Veritaa: Signing Transactions on Arduino

Bachelor Thesis

Serdil Mordeniz

University of Bern
Faculty of Science
B Sc in Computer Science

September 2020

Supervisor: Jakob Schaerer

Head of Research: Prof. Dr. Torsten Braun

UNIVERSITÄT
BERN

University of Bern

Institute of Computer Science

Bachelor of Science in Computer Science

Veritaa: Signing Transactions with Arduino

by Serdil Mordeniz

Abstract

The Internet of Things (IoT) is among the largest sources of data today and in the near future. Millions of sensors calculate and provide different services with all sorts of physical parameters. Securing data in a fast and secure way becomes increasingly more important due to the sheer amount of sensitive data that must be secured. Today data is mostly secured centrally and in a tedious and slow fashion. The discovery of Veritaa, a Distributed Ledger Technology (DLT) with an integrated public key signature store, is counteracting against this fact. The main objective of this work is to discover the compatibility between the Veritaa framework and the Arduino device.. Veritaa consists of a blockchain and the Graph of Trust (GoT). The GoT certifies the authenticity of the smart device, and the blockchain guarantees that the measurement values are immutable. The relations in the GoT are created with transactions, which can be a signing node or an edge. Subsequent to this, it is worth knowing the speed and energy consumption while creating and sending statements to other nodes in the network. In this work we discovered that an Arduino uno device does not have the needed flash memory for all the necessary libraries and global variables. With the slightly advanced version, the Arduino mega, the framework has been implemented successfully. It is possible to generate 23 edges and a signing node in a statement without a memory overflow. However, the maximum bytes to send to the hologram server without splitting the statement is 1024KB. That is why it is possible to only send a signing node and 5 edges which are 991 bytes as a JSON message. Sending and generating this JSON message to the hologram cloud is 9130ms and the sending alone takes approximately 3000ms. These values are justified with a simple time calculation. Furthermore, the average current consumption was 0.1191A at 5V which was determined with an USB meter.

Table of Contents

List of Figures

List of Tables

1 Introduction

Securing data with strong Lightweight Cryptographic (LWC) algorithms on resource constrained devices on the Internet of Things (IoT) is a long outstanding research topic. According to a study conducted in 2020 many kinds of research continue moving forward to find a suitable algorithm that meet the specific demands of the IoT application. The paper [1] provides an overview of the Lightweight Cryptographic (LWC) primitives for IoT environments and presents various LWC algorithms based on their key dimension, block size, structures, and number of rounds. With the upcoming of what is normally known as the “Fourth industrial Revolution” [2], the gathering of data has significantly changed. A huge number of sensors will measure all sort of physical parameters and provide them to different applications and services. These sensor readings are usually centrally controlled and collected.

With the discovery of Distributed Ledger Technology (DLT) data collection is being decentralized. The term DLT describes a technique used to document certain transactions. In contrast to the classic approach, in which a general ledger is usually managed by only one instance, any number of copies of the ledger, which are in principle equal, are maintained decentrally by different nodes. Appropriate measures are taken to ensure that new transactions to be added are adopted in all copies of the ledger and that an agreement (consensus) is reached on the current status of the ledger.

The term blockchain is closely coupled to the term DLT. The term blockchain is also used when an accounting system is managed decentrally and the correct status must be documented because many participants are involved in the accounting process. The procedure of cryptographic chaining in a decentrally managed accounting system is the technical basis for crypto currencies but can also contribute to improving or simplifying transaction security in distributed systems compared to central systems. One of the first applications of Blockchain is the crypto currency Bitcoin [3].

Veritaa is based on a high performance and scalable DLT a Distributed Public Key Infrastructure and Signature Store (DPKISS). DPKISS integrates an immutable database to immutably store declarations that have been signed by key pairs managed by Veritaa. The major invention of Veritaa is the Graph of Trust (GoT), a directed graph that uses relations between identity claims to certify the identities and store signed relations to digital document identifiers [4]. Veritaa provides a framework for signing measured physical parameters. These measured values are signed by a sensor node and the signature is immutably stored on a blockchain. That way, the integrity of data can be guaranteed, and sensor nodes can be authenticated.

At present most of the packets are moved unsecured or exported manually by people. The integrity of these measurement values and the authenticity of their source cannot be validated. But in many healthcare, governmental or industrial applications the validity of the source of data and the integrity of the data itself is extremely important. To enable these features, it is required that measurement values cannot be changed after they have been measured and that the authenticity of their source can be validated.

International Data Corporation (IDC) made an estimate that there will be 41.6 billion IoT devices connected, which will be generating 79.4 zettabytes (ZB) of data in 2025 [5]. One of the most relevant components of IoT are sensors that can measure sensitive and critical data like temperature values or personal data of a patient. Therefore, it is essential to secure the sensitive measurement values with efficient algorithms. While the security aspect is of tremendous importance, efficiency cannot be neglected because of the sheer amount of data that has to be secured. The market for sensors is there. The IoT sensor market was valued at USD 11.91 billion in 2019 and is expected to reach USD 42.67 billion by 2025. A Compound Annual Growth Rate (CAGR) of 24.05% during the forecast period of 2020-2025 is registered. There is a significant increase in the trend of automation. As sensors play the most critical role in every aspect of automation, IoT sensor market is expected to grow significantly in the near future [6]. There is evidence that IoT is further expanding even if the security is not ready yet. The baseline security must be robust, and the security architecture must be designed for long life cycles in the system, which is a huge challenge. Therefore, it is worth researching in this area.

Implementing the standard cryptographic algorithms is very difficult for resource-limited devices due to the size of implementation, the speed or performance, and energy consumption. Lightweight cryptography can use less memory and less computation. The main challenge of securing measurement values with cryptographic functions is that these microcontroller units (MCUs) have limited processing capabilities. It is extremely challenging to implement security when there are limited battery capacities, limited flash memory, and limited Random-Access Memory (RAM).

In this work we investigate Arduino development boards' hashing and signing capabilities and their compatibility with the Veritaa framework. Therefore, we want the Arduino sensor to be operated in a delegate mode closely coupled with a Veritaa node. For that we use the Hologram Cloud, which provides an API and a Dashboard interface to send TCP or UDP messages to any port on a device like Arduino. The sensor node creates statements that are collected and posted by a Veritaa node in a block to the blockchain. Since each sensor node creates its own chain of hashed transactions, even if this closely coupled mode does not write directly to the blockchain, the sensor readings are secured and cannot be deleted or changed because of the consensus among the nodes. The Graph of Trust is to certify the authenticity and the blockchain ensures the immutability of the measured values.

The main contributions of this thesis are summarized as follows:

- Implementing the Veritaa framework and creating Transactions/Statements to find out the compatibility of the Arduino boards with the Veritaa framework
- Connecting the Arduino sensor with the hologram cloud
- Examining the speed and energy consumption of generating and sending them to the Veritaa node

- Finding out the maximum amount of transactions the Arduino board can generate and send

2 Theoretical Background and Related Work

To implement the Veritaa framework we need a signature algorithm to sign a statement for data integrity and authentication. In this chapter, we will give the theoretical background information about the Veritaa framework and the signature algorithms. Since an objective of this thesis is to efficiently protect measurement values, the efficiency of these algorithms is at the forefront of the focus. Finally, the most relevant and well-established research on related works is being summarized and evaluated. In addition, the related work is being compared to Veritaa in terms of speed and energy consumption.

2.1 Veirtaa – The Graph of Trust and the ABCG

Veritaa [4] uses the Graph of Trust (GoT) and an Acyclic Block Confirmation Graph (ABCG) to enable authenticity of the entity and integrity of data. The GoT is used to represent real world relations between entities and digital documents. In the ABCG each block does not confirm a single, but multiple blocks like it was proposed in IOTA [7].

The GoT is a method in Veritaa to determine authenticity of an identity claim. A public key associated with the name of an entity to which it belongs is called an identity claim. Each identity claim may announce its relationship to other identity claims created by other entities. Together with the relations, the identity claims form the GoT and this GoT can be used to derive an identity claim 's authenticity. The relations are formed at the beginning with an identity claim and are ending with either an identity claim or a document identifier. A document identifier is a node consisting of an identifier that uniquely identifies a digital document and asserts its integrity. Each relationship has a type that indicates the relationship type between the identity claims and or the document identifier. In the next chapter, the exact relationships that are possible are described (see section *3.3*). The type of relations that can be formed between two identity claims are trusts audits and validates. Issues, approves and signs are relations that can be formed between an identity claim and a document identifier.

The reason why there are these types of relations is because actions that are performed on the document do not change the document and the actions performed between identity claims are transparent and traceable for all the members in the Veritaa network. Actions are relations between the nodes of the GoT. The actions are non-repudiable and immutable. This is done with the ABCG. The ABCG ensures the integrity of the GoT. To store the GoT in a peer-to-peer network, Veritaa uses the ABCG as a DLT. The ABCG is a particular DLT program optimized to store the GoT. The benefit of ABCG over a single blockchain is that new blocks do not always have to be posted at the end of the ABCG, so new blocks can always be committed by the nodes. If the honest nodes agree to only validate three blocks with the least validation locally, all blocks can be validated over time.

