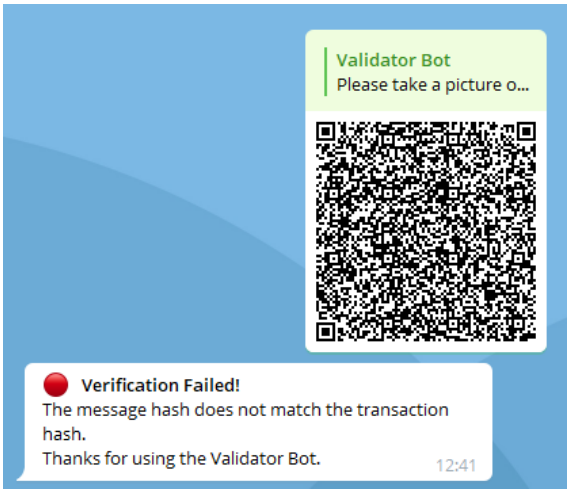


(a) Bot output for scenario A

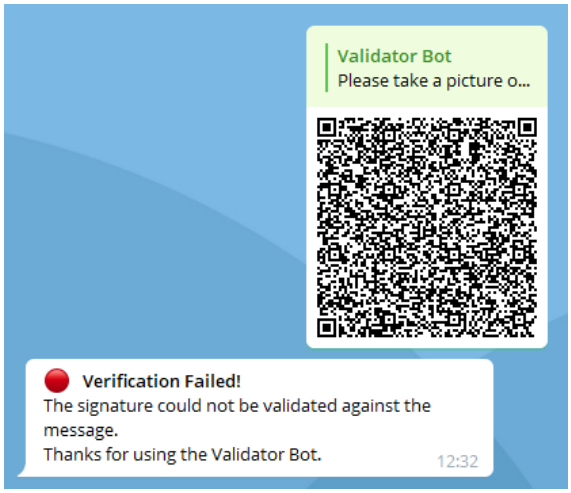


(b) QR Code contents for scenario A

Figure 5.6: Bot output and QR Code contents for scenario A.



(a) Bot output for scenario B



(b) Bot output for scenario C

Figure 5.7: Bot outputs for scenarios B and C

Scenario (b) is a QR Code that had its transaction ID tampered with. In this specific case, a transaction ID for another set of data was appended to the message and then signed. Although it is unlikely to happen, the intention of evaluating it was to show the robustness of the bot against different kinds of attacks, including the one where an attacker in possession of the private key corresponding to an expired certificate tried to sign a fake message. If he simply published a new transaction in the Bitcoin network, the timestamp would not be in between the certificate validity period. To surpass this problem, he could try including the transaction ID of another message issued during the validity period of the certificate.

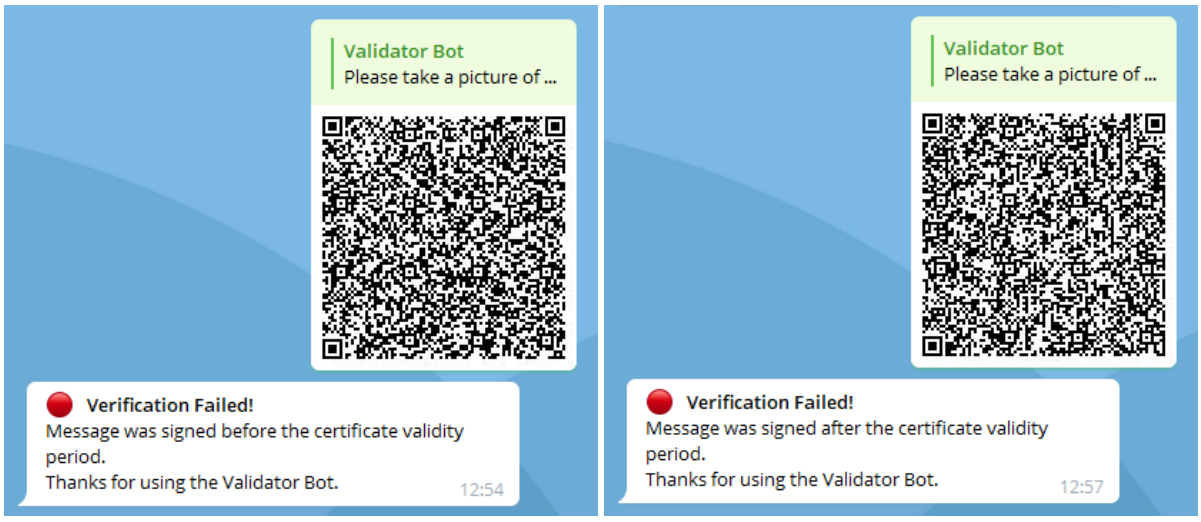
Figure 5.7a shows that the bot was able to detect that the digest of the message and the digest contained in the transaction did not match and returned a message informing the user, frustrating the attacker intentions.

Scenario (c) tests the case where an attacker has a valid message with its corresponding transaction ID but a signature that was taken from another QR Code and so, does not correspond to the data. Figure 5.7b shows that the bot properly detected that the signature did not match the data and informed the user.

Scenarios (d) and (e) were proposed in order to check if the bot would properly validate the certificates validity period, where in scenario (d), the data was signed before the certificate was valid and in scenario (e) the data was signed after the certificate had expired. The bot was able to detect and inform the user in both cases by using the timestamp contained in the Bitcoin transaction ID to establish the date of signature, as shown in Figures 5.8a and 5.8b.

Figure 5.9a shows the validity period of the certificate used in scenario d, starting in June of 2020 and ending in May of 2030 while Figure 5.9b shows information about the Bitcoin transaction, the most important one being the *blocktime* that denotes the timestamp that represent the exact instant when the first block including this transaction was mined on the network. One detail to note is that the *txid* and *blockhash* fields were truncated in the picture, making them appear shorter than they really are.

To assure the date of signature, the bot relied on the fact that an attacker would need a lot of computation power to alter any information already published and accepted by the nodes in the Bitcoin network, as explained in Chapter 2.

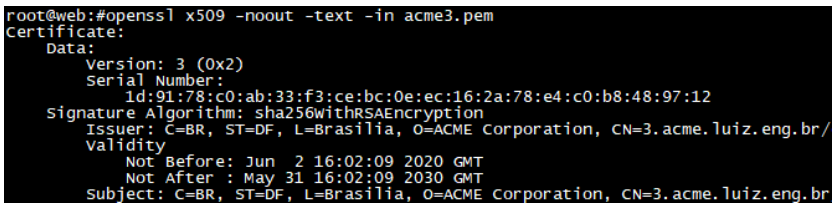


(a) Bot output for scenario D

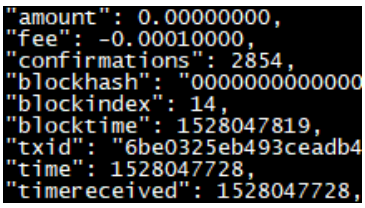
(b) Bot output for scenario E

Figure 5.8: Bot outputs for scenarios D and E

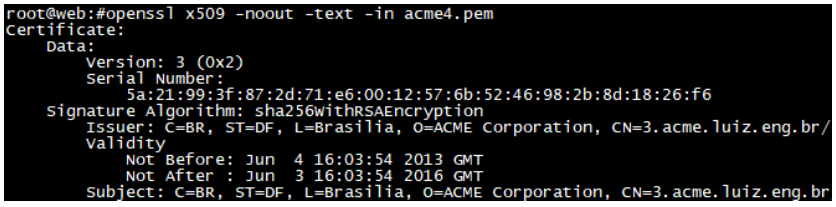
Scenario (f) simulates the simplest attack case: an ill-intentioned person simply tried to alter the information of the JSON message, in this specific case, the driver's name. When changing the data, the attacker not only invalidated the digital signature but also changed the message digest, that now will not match the one contained in the Bitcoin transaction. Because of the way the bot validates the data in the QR Code, it will alert the user that the signature could not be validated against the message and will not proceed further and compare the message digests. Figure 5.10a shows the bot response for this scenario.



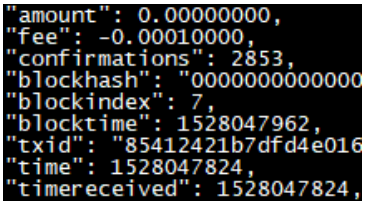
(a) Certificate dates for scenario D



(b) Timestamps for scenario D



(c) Certificate dates for scenario E



(d) Timestamps for scenario E

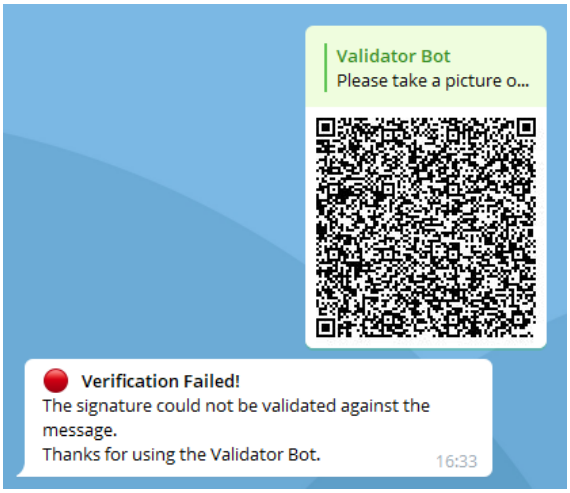
Figure 5.9: Certificate dates and timestamps for scenarios D and E

Another possible way for an attacker to cheat the mechanism is to try to serve a certificate from a malicious DNS server. In scenario (g), the malicious server is presenting the globex.luiz.eng.br DNS zone, with DNSSEC signatures that do not match the key specified in the DS records in the luiz.eng.br zone. When fetching the certificate, the bot performs a full DNSSEC authentication chain validation, as shown in Figure 5.5, and by doing so, is able to detect the mismatch between the signing key and the DS records in the parent zone.

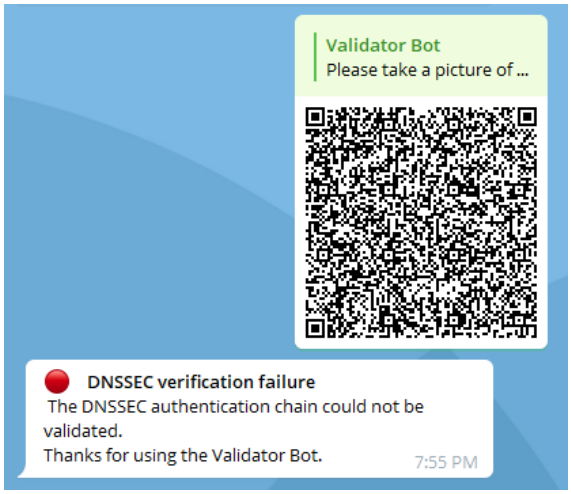
When using a Telegram Bot as a user interface, this problem is harder to happen, as the piece of software running the validations would be running in a controlled server, less susceptible to attacks like that. But in the case where the validation is done using a native Android application, for example, the DNS resolvers provided by the smartphone could not be trusted, hence the importance in testing this scenario.