A validation is done by inserting the hash of the confirmed block in the verifying block header. As the hashes of the verified blocks are stored in the hash of the block, it is assured that it is not possible to modify the previous blocks. If a block in the ABCG changes, the hash will also change, and the successive blocks will then point to a non-existing block. A hash tree is used in the blocks to ensure the immutability of the transactions. Each transaction includes a hash of the previous transaction. The hash of the last transaction is stored in the hash of the block and, thus, if a single transaction were modified, the hash of the block would change. This hash tree makes it difficult to alter transactions.

2.2 Digital Signature Algorithms

A digital signature is an asymmetric cryptosystem in which a sender uses a secret signature key (the private key) to calculate a value for a digital message. This value enables anyone to use the public verification key to verify the undeniable authenticity and integrity of the message. To be able to assign a signature created with a signature key to a signatory, the corresponding verification key must be assigned to the verifier.

The data to be signed and the private key are calculating the signature by a unique calculation rule. Different data must almost certainly lead to a different signature, and the signature must produce a different value for each key. In deterministic digital signature procedures, the digital signature is uniquely defined by the message and the key. In probabilistic digital signature procedures, random values are included in the signature calculation, so that the digital signature for a message and a key can have many different values.

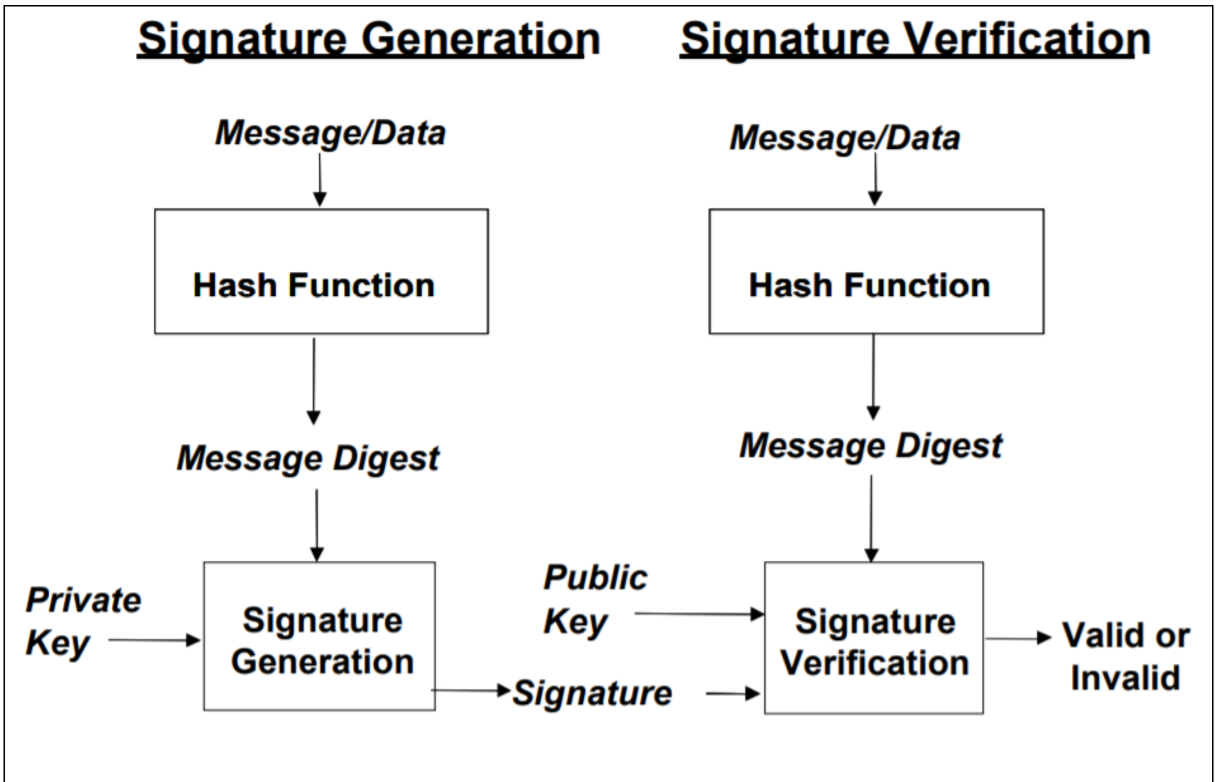

Figure 2.1: Digital Signature Process *[8]*

A digital signature algorithm involves a process for producing signatures, and a process for verifying signatures. A signatory uses the generation process to generate a digital signature on the data. A verifier uses the verification process to verify the signature 's authenticity. A signatory has a private and public key and owns the key pair. As shown in *Figure 2.1* the private key is used in the method of producing signatures. The only person allowed to use the private key to produce digital signatures is the key pair owner.

In a digital signature, the private key is not usually applied directly to the message, but to its hash value, which is calculated from the message using a hash function (such as SHA-3). To prevent attacks, this hash function must be collision-resistant, that is, it must be practically impossible to find two different messages with identical hash values.

If the public key has been assigned to a person by means of a digital certificate, the identity of the signature creator can be determined or verified via the public directory of the certification service provider, due to the fact that there is only one private key corresponding to the public key. The entirety of the technical infrastructure with which the certificates and information about their validity are generated and made publicly available is called PKI (Public Key Infrastructure).

A widespread misunderstanding is that signing is an encryption with the private key of an asymmetric encryption method. This assumption results from the fact that this is indeed the case with a naive and insecure variant of RSA, namely "Textbook RSA". However, this is never the case with secure variants of RSA (e.g. RSA-Full-Domain-Hash (FDH), RSA-Probabilistic-Signature-Scheme (PSS), RSA-Optimal-Asymmetric-Encryption-Padding (OAEP), for more details see [9]), despite certain similarities in details. With other encryption and signature procedures, there are usually only very superficial similarities at most [8]. There is no semantic security for Textbook RSA, so it is not secure against selected plaintext attacks or ciphertext attacks. This is because it is deterministic (the encryption of the same message creates the same ciphertext twice) and multiplicatively homomorphic (the encryption of encrypted values can be changed multiplicatively) which is not the case with the secure variants of RSA.

2.2.1 RSA

A group at M.I.T. discovered a good public key algorithm. It is known by the initials of the three discoverers (Rivest, Shamir and Adleman): RSA. For more than 30 years it has survived all attempts to break it and is considered very strong. A lot of protection today is based on it. For this reason, the 2002 ACM Turing Award was awarded to Rivest, Shamir, and Adleman. Its major disadvantage is that for good security it requires keys of at least 1024 bits, which makes it quite slow. The RSA method is based on certain Number Theory principles [10].

Algorithm 1: RSA Key Generation		
1	:	Choose two distinct prime numbers p and q.
2	:	Compute $n = p * q$.
3	:	Compute λ(n), where λ is Carmichael's totient function. Since $n = p * q, \lambda(n) = lcm(\lambda(p), \lambda(q))$, and since p and q are prime, $\lambda(p) = \varphi(p) = p - 1$ and likewise $\lambda(q) = q - 1$. Hence λ(n) = least common multiple of (p − 1, q − 1).
4	:	Choose an integer e such that 1 < e < λ(n) and $gcd(e, \lambda(n)) = 1$; that is, e and λ(n) are coprime.
5	:	Determine d as $d \equiv e - 1 \ (mod\ \lambda(n))$; that is, d is the modular multiplicative inverse of e modulo λ(n).

The public key is the modulus n and the exponent e. The private key consists of the private exponent d, which must be kept secret. P, q, and λ(n) also need to be kept secret as they can be used to calculate d. They can all be discarded after d is calculated [11].

The integers p and q should be selected at random for security purposes and should be comparable but differ in length by a few digits to make factoring more difficult. Prime integers can be found efficiently using a primality test.

For public and private keys, n is used as a modulus. The key length is its length, generally expressed in bits. N is released in the public key. The least common multiple (lcm) can be calculated with the Euclidean algorithm because $lcm(a, b) = \frac{|ab|}{gcd(a,b)}$.

Algorithm 2: RSA Signature Generation [12]		
Input	:	Message to be signed (msg)
Output	:	Signature (s)
1	:	Compute the message hash: $h = hash(msg)$
2	:	Compute h with the private exponent d to calculate the signature: $s = h^d \ (mod\ n)$

The hash h should be in the range [0...n). The obtained signature s is an integer in the range [0...n). Security relies on the fact that there are no efficient algorithms to decompose the prime numbers p and q.

Algorithm 3: RSA Signature Verification [12]

```
Input   :  Signature (s)
Output  :  True or false
1          Calculate the message hash: h = hash(msg)
2       :  Obtain A's authentic public key (n, e).
3       :  Compute h' with the public exponent e: h' $= s^e \ (mod\ n)$
4       :  Compare the resulting hash value h' with the message's actual hash value h
```

Signed messages guarantee data integrity and authenticity, provided the private key has really been kept secret. Due to the homomorphic nature of RSA, only one message can be signed with this method. If there are two signatures msg1 and msg2, an attacker can multiply them to calculate the message signature msg1*msg2. This problem can be circumvented by not signing the message itself. Instead, a collision resistant hash function h specified in addition to the signature procedure is used to calculate the hash value h(msg) of the message msg. This is signed with the private key to get the actual signature. The recipient can verify the signature thus obtained with the public key and receives a value h'. He compares this value with the hash value h(msg) of the message he received. If both values $h(msg) = h'$ match, it can be assumed with high probability that the message was transmitted without errors and is not fake. However, even this modification does not meet modern security requirements, so procedures such as RSA-Probabilistic-Signature-Scheme (PSS) are used to sign with RSA [8].