In scenario (h), the QR Code data points to a certificate under the record 2.acme.luiz.eng.br, that does not exist in the acme.luiz.eng.br. There are two possible reasons for a certificate record to not be found in a zone: the record never existed or the certificate was revoked. Because the system relies self-signed certificates, the use of Certificate Revocation Lists (CRLs) is not possible (Housley et al. 2002), so when revoking a certificate it simply gets removed from the DNS. The consequence is that when trying to validate a message signed using the key pair tied to that certificate, like the example in Figure 5.3h, the bot will not be able to retrieve the certificate and by consequence, invalidate the signature.

Figure 5.10c contains the response from the bot when trying to validate a signature that has no corresponding certificate published in the DNS.

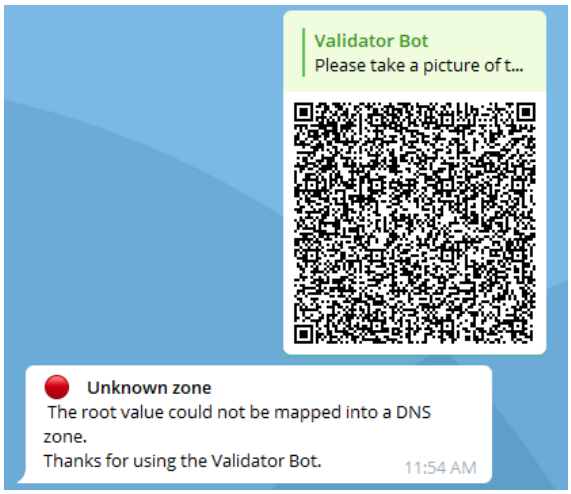


(a) Bot output for scenario F



(b) Bot output for scenario G

(c) Bot output for scenario H



(d) Bot output for scenario I

Figure 5.10: Bot outputs for scenarios F, G, H and I

As explained in Section 4.6.1.2, to build the complete DNS name when retrieving the certificates, the bot maps the value contained in the *_root* field to a zone name using a table built into it. That way, if an attacker crafts a message with an unknown value in the *_root* field, the bot is not able to map it into a valid zone, a case simulated in scenario (i). When the bot does not find an entry in the table that corresponds to the value in the message it informs the user and ends the validation process, as seen in Figure 5.10d.

Scenario (j) shows the disadvantage of implementing the secure timestamping using the Bitcoin blockchain. Because of the bigger block time of the Bitcoin protocol, a transaction is not confirmed instantly, meaning that the bot is not able to validate a signature right after issuance. While waiting for the 6 confirmations of the transaction, the bot is not able to deem the transaction as invalid or not as well as the signature, so instead of informing the user that the signature is invalid, it simply informs that the transaction does not have enough confirmations and asks the user to try again after one hour, as shown in Figure 5.11.

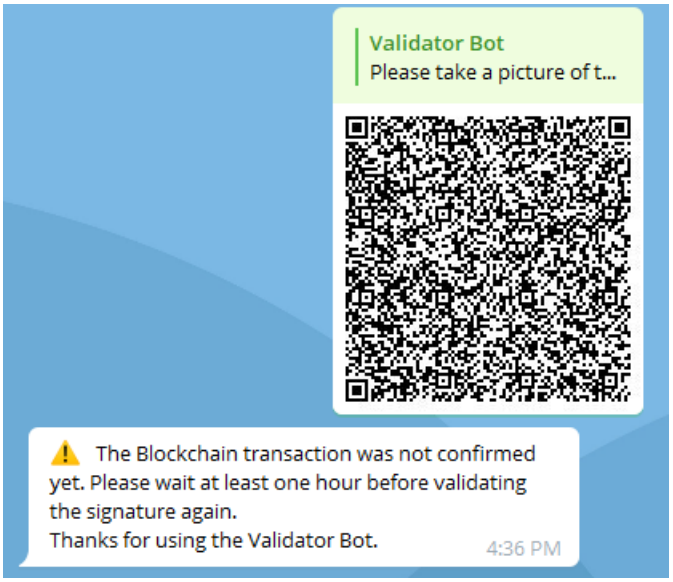


Figure 5.11: Bot output for scenario J.

5.3 ATTACK VECTORS

The use of the proposed technologies, bring security implications that were not explicitly addressed in this work. For example, an attacker may target the components of the system like the message signing services or even try to compromise the keys, instead of attacking the mechanism itself.

Another aspect is that because the system relies on DNS for key retrieval, another attack that may be tried is a Denial of Service, by making the DNS servers serving the keys unavailable and thus, impacting signature validation.

6 CONCLUSION

With the increased popularity of smartphones, the issuance of digital documents by governments and entities has grown, enabling citizens to carry digital versions of their documents in their smartphones instead of a physical paper version in the wallet.

This work proposed a mechanism to authenticate the information contained in the digital document using Digital Signatures, DNSSEC and Blockchain, guaranteeing integrity, authentication and non-repudiation. The proposed design was used to implement a proof of concept that was later tested against different attack scenarios in order to evaluate its resilience.

During the research phase, one of the issues that arose was the inability to verify if a document was signed during the certificate validity period due to the lack of a timestamp in the message. The use of a Time Stamping Authority as per RFC 3161 was unfeasible given the amount of data needed and the restricted storage space in the QR Codes. To overcome this problem, a proof of existence mechanism relying on the Bitcoin Blockchain was developed and effectively addressed the problem, as evaluated in the results.

Different from previous work, the proposed mechanism distributes the public keys used for signature verification in the form of digital certificates distributed using the DNS system with DNSSEC implemented for security, enabling entities issuing documents to roll out new keys without having to update the verifying applications and without human intervention. The mechanism also offers reduced complexity as the use of PKI is completely optional and the revocation of certificates is done by simply removing them from the DNS zone. The simplicity brings one disadvantage: after revoking a certificate, all documents signed using the corresponding private key are deemed invalid, even if issued before the certificate revocation.

The mechanism can be implemented using any Digital Signature algorithm supporting X.509 certificates, bringing flexibility to the system, enabling it to fit an algorithm that has faster public key operations or an algorithm that has faster private key operations, although the difference in performance is not very big. A few examples of supported algorithms are Elliptic Curve Cryptography (ECC), Digital Signature Algorithm (DSA), Elliptic Curve Digital Signature Algorithm (ECDSA) and RSA.

The implemented proof of concept was evaluated with a focus on the security of the system and in order to properly evaluate it, 10 scenarios were proposed covering aspects like message tampering, signature tampering, Bitcoin Transaction ID tampering, proper validation of the certificate validity period, validation of the DNSSEC authentication chain, rogue certificate detection, rogue zone detection, validation before transaction confirmation as well as a valid working scenario. In the 8 attack scenarios, the proof of concept was able to deter and properly identify the attacks being tried, demonstrating the robustness of the proposed mechanism.

From a performance point of view, the signature validation process is in theory able to scale together with the DNS system, although one limiting factor perceived was the mechanism used for retrieving the transaction information from the Bitcoin network, in the implemented proof of concept, a Web API. The document signing process scalability is limited by the cost of publishing a Bitcoin transaction and the theoretical throughput limit of 7 transactions per second on the Bitcoin protocol. A possible solution for this problem is proposed in Section 6.1.

Is is important to mention that not all the security aspects of this mechanism were evaluated in this work. A real-world implementation of what is described here must be subject to penetration tests in order to make it more robust and eventually find out vulnerabilities and flaws not perceived by the author.

6.1 FUTURE WORK

During the execution of this project additional ideas emerged and could not be included because of a time or complexity constraint. The list below contains a list of the ideas and a brief explanation about them:

- Implementation of the validating application using a native Android or iOS application, eliminating a third-party (the Telegram Bot API) from the system and bringing more reliability;
- Use of smart contracts to implement the distributed authenticated timestamping, increasing the scalability of the document signing process and reducing costs;
- Propose a way of including a picture of the person together with the data, be it through an external URL or other method.

BIBLIOGRAPHY

Bitcoin Developer Guide 2018 BITCOIN Developer Guide. 2018. Available from Internet: <https://bitcoin.org/en/developer-guide>.

Boeyen et al. 2008 BOEYEN, S.; SANTESSON, S.; POLK, T.; HOUSLEY, R.; FARRELL, S.; COOPER, D. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC Editor, 2008. RFC 5280. (Request for Comments, 5280). Available from Internet: <https://rfc-editor.org/rfc/rfc5280.txt>.

Bray 2014 BRAY, T. *The javascript object notation (json) data interchange format*. 2014.

Breitinger and Gipp 2017 BREITINGER, C.; GIPP, B. Virtualpatent - enabling the traceability of ideas shared online using decentralized trusted timestamping. In: *Proceedings of the 15th International Symposium of Information Science*. [S.l.: s.n.], 2017.

Brito and Castillo 2013 BRITO, J.; CASTILLO, A. *Bitcoin: A primer for policymakers*. [S.l.]: Mercatus Center at George Mason University, 2013.

Brown et al. 2010 BROWN, D. R. L.; DANG, Q.; POLK, T.; SANTESSON, S.; MORIARTY, K. *Internet X.509 Public Key Infrastructure: Additional Algorithms and Identifiers for DSA and ECDSA*. RFC Editor, 2010. RFC 5758. (Request for Comments, 5758). Available from Internet: <https://rfc-editor.org/rfc/rfc5758.txt>.

Buffers 2011 BUFFERS, P. *Google's data interchange format*. 2011.

Clark and Essex 2012 CLARK, J.; ESSEX, A. Commitcoin: Carbon dating commitments with bitcoin. In: SPRINGER. *International Conference on Financial Cryptography and Data Security*. [S.l.], 2012. p. 390–398.

CoinMarketCap 2018 COINMARKETCAP. *Bitcoin (BTC) price, charts, market cap, and other metrics*. 2018. (Accessed on 2018-06-11). Available from Internet: <https://coinmarketcap.com/currencies/bitcoin/>.

Consortium 2018 CONSORTIUM, I. S. *BIND Open Source DNS Server*. 2018. Available from Internet: <https://www.isc.org/downloads/bind/>. Cited: 2018-06-14.

DigitalOcean 2016 DIGITALOCEAN. *How To Setup DNSSEC on an Authoritative BIND DNS Server*. DigitalOcean, 2016. Available from Internet: <https://www.digitalocean.com/community/tutorials/how-to-setup-dnssec-on-an-authoritative-bind-dns-server--2>.

DMV 2013 DMV, N. Y. S. *Sample New York State DMV Photo Documents*. 2013. Available from Internet: <https://dmv.ny.gov/id-card/sample-photo-documents>. Cited: 2017-07-10.

Dukhovni and Hardaker 2015 DUKHOVNI, V.; HARDAKER, W. *The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance*. RFC Editor, 2015. RFC 7671. (Request for Comments, 7671). Available from Internet: <https://rfc-editor.org/rfc/rfc7671.txt>.