2.2.2 Efficient RSA Signature algorithm

In practice the parameters used in the RSA public-key scheme tend to be very long (1024-4096 bit). A major undesirable consequence of this is that typically, RSA computations are slow unless special measures are taken.

There are several ways to improve the efficiency of RSA:

1. Chinese Remainder Theorem (CRT) [13]
2. Square and multiply method [14]
3. Short exponents [15]
4. Batch RSA [16]
5. Multi-Prime RSA [17]
6. Multi-Power RSA [18]
7. Offline RSA-Key Generation [19]

The CRT is a method that accelerates RSA decryption/RSA signature generation by a factor of 4 [13]. Using the CRT, messages can be decrypted or signed more efficiently. Because the modulus N is very large, the bit representations of numbers used in the computer are also very long. The Chinese remainder theorem allows you to perform calculations in the two smaller groups of size p and q instead of in a group of size N, and to reassemble the result afterwards. Since the numbers here are much smaller, this calculation is faster overall.

Further improvement can be achieved with the square and multiply method. Since the main operation of the signature algorithm is modular exponentiation, it is worth making it more efficient. For the simple and slow exponentiation of x^k, (k-1) multiplications are needed. For the square and multiply method, the loop is only run through $\log_2 k$ times. In each loop, there is a squaring (the first squaring can be neglected) and possibly a multiplication. Asymptotically become $O(log(k))$ operations are required, whereas $O(k)$ operations are required for simple exponentiation. O denotes an asymptotic upper bound for the runtime behavior of the algorithm. As it can easily be seen, binary exponentiation is much more efficient than the simple procedure. This reduced demand on computing power is enormous for large bases and exponents [14]. The next evolution of the square and multiply method is an improved exponentiation algorithm for RSA [20]. In that paper a new exponentiation algorithm is presented that works in parallel, requires fewer multiplications and therefore has less delay. This technique is, therefore, more useful for larger key computations.

Furthermore, there is a method that includes the public key e to accelerate the RSA verification algorithm by making e small. It turns out, that you can choose the public key e to be a very small value. The three values $e = 3$, $e = 17$ and $e = 2^{16} + 1 = 65'537$ are the most common keys selected today. RSA is still secure with such short public exponents. In general the private exponent d still has the full bit length l. Otherwise it would be easy to brute-force the private key, i.e. reveal d by exhaustive searching [15].

The principle of batching is performing multiple simultaneous encryption or signature operation. In practice, the new variant performs several modular exponentiations effectively, at the cost of a single modular exponentiation. It leads to a very fast RSA-like scheme if RSA is to be performed at any central location or when pure-RSA encryption is to be performed [16].

Some other technique is the so called Multi-Prime RSA. Multi-Prime RSA is an RSA variant, where the modulus is the product of more than two distinct primes. The standard PKCS#1 is supported by having more than two prime factors. This is called the key to "multi-prime." On the plus side this could offer some improvement in performance. On the negative side the modulus may be weakened by using too small factors. The bottom line is that three or four primes will be tolerable for normal sizes and give a nice boost in efficiency, but it is not recommended to go beyond that. Public-key activities are not in any way affected by it [17].

Multi-Power RSA is a variation of Multi-Prime RSA. In Multi-Prime RSA, the N module consists of different primes, while in Multi-Power RSA, the $N = p^k * q$. This standardized modulus provides a more efficient decryption than Multi-Prime RSA. Takagi first defined Multi-Power RSA in 1998 [18].

Another speedup technique of RSA is the offline RSA-key generation. Since the RSA key generation takes a substantial amount of computations of the overall RSA algorithm, the keys could be generated offline and later when the RSA is used carried out to a database. One such example is from 2012 from Sami A. Nagar and Saad Alshamma where they aim to speed up the implementation of the RSA algorithm during data transmission between various communication networks and the Internet, which is determined to produce the keys by a program prepared in a C# language and then save those key values in the databases generated by SQL Server [19].

2.2.3 Ed25519

Edwards-curve Digital Signature Algorithm (EdDSA) is a modern and stable digital signature algorithm based on performance-optimized elliptic curves, such as the 255-bit Curve25519 curve. EdDSA signatures use the elliptic curves from Edwards (for performance reasons) edwards25519. The EdDSA algorithm is based on the algorithm for the Schnorr signature and relies on the problem of the elliptic curve discrete logarithm problem (ECDLP). The RFC 8032 theoretically defines the EdDSA signature algorithm and its variants. A 256-bit ECDSA signature has the same protection power as the RSA signature of 3072-bit [21].

Some RSA-type systems provide faster verification. But this advantage decreases as the level of security increases, and much slower signatures and much bigger keys outweigh the advantage for many applications. For low cycle functions higher performance is achieved. For example, rwb0fuz1024 (1024-bit Rabin – Williams) uses 12304 cycles to sign but 1751284 cycles and 128 bytes for a public key; ronald1024 (1024-bit RSA) uses 60300 cycles to sign but 2171124 cycles and 128 bytes for a public key; ronald3072 (3072-bit RSA) uses 231536 cycles to verify but an astounding 31456912 cycles to sign and 384 bytes for a public key. In the original paper that introduced Ed25519 they used 134000 cycles for verification, 87548 cycles for signing, and 32 bytes for a public key [22].

2.2.3.1 Ed25519 Key Generation

The private key is generated from a random integer, known as the seed, which like the curve order, should have similar bit length. The seed is hashed first, then the last few bits are removed, corresponding to the curve cofactor (8 for Ed25519), then the highest bit is removed, and the second highest bit is set. These transformations ensure that the private key will always belong to the same elliptic curve (EC) points subgroup on the curve, and that the private keys will always have the same bit length.

We assume the elliptic curve for the EdDSA algorithm comes with a generator point **G** and a subgroup order ***q*** for the EC points, generated from **G**.

The Public key pubKey is a point on the elliptic curve, determined by multiplication of EC points: pubKey = privKey * G (private key, multiplied for curve by generator point G). The public key is encoded as a compressed EC point: the y-coordinate, together with the x-coordinate 's lowest bit (the parity). The public key for Ed25519 is 32 Bytes.

2.2.3.2 EdDSA Sign

The EdDSA signing algorithm (RFC 8032) takes as its input a text message msg + private key privKey from the signer EdDSA and generates a pair of integers {R, s} as the output. Signing the EdDSA works as follows (with slight simplifications):

Algorithm 4: EdDSA Sign [21]

```
1  :  Calculate pubKey = privKey * G
2  :  Deterministically generate a secret integer
      r = hash (hash(privKey) + msg) mod q (this is a bit simplified)
3  :  Calculate the public key point behind r by multiplying it by the curve generator: R
      = r * G
4  :  Calculate h = hash (R + pubKey + msg) mod q
5  :  Calculate s = (r + h * privKey) mod q
6  :  Return the signature {R, s}
```

The digital signature created for Ed25519 is 64 bytes (32 + 32 bytes). It contains a compact point R and an integer s.

2.2.3.3 EdDSA Verify Signature

The EdDSA signature verification algorithm [23] takes a text message msg, the EdDSA public key pubKey, the EdDSA signature {R, s} as an input and generates a Boolean value (valid or false signature) as an output. The EdDSA verification algorithm (with slight simplifications) works as follows:

Algorithm 5: EdDSA Verify Signature [21]

```
1 : Calculate $h$ = hash ($R$ + $pubKey$ + $msg$) mod $q$
2 : Calculate $P1$ = $s$ * $G$
3 : Calculate $P2$ = $R$ + $h$ * $pubKey$
4 : Return $P1$ == $P2$
```

2.3 Related Work

In this section, we are going to compare and explain several existing solutions on security and privacy in IoT devices. The solutions are either block-based, which have their origin from blockchain, or DAG-based security systems, which have their origin from the tangle. Our focus is on Veritaa since the framework has been implemented in this work on an Arduino device. Veritaa uses a DAG-based system.

2.3.1 Overview of Block-based and DAG-based Security Systems

A blockchain [3] is a centralized and tamper-resistant network which is not owned by anyone but can be accessed by anyone. New blocks may be attached to current blocks if those in the network accept the new block. Also, it is not feasible to modify or erase blocks once they are recorded. The blockchains were designed to operate with adversarial agents on an insecure network. It ensures data integrity by preventing data erasure or abuse by using sophisticated and compute-intensive stable hash algorithms to capture data modification. Such compute-intensive algorithms are part of the proof of work which is a consensus process by different nodes in a network. The consensus can agree on new data or identify a modification in the network. Through using cryptographic algorithms Blockchain solves the privacy problems of IoT networks. By using tamper-resistant ledgers, it also addresses the security problems in IoT networks. *Figure 2.2* shows an example of a block-based security system. When we are talking about block-based security systems we mean that an underlying entity in the system can only confirm one block.

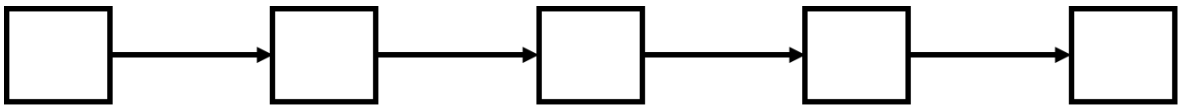

Figure 2.2: Block-based security system

Tangle [7] on the other hand is a newer technology introduced by crypto-currency IOTA for distributed ledgers. Inevitably, the tangle passes the blockchain as their next evolutionary step, it requires no complex, time-consuming and computer-intensive consensus protocol. It does not use blocks for processing transactions, either. Every transaction is by itself a separate block and will allow two older transactions to be added to the ledger. Two older transactions are accepted using proof of work. Tangle uses the DAG (see *Figure 2.3*) which connects each transaction to two older transactions by accepting it. When we are talking about tangle-based systems, we mean that an underlying entity in the system can confirm more than one block uses a DAG.

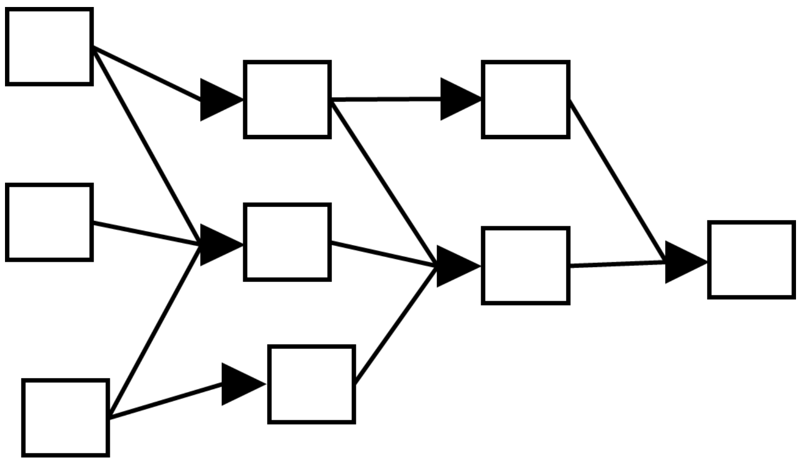

Figure 2.3: DAG-based security system

The current study of the literature shows that these two types of DLT's are used most often for resource constrained IoT networks. Table *2.1* gives an overview about the existing and most relevant DLT's security and privacy solutions on IoT devices.

Table 2.1: Overview of existing solutions in IoT devices

Existing solutions on security and privacy in IoT devices	Block-based	DAG-based	Others
Towards an Optimized BlockChain for IoT [24]	X		
Blockchain based Data Integrity Service Framework for IoT data [25]	X		
BIFF: A Blockchain-based IoT Forensics Framework with Identity Privacy [26]	X		
TangoChain: A Lightweight Distributed Ledger for Internet of Things Devices in Smart Cities [27]		X	
DIoTA: Decentralized-Ledger-Based Framework for Data Authenticity Protection in IoT Systems [28]		X	
IOTA-VPKI: a DLT-based and Resource Efficient Vehicular Public Key Infrastructure [29]		X	
A Blockchain Solution based on Directed Acyclic Graph for IoT Data Security using IoTA Tangle [30]		X	
A Hypergraph-Based Blockchain Model and Application in Internet of Things-Enabled Smart Homes [31]	X		X
Veritaa - The Graph of Trust [4]		X	

2.3.2 Block-based security systems

IoT security is difficult because most devices have low resource capacities, system heterogeneity and lack of standardization. In addition, many of these IoT devices gather and exchange vast volumes of data from our personal spaces, thus opening questions about privacy that are important. The papers [24], [25], [26] all introduced a new way of BC-based security and privacy for IoT devices.

The paper [24] proposes distributed trust to reduce the processing time for block validation. The authors of the paper [24] tested their approach in a smart home setting, but their work can also be used for broader IoT applications for providing security and privacy. Simulations showed that their approach substantially reduces packet and overhead processing time compared to the BC implementation in Bitcoin. Their focus was on the performance and the overhead rather than the energy consumption. Ultimately, their approach decreases the processing time by approximately 50%. Their work can be used for the mutual trust approach outlined in that paper may also be used in other BC-based systems if the network security allows it.

Another example of a BC-based system is the paper [25], where they are proposing a data integrity service based blockchain system. In that system, a more robust assurance of data integrity can be given to both Data Owners and Data Users, without relying on any third-party auditors. However, their implementation has still low efficiency as they are also aware of that. They have only implemented the fundamental functions of their protocol and want to further improve their approach in future.

Lastly there is BIFF a Blockchain-based IoT Forensics Framework with Identity Privacy which was published in 2018 [26]. In that paper, they propose a permitted IoT forensics system centered on blockchain to enhance the integrity, authenticity, and non-repudiation properties of the collected proof. They formally define the system architecture, provide details of the framework, and propose a cryptographic-based approach to mitigate privacy concerns about identity. They use a different type of consensus protocol called Byzantine Fault Tolerance (BFT), which is typically used in a permissioned blockchain. The system selects one "master" from the specified entities for each pre-defined epoch (e.g., servers under LEA control). This leader then gathers the unconfirmed transactions, forms a block, and integrates his ID into the field of the miner ID. This specific block is then transmitted and checked by the Group to the entire network. If a predefined threshold is passed by the number of positive testing, this particular block is considered valid and written in the immutable ledger. Their framework is different, but their transaction format is similar to Veritaa's. However, they did not test their framework yet. One potential future work that they announced is to integrate the system into an IoT testbed comprising a heterogeneous set of devices, to test the functionality of the system and to benchmark the efficiency [26]. For comparison, in Veritaa, consensus is achieved if only valid blocks are added to the ABCG. A block is considered valid if its hash tree contains only matching hashes, if it confirms at least three legitimate and no invalid blocks, if it is signed by an identity claim that occurs either in the GoT or in the same block, and if it contains a valid transaction list [4].

2.3.3 DAG-based security systems

The literature about Tangle-based (see section 2.3.1) security framework for IoT devices was a lot less common than the BC-based systems because tangle is relatively new. Nevertheless, the papers [28], [27], [30] all introduces a DAG-based security system.

For example, there is DIoTA [28] a novel, decentralized, ledger based IoT system authentication platform. To enable IoT devices and data protection, DIoTA uses a two-layer, decentralized ledger architecture along with a lightweight data authentication mechanism. They also evaluate DIoTA 's efficiency and protection. If we compare DIoTA with Veritaa, Veritaa is faster than DIoTA. It is also more secure because DIoTA uses global ledger nodes to monitor and track updates of each edge ledger to facilitate information exchange between different edge ledgers and prevent data modification. On the other hand, in Veritaa there are no global edges. This means that there is no centralized point for adversaries.

Compared with conventional linear blockchain, TangoChain [27] is a DAG. In TangoChain, each DAG node contains a single transaction created by an IoT node that finds two additional transactions already attached to TangoChain to verify their validity. To publish a transaction, there is a small proof of work to prevent denial of service attacks. Furthermore, they plan on doing experiments with realistic use cases to show TangoChain 's performance on a testbed and a network setup [27].

Another solution, which is not for conventional IoT devices but for car to car communication, is IOTA-VPKI, which stands for IOTA vehicular public key infrastructure. The efficacy of DLT-based VPKI will be assessed in the IoT project sponsored by EU Horizon 2020.

The research proposed in 2020 in paper [30] focuses on the use of IOTA Tangle's Masked Authenticated Messaging (MAM) function to ensure that IOT sensor data is transmitted and that guarantees the reliability and confidentiality of the data being transferred. MAM is a secure protocol to transfer and access encrypted or masked data, consisting of messages transmitted by zero value transactions, to the Tangle. Using the MAM module, nodes or devices connected to the IOTA Tangle can transmit their messages to a "Channel" in a masked and authenticated form. A Raspberry Pi was used to send and publish messages. After the node is connected to the IOTA network and collects the sensor data at predefined intervals, this collected sensor data is then released to the IOTA Tangle using the MAM functionality. It is time-stamped after the node collects the sensor data and then an encrypted message is generated. This encrypted message is being added to the Tangle. This provides confidentiality, integrity and authentication of the data that has been transferred. One weakness of this research is that they have not evaluated the performance of the proposed solution yet. They noted that further research work is needed in the direction for enhancing the transaction rate, to handle the high rate of IOT data more smoothly and also the development of different robust consensus mechanism is needed [30].

The paper [31] proposes a hypergraph-based blockchain model. This model aims to reduce storage consumption and solve the additional security problems. The author of paper [31] use the hyperedge as the storage node structure and turn the entire networked data storage into a part time network storage. They discuss the model and security strategy design in detail, introduce some use cases in a smart home network and evaluate the model's storage performance through simulation, experiments, and network assessment.

From the study of the literature of related works about BC- or DAG-based systems Veritaa is on the cutting edge in terms of speed and efficiency. All the papers show that they did not do a performance test yet or Veritaa is more effective.