Electrum 2018 ELECTRUM, S. *Electrum DNSSEC Library*. 2018. Available from Internet: <https://github.com/spesmilo/electrum/blob/master/lib/dnssec.py>.

Eternnoir 2018 ETERNNOIR. *eternnoir/pyTelegramBotAPI*. 2018. Available from Internet: <https://github.com/eternnoir/pyTelegramBotAPI>.

FIPS 2013 FIPS, P. 186-4. *Digital Signature Standard (DSS)*, 2013.

FIPS 2015 FIPS, P. 180-4. *Secure hash standard (SHS)*, 2015.

Garain and Halder 2008 GARAIN, U.; HALDER, B. On automatic authenticity verification of printed security documents. In: IEEE. *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*. [S.l.], 2008. p. 706–713.

Gipp, Meuschke and Gernandt 2015 GIPP, B.; MEUSCHKE, N.; GERNANDT, A. Decentralized trusted timestamping using the crypto currency bitcoin. *arXiv preprint arXiv:1502.04015*, 2015.

Housley et al. 2002 HOUSLEY, R.; POLK, T.; FORD, D. W. S.; SOLO, D. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC Editor, 2002. RFC 3280. (Request for Comments, 3280). Available from Internet: <https://rfc-editor.org/rfc/rfc3280.txt>.

Housley, Schaad and Kaliski 2005 HOUSLEY, R.; SCHAAD, J.; KALISKI, B. *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC Editor, 2005. RFC 4055. (Request for Comments, 4055). Available from Internet: <https://rfc-editor.org/rfc/rfc4055.txt>.

IANA 2017 IANA. *Domain Name System Security (DNSSEC) Algorithm Numbers*. 2017. Available from Internet: <https://www.iana.org/assignments/dns-sec-alg-numbers/dns-sec-alg-numbers.xhtml>.

Infoblox 2018 INFOBLOX. *What is Domain Name System Security Extensions (DNSSEC)?* 2018. Available from Internet: <https://www.infoblox.com/glossary/domain-name-system-security-extensions-dnssec/>. Cited: 2018-06-14.

Jansma and Arrendondo 2004 JANSMA, N.; ARRENDONDO, B. Performance comparison of elliptic curve and rsa digital signatures. *nicj. net/files*, 2004.

Josefsson 2006 JOSEFSSON, S. *Storing Certificates in the Domain Name System (DNS)*. RFC Editor, 2006. RFC 4398. (Request for Comments, 4398). Available from Internet: <https://rfc-editor.org/rfc/rfc4398.txt>.

Josefsson 2006 JOSEFSSON, S. *The Base16, Base32, and Base64 Data Encodings*. RFC Editor, 2006. RFC 4648. (Request for Comments, 4648). Available from Internet: <https://rfc-editor.org/rfc/rfc4648.txt>.

Kaliski and Staddon 1998 KALISKI, B.; STADDON, J. *PKCS #1: RSA Cryptography Specifications Version 2.0*. RFC Editor, 1998. RFC 2437. (Request for Comments, 2437). Available from Internet: <https://rfc-editor.org/rfc/rfc2437.txt>.

Kurose and Ross 2010 KUROSE, J.; ROSS, K. *Computer Networking: A Top-down Approach*. [S.l.]: Addison-Wesley, 2010. (Pearson International edition). ISBN 9780136079675.

Lincolnloop 2018 LINCOLNLOOP. *lincolnloop/python-qrcode*. 2018. Available from Internet: <https://github.com/lincolnloop/python-qrcode>.

Mockapetris 1987 MOCKAPETRIS, P. *Domain names - implementation and specification*. RFC Editor, 1987. RFC 1035. (Request for Comments, 1035). Available from Internet: <https://rfc-editor.org/rfc/rfc1035.txt>.

Moriarty et al. 2016 MORIARTY, K.; KALISKI, B.; JONSSON, J.; RUSCH, A. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC Editor, 2016. RFC 8017. (Request for Comments, 8017). Available from Internet: <https://rfc-editor.org/rfc/rfc8017.txt>.

Nakamoto 2008 NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. 2008.

NIST 2013 NIST, S. Nist sp 800-81-2 secure domain name system (dns) deployment guide. 2013.

NIST 2015 NIST, S. *NIST 800-57 Part 3 Rev 1, Recommendation for Key Management: Part 3: Application-Specific Key Management Guidance*. 2015.

Pillow: the friendly PIL fork 2018 PILLOW: the friendly PIL fork. 2018. Available from Internet: <http://python-pillow.org/>.

Polk et al. 2009 POLK, T.; HOUSLEY, R.; TURNER, S.; BROWN, D. R. L.; YIU, K. *Elliptic Curve Cryptography Subject Public Key Information*. RFC Editor, 2009. RFC 5480. (Request for Comments, 5480). Available from Internet: <https://rfc-editor.org/rfc/rfc5480.txt>.

Polyconseil 2018 POLYCONSEIL. *Polyconseil/zbarlight*. 2018. Available from Internet: <https://github.com/Polyconseil/zbarlight>.

Python 2018 PYTHON. *The Python Tutorial*. 2018. Available from Internet: <https://docs.python.org/3/tutorial/index.html>. Cited: 2018-06-14.

Python Cryptographic Authority 2013 Python Cryptographic Authority. *Welcome to pyca/cryptography — Cryptography 2.0.dev1 documentation*. 2013. Https://cryptography.io/en/latest/. (Accessed on 2017-07-10).

Rose et al. 2005 ROSE, S.; LARSON, M.; MASSEY, D.; AUSTEIN, R.; ARENDS, R. *DNS Security Introduction and Requirements*. RFC Editor, 2005. RFC 4033. (Request for Comments, 4033). Available from Internet: <https://rfc-editor.org/rfc/rfc4033.txt>.