3 Veritaa Framework

The purpose of Veritaa is to verify the origin, integrity, and history of digital documents. The GoT represents real world relations between entities and use this information to certify the identities. In order to create relations and build up the GoT, Veritaa provides a signing node and an edge for IoT devices like Arduino. The signing node is necessary to initially connect to the network. Edges represent declarations and actions that have been performed and signed by entities. Together they form a statement. A statement usually consists of a signing node and several edges. Since our main goal is to find out if the Arduino devices are capable to be secured by the Veritaa framework, we must explain how the framework is built up in detail. For that reason, we are going to introduce the statement structure. We will provide some context information about Veritaa to understand about what the individual fields are about. For a more detailed explanation of Veritaa, the original paper [4] which proposed Veritaa can be used.

3.1 Statement Structure

A statement consists of several fields (see *Table 3.1*). These fields all have a specific size. The transaction_counter counts the number of transactions in the block for a Veritaa node to understand how many transactions were sent. The transaction field is a list of a signing node and several edges. How many transactions a statement consists of is up to the creator that is also the reason this field can be dynamic in length. The creator is a unique hash value. The signature length depends on the signature algorithm. We are using the most efficient solution which is Ed25519 with a signature length of 64 bytes.

Table 3.1: Veritaa: Statement

Field	Description	Size [Bytes]
transaction_counter	number of transactions in the block	4
transactions	list of the transactions	dynamic
creator	hash of the entity that created the statement	32
signature_length		2
signature		signature_length

3.2 Transaction: Signing Node

The Graph of Trust (GoT) is built out of transactions that contain relations between identity claims or an identity claim and a document identifier. The GoT provides two methods to prove the authenticity of identity claims. The first method is domain vetting. Domain vetting is the procedure on doing background checks with certain methods to identify its trustworthiness. Each identity claim can have a validation domain to a well-known folder that contains a file with the public keys that are owned by the domain holder. This can be achieved with the last field of the signing node which is the validation link. The first four fields are the same in every transaction. The type field gives the form of a transaction which provides either domain vetting or if the GoT itself is being used to prove the authenticity of identity claims. For domain vetting we use the signing node which is identified by a 0 in the type field (see *Table 3.2*).

One of the most important fields in a transaction is the previous transaction field. When you initially create a transaction that field is set to null. The second transaction contains the hash of the previous transaction and so on. This leads to the last transaction's previous transaction field containing all the other hash values from the previous transactions. If the transaction was changed or altered in the process a different hash value will be calculated. That way the Veritaa nodes can validate the integrity of the transactions. The inception and expiry date. We use Unix timestamps. The field algorithm must be added to the framework, so that validating node knows which verification algorithm should be used. The signature algorithms that we have tested are P-521, Ed25519 and RSA. In our experiments Ed25519 was more convenient and faster than the other algorithms.

Table 3.2: Transaction: Signing Node

Field	Description	Size [Bytes]
transaction_length	length of the transaction	2
previous_transaction	hash of the previous transaction or null for the first	32
inception_date		8
expiry_date		8
type	**Signing Node = 0** or Edge = 1	1
node_type	Organization = 0	1
object_hash	hash (type + node type + name + public key + validation link)	32
name_length		2
name	name of the node	name_length
algorithm	Either P-521, Ed25519 or RSA	1
public_key_length		2
public_key	public key of the identity claim	public key length
validation_link_length		2
validation_link	validation link for vetting	validation link length

3.3 Transaction: Edge

The second way to establish evidence of an identity claim 's validity is through the GoT itself. The identity claims along with the relationships form the GoT, and this GoT is being used to deduce an identity claim 's validity. In order to make these relations with another node in the network or with a document identifier, the field edge_type is being introduced (see *Table 3.3*). There are 8 possible edge types possible. That is why one byte is being reserved.

Table 3.3: Transaction: Edge

Field	**Description**	**Size [Bytes]**
transaction_length	length of the transaction	2
previous_transaction	hash of the previous transaction or null for the first	32
inception_date		8
expiry_date		8
type	Signing Node = 0 or **Edge = 1**	1
edge_type	0 = ISSUES 1 = VALIDATES 2 = AUDITS 3 = REVOKES 4 = APPROVES 5 = TRUSTS 6 = REQ_SUBSIGNING 7 = GRANT_SUBSIGNING	1
object_hash	the hash of the node that is endorsed by this edge	32
payload_length		2
payload		payload length

3.4 Subsigning Entity

After the statements are generated, they are sent to a signing entity by a subsigning entity. The subsigning entity requests the signing and the signing entity approves it. A pairing is achieved when the signing entity grants the subsigning rights. When this paring is done, the subsigning entity can send statements to the signing entity and the signing entity then puts the statement in a block and commits it to the ABCG. In *Figure 3.1*, A1 is the subsigning entity and A2 represents the signing entity. After A1 requests subsigning rights, A2 can grant it. Now A1 is able to issue data in the network. The sensor node was paired with the owner and therefore, all entities that trust the owner's identity can also trust the values of A1. A1 is in this work our Arduino device, but it could be any other device that gathers some type of sensitive data and A2 is a node that is already trusted by the network.

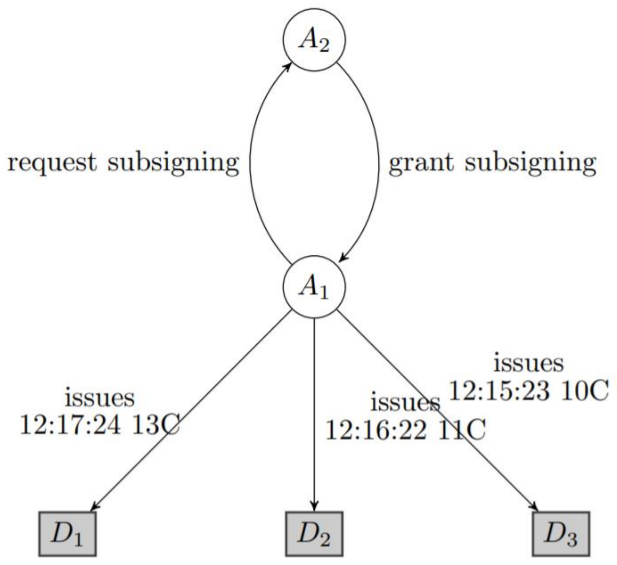

Figure 3.1: Subsigning Entity

4 System Implementation

To send data to other nodes in the network on an Arduino device is only possible with a compatible shield. For that we use the SparkFun LTE CAT M1/NB-IoT shield. For the implementation in the next chapter we need to know how the components work together and how limited their resources are. In *Figure 4.1* (see below) you can see our system architecture to send statements to a Veritaa node. This architecture is only built to test the speed and energy consumption to generate and send statements on Arduino. We are not using real measurement values, but instead we take any temperature value and a timestamp. Hologram is an IoT connectivity platform. They are partnered with all big networks in the U.S and 550 carriers worldwide and their global IoT SIM card provides an optimal coverage. With hologram, companies avoid the headache of negotiating contracts with carriers while ensuring access to every available cellular network and technology. We use the hologram cloud to simulate our tests. In this chapter we are going to specify and explain all the necessary steps and the system components to send data from an Arduino device to the hologram cloud which forwards it to the Veritaa node.

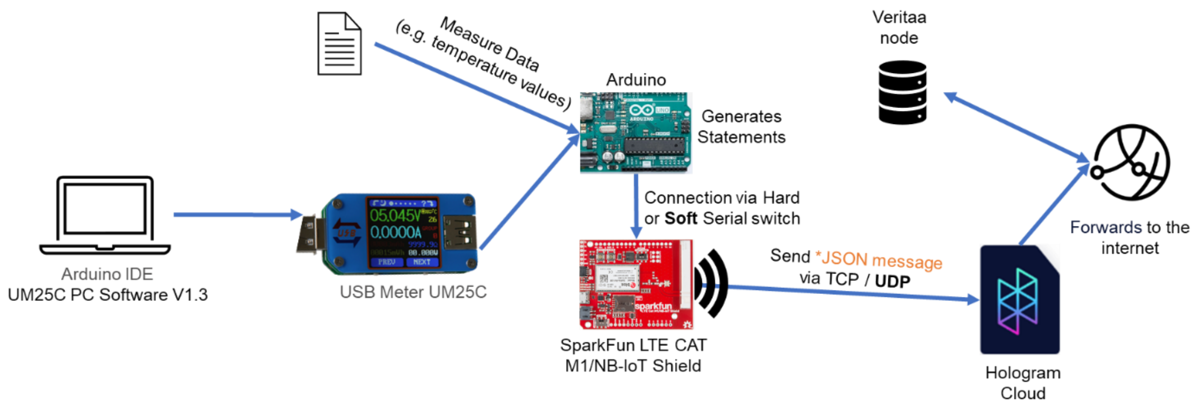

Figure 4.1: Overview of the system architecture

4.1 Device Connections

An USB meter is used to measure the current flow. UM25C PC software records voltage and current values in a 0.55 seconds interval. The Sparkfun LTE CAT shield receives power from the Arduino's 5V supply pin. The data communication between the Arduino and the shield occurs via an AT command interface over a simple UART RX and TX pins. The serial switch can either be set to HARD for the pins 0/1 and SOFT for the pins 8/9. We use the SOFT mode.

The Hologram Socket API provides a low-level TCP socket interface to connect with the Hologram Cloud to the shield.

4.2 Sending Statements to the Hologram Cloud

To register to the hologram cloud, we use a hologram SIM card which must be activated. Afterwards we have access to the internet via hologram. Before you run the registration sketch for the connection with the mobile network you need to adjust the mobile network provider to default because it is set to an American provider. After 30 seconds the shield finds the available mobile networks in your area and you can select one. After a few seconds, you should see the status on your device's Hologram dashboard to be activated.

To send a hologram message a device key is necessary which you can generate on the hologram dashboard. The generated device key must be updated in the sending sketch. Furthermore, the pins must be changed to 8/9 because we use SOFT mode. Finally, you can send a JSON-encoded string with socketWrite. The JSON-encoded string includes the following fields:

- devicekey (k) -- String. Eight-character Device Key used for authentication
- data (d) -- String. Base64-encoded statement
- tags (t) - String or array of strings (not used)

4.3 Hardware limitations of each component

First, we tried to implement the Veritaa framework with the Arduino uno. However, the Arduino uno has not enough flash memory to implement the source code and all the necessary libraries. Ed25519, SHA3, Base64 and SparkFun library were the necessary libraries. Our sketch took more than 41KB of storage space. below you can see the output from the Arduino IDE.

Table 4.1: Output from the Arduino IDE

```
Sketch uses 41312 bytes (128%) of program storage space. Maximum is 32256 bytes.
```

The reason we tried another board was because the Arduino uno did not have enough flash memory for all the necessary libraries (see Table *4.1*). We chose a board with more flash memory which is the Arduino mega. To compare the used Arduino boards, we give an overview about the most important hardware components. It is also necessary to know some hardware restrictions for the implementation.

4.3.1 Arduino uno rev3

First, there is the flash memory of 32KB, which saves the program that is loaded from the Arduino IDE. There is also a library called progmem. With that library you can store constant data on the flash to not waste any other memory (like the valuable SRAM). The SRAM has 2KB. Then there is a CPU which controls everything that goes on within the device. Finally, there is the Electrically Erasable Programmable Read Only Memory (EEPROM) of only 1KB (see Table *4.2*).

Table 4.2: Arduino UNO ATmega328P hardware specification

Microcontroller	ATmega328P
Operating Voltage	5V
Flash Memory	32 KB of which 0.5 KB used by bootloader
SRAM	2 KB
EEPROM	1 KB
Clock Speed	16 MHz

4.3.2 Arduino mega 2560

In *Table 4.3* you can see the Arduino mega's 2560 hardware specification. Arduino mega is an upgrade in terms of memory space. The Operating Voltage and Clock speed is the same as in Arduino uno.

Table 4.3: Arduino MEGA 2560 hardware specification

Microcontroller	ATmega2560
Operating Voltage	5V
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

4.4 Interrupt problem with Serial Pin 8 on Arduino Mega and the solution

Standard bootloader of the Arduino uno/mega only supports hardware serial pins 0/1 for updating the program (Uploading Sketch). Therefore, hardware serial pins 0/1 is reserved for updating the program.

Software Serial pins 8/9 works fine with Arduino uno for Rx and Tx. But not all pins of the Arduino mega support interrupts, so only the following pins can be used for Rx:10, 11, 12, 13, 14, 15, 50, 51, 52, 53, A8 (62), A9 (63), A10 (64), A11 (65), A12 (66), A13 (67), A14 (68), A15 (69).

We used pin 10 for Rx. That is why a small change in the wiring had to be made. On Arduino mega the pin 10 had to be connected to Sparkfun LTE shield's pin 8. The Tx pin 9 works fine because no interrupt is necessary. *Figure 4.2* (see below) shows the wiring that was necessary on Arduino Mega to connect with the SparkFun LTE CAT M1 / NB-IoT Shield. The SparkFun LTE CAT M1 / NB-IoT Shield offers connectivity to data networks around the globe for an Arduino or Arduino-compatible microcontroller. LTE stands for Long Term Evolution, which is also known as 4G, a standard for wireless broadband communication for mobile devices and data terminals.

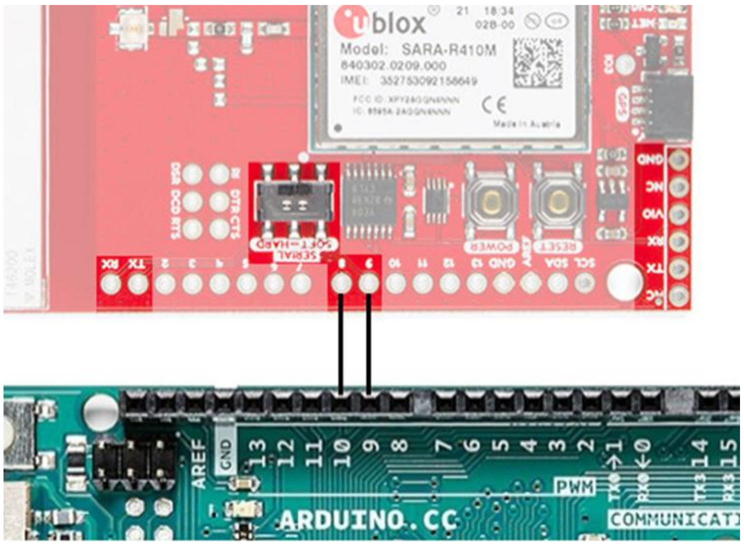

Figure 4.2: Wiring of SOFT serial pins Rx Tx

4.5 SparkFun LTE CAT M1/NB-IoT firmware update necessary

Cellular technologies unique to IoT are a modern and exciting branch of cellular technology. However, as with all emerging technologies, as the sector matures, there are some growing pains that must get tackled. Sending messages to the hologram cloud does not show up on the dashboard without a firmware update of the LTE shield. After the update of the Cat-M1 R410 Nova's Firmware [32], everything was working as expected. The installation process is only possible on a Raspberry Pi or Linux machine. Additionally, this firmware update will change the behavior of the blue LED on the LTE shield. Instead of slowly blinking while connected with the Hologram network, it will now stay solid.

5 Implementation of the Extension of the Veritaa Framework

In this chapter the implementation of the Veritaa framework on Arduino mega is explained. To not waste any valuable SRAM a lot of thought was spent to make the code as efficient as possible. Considering the statement structure (see chapter 3) is relatively complex the right variables must be saved at the right place.

5.1 Setup Requirements

For serial data transmission, we must set the data baud rate of 9600 to communicate with the serial monitor. Once the serial monitor is open the baud rate must be changed to the same one as in the sketch so that the communication happens.

After the connection with the LTE-Shield over a serial interface is established, the private and public key will be generated with Ed25519 and saved in global variables. Next, a hash of the creator with SHA3-256 will be generated and saved in a global variable too. Finally, transaction counter and the last hash of the object is initialized. All these variables are necessary for the Veritaa framework (see *Figure 5.1*).

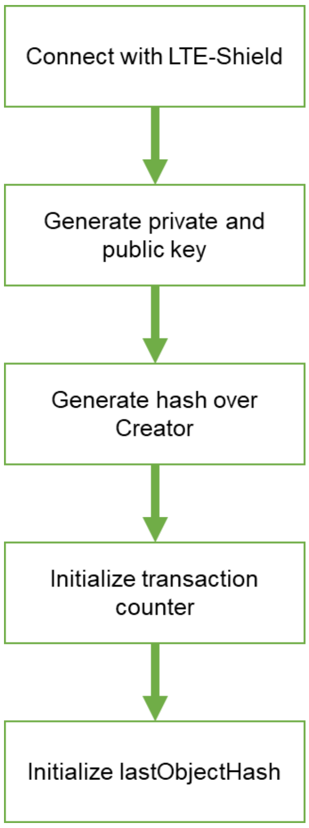
Connect with LTE-Shield
Generate private and public key
Generate hash over Creator
Initialize transaction counter
Initialize lastObjectHash

Figure 5.1: Steps needed in Setup to send statements

5.2 Flowchart Loop

After the setup, an initial signing node is generated and saved in a statement buffer (1. see Figure *5.2*). A hash is created from the following data and saved to a global variable "lastObjectHash":

- nodeName
- type
- nodeType
- publicKey
- validationLink

The global variable "lastObjectHash" is used when the next transaction is generated.

The data of the edge transaction is added to a buffer in the appropriate place (2. see Figure *5.2*). A temperature value (integer) and a timestamp (unsigned long) are taken as payload. A hash is created from the payload and stored on a global variable (lastObjectHash). The global variable "lastObjectHash" is used when the next transaction is generated like in the generation of the signing node. The generation of edges and the saving in the buffer will be repeated COUNT_EDGE-times.

Next, a hash of the last object hash and the hash of the creator is generated (3.) and signed (4.) with the private key. The signature length and this signature is added in the buffer in the right place. Therefore, the statement is created (5.) and lastly, we must encode the statement to base64 (6.) because it is one of the most common used encoding schemes and more efficient than hex encoded. There are some important plus points of hex encoding. It is easy to grasp and implement. Each byte is encoded as a separate character pair. To achieve a more effective encoding, Base64 uses a wider character set. It is not meant to be readable by humans in any way, but it is meant to be compatible with as many systems as possible. The algorithm is more complicated than hex encoding, but there is a 33% increase in the data size. This encoded message can now be sent as a hologram JSON message (7. & 8.). This loop will be repeated COUNT_STATEMENT-times (see *Figure 5.2*).