Rose et al. 2005 ROSE, S.; LARSON, M.; MASSEY, D.; AUSTEIN, R.; ARENDS, R. *Resource Records for the DNS Security Extensions*. RFC Editor, 2005. RFC 4034. (Request for Comments, 4034). Available from Internet: <https://rfc-editor.org/rfc/rfc4034.txt>.

Software 2018 SOFTWARE, F. K. *Insomnia*. 2018. Available from Internet: <https://insomnia.rest/>.

Soon 2008 SOON, T. J. Qr code. *Synthesis Journal*, v. 2008, p. 59–78, 2008.

Telegram 2018 TELEGRAM. *Telegram F.A.Q.* 2018. Available from Internet: <https://telegram.org/faq>.

Tornado 2018 TORNADO. *Tornado Web Server - Tornado 5.0.2 documentation*. 2018. Available from Internet: <http://www.tornadoweb.org/en/stable/>. Cited: 2018-06-14.

Verisign Inc. 2011 Verisign Inc. *DNSSEC Analyzer*. 2011. Https://dnssec-debugger.verisignlabs.com/. (Accessed on 2017-07-10).

Videntity 2013 VIDENTITY. *How to Serve Public Certificates with BIND for the Direct Project*. 2013. Available from Internet: <http://www.videntity.com/2013/08/how-to-serve-public-certificates-in-bind-for-the-direct-project/>.

Warasart and Kuacharoen 2012 WARASART, M.; KUACHAROEN, P. *Paper-based Document Authentication using Digital Signature and QR Code*. 2012.

WAVE 2018 WAVE, D. *DENSO WAVE, the Inventor of QR Code*. 2018. Available from Internet: <http://www.qrcode.com/en/>.

Wiki 2018 WIKI, B. *Block hashing algorithm*. 2018. Available from Internet: <https://en.bitcoin.it/wiki/Block_hashing_algorithm>.

Wiki 2018 WIKI, B. *Controlled Supply*. 2018. Available from Internet: <https://en.bitcoin.it/wiki/Controlled_supply>.

Wiki 2018 WIKI, B. *Network*. 2018. Available from Internet: <https://en.bitcoin.it/wiki/Network>.

Wiki 2018 WIKI, B. *Proof of work*. 2018. Available from Internet: <https://en.bitcoin.it/wiki/Proof_of_work>.

Wiki 2018 WIKI, B. *Transaction*. 2018. Available from Internet: <https://en.bitcoin.it/wiki/Transaction>.

Wireshark 2018 WIRESHARK. *Download*. 2018. Available from Internet: <https://www.wireshark.org/>.

ZBar 2010 ZBAR. *ZBar bar code reader*. 2010. Available from Internet: <http://zbar.sourceforge.net/>.

Zuccherato et al. 2001 ZUCCHERATO, R.; CAIN, P.; ADAMS, D. C.; PINKAS, D. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*. RFC Editor, 2001. RFC 3161. (Request for Comments, 3161). Available from Internet: <https://rfc-editor.org/rfc/rfc3161.txt>.

APPENDIX

I.1 DNS HELPER SCRIPTS

Listing 1: Python script to generate CERT RRs [Adapted from: (Videntity 2013)]

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# vim: ai ts=4 sts=4 et sw=4

# BIND Certificate Convert - Read in a host name and PEM certificate path and write out the
    ↪ corresponding line for
# BIND's zone file

import argparse
import dns.rdata
import dns.dnssec
import dns.rdataclass
import dns.rdatatype
import dns.rdtypes.ANY.DNSKEY

from cryptography import x509
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization

algorithms = {
    "1.2.840.113549.1.1.4": "RSAMD5",
    "1.2.840.113549.1.1.5": "RSASHA1",
    "1.2.840.113549.1.1.11": "RSASHA256",
    "1.2.840.113549.1.1.13": "RSASHA512",
    "1.2.840.10045.4.3.2": "ECDSAP256SHA256",
    "1.2.840.10045.4.3.3": "ECDSAP384SHA384",
    "1.2.840.10040.4.3": "DSA"
}


def find_between(s, first, last):
    try:
        start = s.index(first) + len(first)
        end = s.index(last, start)
        return s[start:end]
    except ValueError:
        return ""


def bind_cert_convert(hostname, certpath):
    # Loading the serialized public key form the X.509 certificate
    with open(certpath, "rb") as cert_file:
        cert = x509.load_pem_x509_certificate(cert_file.read(), default_backend())

    # public_key = cert.public_key()
    #
    pem = cert.public_bytes(
        encoding=serialization.Encoding.PEM
    )

    der = cert.public_bytes(
        encoding=serialization.Encoding.DER
    )

```

```
    output = pem.decode()

    algo = algorithms.get(cert.signature_algorithm_oid.dotted_string)

    clean_pk = find_between(output,
                            "-----BEGIN CERTIFICATE-----",
                            "-----END CERTIFICATE-----")

    clean_pk = clean_pk.replace("\n", "")
    decoded_clean_pk = der

    dnskey = dns.rdtypes.ANY.DNSKEY.DNSKEY(dns.rdataclass.IN,
                                           dns.rdatatype.DNSKEY, 0, 0,
                                           dns.dnssec.RSASHA256,
                                           decoded_clean_pk)

    bind_entry = "%s\tIN\tCERT\tPKIX %s %s %s" % (hostname,
                                                  dns.dnssec.key_id(dnskey),
                                                  algo,
                                                  clean_pk)

    print(bind_entry)


if __name__ == "__main__":
    # Parsing command line arguments
    parser = argparse.ArgumentParser(
        description='Generate a CERT DNS record containing the certificate to be published')
    parser.add_argument('hostname', type=str, help='The hostname for the DNS entry')
    parser.add_argument('certfile', type=str, help='File containing the PEM encoded
        ↪ certificate and public key')

    args = parser.parse_args()

    bind_cert_convert(args.hostname, args.certfile)
```

Listing 2: Bash script for automated zone signing [Adapted from: (DigitalOcean 2016)]