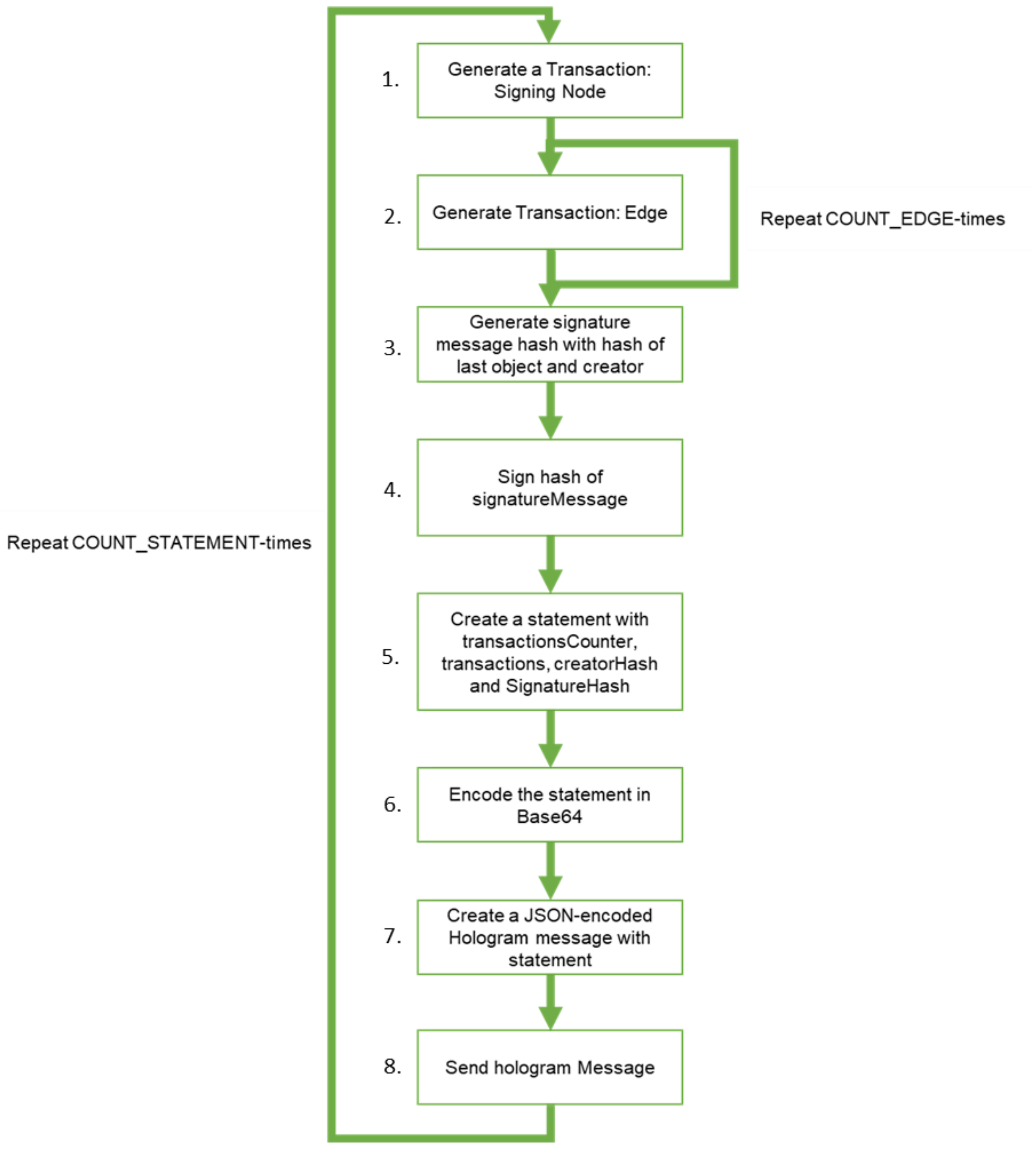
1.
Generate a Transaction: Signing Node
2.
Generate Transaction: Edge
Repeat COUNT_EDGE-times
3.
Generate signature message hash with hash of last object and creator
4.
Sign hash of signatureMessage
Repeat COUNT_STATEMENT-times
5.
Create a statement with transactionsCounter, transactions, creatorHash and SignatureHash
6.
Encode the statement in Base64
7.
Create a JSON-encoded Hologram message with statement
8.
Send hologram Message

Figure 5.2: Steps needed in Loop to send statements

6 Experiments and Results

In this section we are presenting our findings about the performance of the Veritaa framework and the energy consumption. With our first experiment we found the maximum amount of transactions in a statement to generate and send it to the hologram cloud. We determined the maximum amount by starting with an initial signing node and successively adding more edges to it until a SRAM overflow happened. In a second experiment we measured the time and energy consumption for sending statements for a Veritaa system coupled with an Arduino mega board.

6.1 Determination of a Statement Size

A statement was introduced in section *3.1*. To determine the size of a transaction for our measurements, the following values (red) were taken for the fields with variable lengths. The other values for Veritaa framework are fixed.

All the fields are fixed in size, except name, public key, and validation link for the signing node. For the edge only the payload field size is variable.

Table 6.1: Size of Transaction: Signing Node

Field	**Size [Bytes]**
transaction length	2
previous transaction	32
inception date	8
expiry date	8
type	1
node type	1
object hash	32
name length	2
name	5
algorithm	1
public key length	2
public key	32
validation link length	2
validation link	15
Total	**143**

Table 6.2: Size of Transaction: Edge

Field	**Size [Bytes]**
transaction length	2
previous transaction	32
inception date	8
expiry date	8
type	1
edge type	1
object hash	32
payload length	2
payload	10
Total	**96**

As one can see in *Table 6.1* and *Table 6.2* the size of a signing node is 143 bytes and an edge is 96 bytes in our example.

To determine how many edges can be generated on Arduino mega in a statement, the number of edges was increased step by step. The tests could be performed without errors up to 23 edges. Afterwards memory overflow occurred.

As shown in *Table 6.3*, the size of a statement with 23 edges and a signing node occupies 2453 bytes for the plain data and 3295 bytes for the JSON message in SRAM.

To encode the plain data to base64 we need to keep the plain data in memory. Together with the encoded JSON message the SRAM occupies (2453 bytes + 3295 bytes) 5748 bytes. The rest of the SRAM is used for the other functions and global variables.

Table 6.3: Maximum size of a statement with 23 edges and a signing node

Statement data	**Bytes**	**Quantity**	**Size (Bytes)**
Transaction counter	4	1	4
Signing node	143	1	143
Edge	96	**23**	2208
Hash of the Creator	32	1	32
Signature length	2	1	2
Signature	64	1	64
Total plain data			**2453**
Message (Base64-encoded)			3272
JSON-encoded	23	1	23
JSON-Message to send			**3295**

Our goal is to send the maximum amount of transaction in one execution. The first discovery is that the Arduino mega can generate 23 edges and a signing node in a statement without a memory overflow. This was found by incrementing the amount of edges. After further tests we discovered that more than 1024 bytes could not be sent to the hologram cloud. The reason why only 1024 bytes could be sent has to be further investigated. Our assumption is that this limitation is based on the hologram server JSON message limit or the maximum packet size of the narrowband-IoT shield.

Our second discovery was that Arduino mega could maximally send 5 edges and a signing node in one iteration of the loop which is in the scope of 1024 bytes that could be sent. To determine the maximum number of edges in a statement with a message size of 1024 bytes, the following calculation can be performed:

Table 6.4: Maximum size of a statement that we could send to the hologram cloud

Statement data	**Bytes**	**Quantity**	**Size (Bytes)**
Transaction counter	4	1	4
Signing node	143	1	143
Edge	96	**5**	490
Hash of the Creator	32	1	32
Signature length	2	1	2
Signature	64	1	64
Total plain data			**735**
Message (Base64-encoded)			968
JSON-encoded	23	1	23
JSON-Message to send			**991**

As you can see in *Table 6.4* the total plain data is 735 bytes and after it is base64-encoded 968 bytes are used. The JSON message needs 23 additional characters which must be added in the calculation. The maximum size of a JSON message to send is 991 bytes. If we would increment the amount of edges to 6, the JSON message to send would be above 1024 bytes and that is why only 5 edges with a signing node is possible to send in one execution.

6.2 Time Measurement of individual functions

First, the execution time of individual functions were measured by the program using the Arduino time function millis.

The example code in *Figure 6.1* outputs the number of milliseconds on the serial interface that the Arduino board needs to execute the function getEd25519Keys.

```
uint8_t privateKey[32];
uint8_t publicKey[32];

unsigned long time1, time2;

void setup(){
  Serial.begin(9600);

  time1 = millis();

  getEd25519Keys(privateKey, publicKey);

  time2 = millis();
  Serial.println(time2-time1);
}

void loop() {

}
```

Figure 6.1: An example code to measure execution time of getEd25519Keys

After 50 measurements, the following times per function were determined (see *Table 6.5*):

Table 6.5: Time measurement for each function

Function	**Time (ms)**	**Description**
LTE Shield connecting	902	Per setup First connection 20 seconds
getEd25519Keys	7202	Per setup
createCreatorHash	9	Per setup
signingnode2Buffer	9	Per loop
Edge2Buffer	9	Many times per loop
createHashForSignature	9	Per loop
sign_ED25519	6096	Per loop
Base64.encode statement (1 Signing Node + 5 Edge)	2	Plain Data: 725 Byte Base64.Encoded Data: 968 Byte
sendHologramMessage	see *Figure 6.2*	

The key generation, the signing, and the sending of the hologram message takes the most amount of time. However, the keys only must be generated once in the setup. The other values can be neglected. To send a statement to the hologram server as a JSON message of 991 bytes (see *Table 6.4*) it takes approximately 3000ms.