```
#!/bin/bash
die() { echo "$@" 1>&2 ; exit 1; }
[[ $# -gt 0 ]] || die "Invalid number of arguments"
PDIR=`pwd`
ZONEDIR="/etc/bind/zones"
ZONE=$1
ZONEFILE="$ZONEDIR/$ZONE.zone"
DNSSERVICE="bind9"
cd $ZONEDIR
[[ -f $ZONEFILE ]] || die "Zone file does not exists"
SERIAL=`/usr/sbin/named-checkzone $ZONE $ZONEFILE | egrep -ho '[0-9]{10}'`
sed -i 's/'$SERIAL'/'$(($SERIAL+1))'/' $ZONEFILE
/usr/sbin/dnssec-signzone -A -3 $(head -c 1000 /dev/urandom | sha1sum | cut -b 1-16) -N
    ↪ increment -o $ZONE -t $ZONEFILE
service $DNSSERVICE reload
cd $PDIR
```

I.2 CERT DNS ENTRY EXAMPLE

Listing 3: Example CERT RR in the acme.luiz.eng.br zone [Source: Author]

```
1  1               IN      CERT    PKIX 9461 RSASHA256
2  ↪ MIIEJjCCAw6gAwIBAgIBATANBgkqhkiG9w0BAQsFADCBnzELMAkGA1UEBhMCQlIxGTAXBgNVBAgMEERpc3RyaX
3  ↪ RvIEZlZGVyYWwxETAPBgNVBAcMCEJyYXNpbGlhMR8wHQYDVQQKDBZMdWl6J3MgR3JhZCBQcm9qZWN0IENBMR8w
4  ↪ HQYDVQQDDBZMdWl6J3MgR3JhZCBQcm9qZWN0IENBMSAwHgYJKoZIhvcNAQkBFhFlbWFpbEBsdWl6LmVuZy5icj
5  ↪ AeFw0xNzA2MDQyMzA4MTRaFw0xODA2MDQyMzA4MTRaMH8xCzAJBgNVBAYTAkJSMRkwFwYDVQQIDBBEaXN0cml0
6  ↪ byBGZWRlcmFsMRkwFwYDVQQKDBBBQ01FIENvcnBvcmF0aW9uMRkwFwYDVQQDDBBBQ01FIENvcnBvcmF0aW9uMR
7  ↪ 8wHQYJKoZIhvcNAQkBFhBhY21lQGx1aXouZW5nLmJyMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
8  ↪ 1WNoxOxIt5b4tqdjEdcmTaKEMwe1T4WwZuOwrg/DHfs3dUfMRl00pWgUUGzIXRTGGkq7hfmKTiJgOMuHQIhPEZ
9  ↪ hEp55OzaHFHdKo3rrdkrlaph7c0DLrDMZ8hmU5gPrrC4TC3lOQGEUbx1T5oM3jrRwFta/XqMTwVkuewmAkSA3O
10 ↪ VUEMa1f8Jbtfk37wp5Ya4ITmewMfA6Zxz2a4wsVRg2nCTR54kVuEoy4RIfn12yuFIHJw1IcaUHsQM1ESj+X8Zz
11 ↪ E1n5TuXfMP1jzWMruRp40ABTy4C4FdkRatYb9Um90iMPsrd3eSOPOw4G5k8cFk7r0Si4N98HEM26p908206QID
12 ↪ AQABo4GLMIGIMAkGA1UdEwQCMAAwOwYJYIZIAYb4QgENBC4WLEx1aXoncyBHcmFkIFByb2plY3QgQ0EgR2VuZX
13 ↪ JhdGVkIENlcnRpZmljYXRlMB0GA1UdDgQWBBTO5gLyPwpxswg/+eD2ar2REJCMKTAfBgNVHSMEGDAWgBQQsSAK
14 ↪ TjKPPyHTefWeqswR5MyYmDANBgkqhkiG9w0BAQsFAAOCAQEAMO/QhIBP1HoBW5ks9e/x6HVAN/X7GHfWUYU4is
15 ↪ QCEe7J6at+cmlLXd1/biOC9HnxzRlwnKjqe4TK8/OCac/6/zS7EgwZoiBNhX8vLPe4ne3uCpcGDi66f5kQEfSA
16 ↪ uWehDyHN+WIFXQED5C6u94FU3q6+qMT7ZEgGwh2H9nkCFKNNi1yV2pwf1hB1kVDbGjuEsYVv9o3ea8uQp1xQGb
17 ↪ IT83ZIdKRUyVHhBkfpko7idXazvHODUQL9JEEqKO4ZuzitFjzWhhJz8x7QdBu8EZYGZ5ZS+tohI6KBrhQZaFYo
18 ↪ RYJvATqighecO29V216npGYREqyeqm2jqPl36GyA+84d8Q==
```

Listing 4: ASCII representation of certificate under 1.acme.luiz.eng.br zone [Source: Author]

```
1  Certificate:
2      Data:
3          Version: 3 (0x2)
4          Serial Number: 1 (0x1)
5      Signature Algorithm: sha256WithRSAEncryption
6          Issuer: C = BR, ST = Distrito Federal, L = Brasilia, O = Luiz's Grad Project CA, CN =
               ↪  Luiz's Grad Project CA, emailAddress = email@luiz.eng.br
7          Validity
8              Not Before: Jun  4 23:08:14 2017 GMT
9              Not After : Jun  4 23:08:14 2018 GMT
10         Subject: C = BR, ST = Distrito Federal, O = ACME Corporation, CN = ACME Corporation,
               ↪ emailAddress = acme@luiz.eng.br
11         Subject Public Key Info:
12             Public Key Algorithm: rsaEncryption
13                 Public-Key: (2048 bit)
14                 Modulus:
15                     00:d5:63:68:c4:ec:48:b7:96:f8:b6:a7:63:11:d7:
16                     26:4d:a2:84:33:07:b5:4f:85:b0:66:e3:b0:ae:0f:
17                     c3:1d:fb:37:75:47:cc:46:5d:34:a5:68:14:50:6c:
18                     c8:5d:14:c6:1a:4a:bb:85:f9:8a:4e:22:60:38:cb:
19                     87:40:88:4f:11:98:44:a7:9e:4e:cd:a1:c5:1d:d2:
20                     a8:de:ba:dd:92:b9:5a:a6:1e:dc:d0:32:eb:0c:c6:
21                     7c:86:65:39:80:fa:eb:0b:84:c2:de:53:90:18:45:
22                     1b:c7:54:f9:a0:cd:e3:ad:1c:05:b5:af:d7:a8:c4:
23                     f0:56:4b:9e:c2:60:24:48:0d:ce:55:41:0c:6b:57:
24                     fc:25:bb:5f:93:7e:f0:a7:96:1a:e0:84:e6:7b:03:
25                     1f:03:a6:71:cf:66:b8:c2:c5:51:83:69:c2:4d:1e:
26                     78:91:5b:84:a3:2e:11:21:f9:f5:db:2b:85:20:72:
27                     70:d4:87:1a:50:7b:10:33:51:12:8f:e5:fc:67:31:
28                     35:9f:94:ee:5d:f3:0f:d6:3c:d6:32:bb:91:a7:8d:
29                     00:05:3c:b8:0b:81:5d:91:16:ad:61:bf:54:9b:dd:
```

```
                    22:30:fb:2b:77:77:92:38:f3:b0:e0:6e:64:f1:c1:
                    64:ee:bd:12:8b:83:7d:f0:71:0c:db:aa:7d:d3:cd:
                    b4:e9
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                Luiz's Grad Project CA Generated Certificate
            X509v3 Subject Key Identifier:
                CE:E6:02:F2:3F:0A:71:B3:08:3F:F9:E0:F6:6A:BD:91:10:90:8C:29
            X509v3 Authority Key Identifier:
                keyid:10:B1:20:0A:4E:32:8F:3F:21:D3:79:F5:9E:AA:CC:11:E4:CC:98:98

    Signature Algorithm: sha256WithRSAEncryption
         30:ef:d0:84:80:4f:d4:7a:01:5b:99:2c:f5:ef:f1:e8:75:40:
         37:f5:fb:18:77:d6:51:85:38:8a:c4:02:11:ee:c9:e9:ab:7e:
         72:69:4b:5d:dd:7f:6e:23:82:f4:79:f1:cd:19:70:9c:a8:ea:
         7b:84:ca:f3:f3:82:69:cf:fa:ff:34:bb:12:0c:19:a2:20:4d:
         85:7f:2f:2c:f7:b8:9d:ed:ee:0a:97:06:0e:2e:ba:7f:99:10:
         11:f4:80:b9:67:a1:0f:21:cd:f9:62:05:5d:01:03:e4:2e:ae:
         f7:81:54:de:ae:be:a8:c4:fb:64:48:06:c2:1d:87:f6:79:02:
         14:a3:4d:8b:5c:95:da:9c:1f:d6:10:75:91:50:db:1a:3b:84:
         b1:85:6f:f6:8d:de:6b:cb:90:a7:5c:50:19:b2:13:f3:76:48:
         74:a4:54:c9:51:e1:06:47:e9:92:8e:e2:75:76:b3:bc:73:83:
         51:02:fd:24:41:2a:28:ee:19:bb:38:ad:16:3c:d6:86:12:73:
         f3:1e:d0:74:1b:bc:11:96:06:67:96:52:fa:da:21:23:a2:81:
         ae:14:19:68:56:28:45:82:6f:01:3a:a2:82:17:9c:3b:6f:55:
         db:5e:a7:a4:66:11:12:ac:9e:aa:6d:a3:a8:f9:77:e8:6c:80:
         fb:ce:1d:f1
```

I.3 SIGNING A MESSAGE THROUGH THE API

To obtain a digitally signed QR Code, a user issues a POST request to the */sign* endpoint of the API, with the JSON message contained in the request body. Listing 5 shows an example curl request.

Listing 5: Curl request to sign a message [Source: Author]

```
curl "https://tcc.luiz.eng.br/api/sign" -H "Accept: image/png" -H "Content-Type: application/
    ↪ json; charset=UTF-8" --data '{"_id": 5, "_root": "acme", "Name": "Michael M. Motorist"
    ↪ , "DOB": "08/31/1978", "ID": "123 456 789", "Class": "D", "Issued": "08/31/2013", "
    ↪ Expires": "08/31/2021"}' > qrcode.png
```

The command produces an image file called qrcode.png that can be then verified using the bot by sending a message containing "/start" to the Telegram account @validator_bot and following the instructions given by the bot. When it asks for a picture, just send the qrcode.png file and the bot will then answer with the validation result.