To calculate how long it would take to send the maximum amount of edges in a statement (23 edges and a signing node) we measured how long it takes to send messages to the hologram cloud. Sending only involves the sending of data. That means opening, connecting, and writing data to a socket. After successfully writing the data to the socket, it must be closed. *Figure 6.2* shows the sending time to send data in 100-byte intervals. The graph below can be extrapolated to the number of bytes 23 edges and a signing node need in order to determine the time to send the maximum amount of transactions in a statement.

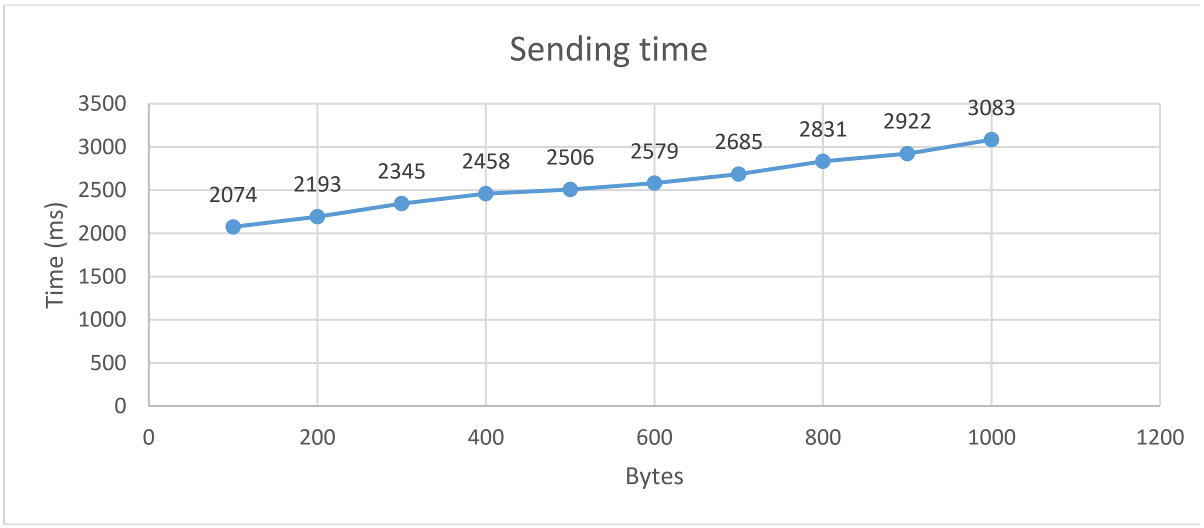

Figure 6.2: Time measurement to send hologram message in 100 byte intervals

Since we could not send the maximum amount of bytes in one execution because of the above explained reasons in chapter 6.1. To determine the time of the maximum size of a statement to send to the hologram server we can use linear extrapolation. Excel has a convenient function for this which is called TREND. This function extrapolates any number of values in a dataset. In *Figure 6.3* (see below) the statement size of 3295 bytes is extrapolated over the 10 measured values from *Figure 6.2*. Therefore, to send the maximum size of a signing node with 23 edges it would approximately take **5457ms**.

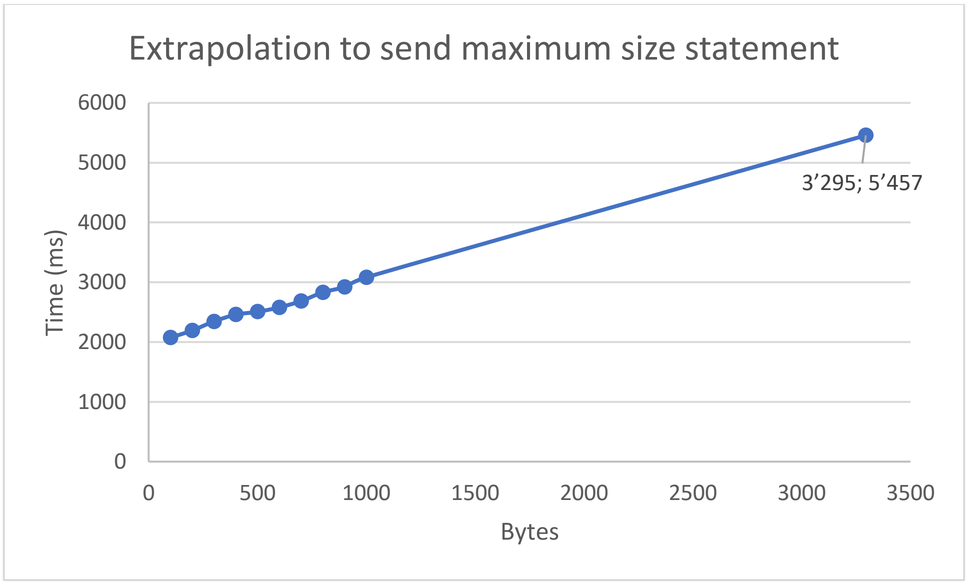

Figure 6.3: Extrapolation to send maximum size statement

6.3 Time Measurement and Energy consumption with USB-meter UM25C

We used a function in Arduino libraries that has a millis function. This function returns the number of milliseconds since the Arduino board started the current program. To verify the time measurements with the millis function, a measuring system with UM25C was set up (see *Figure 4.1*). With UM25C PC software V1.3 the time and the supply current and voltage of the connected device can be measured and recorded.

6.3.1 Current and time measurement

To increase the accuracy of the average value, the generation and sending of the statement to hologram servers was repeated 40 times and then the average value was determined (red).

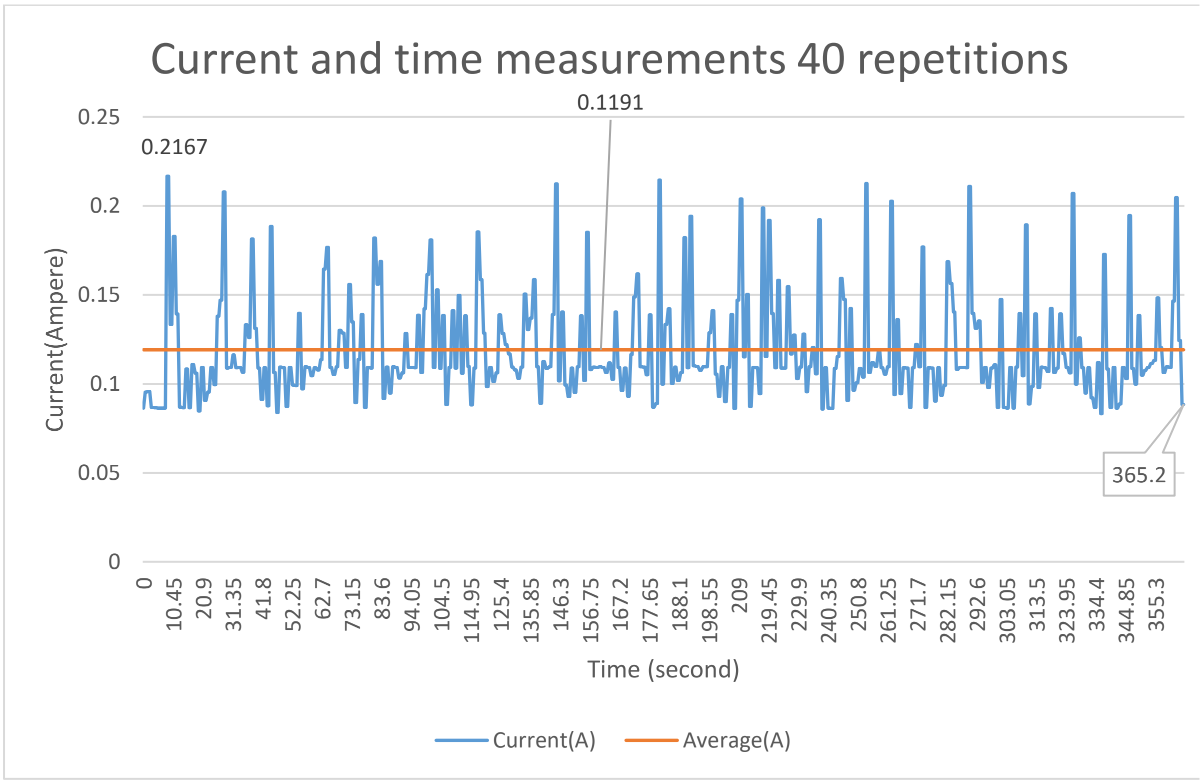

Figure 6.4: Current and time measurements 40 repetitions

These measurement values were recorded in a loop. The loop was repeating itself for 40 times. One-time execution of the loop can be summarized as follows:

- Generation of a statement with a signing node and 5 edges
- Signing the statement

- Base64 encoding of the statements
- Sending the JSON message to the hologram cloud

The supply voltage stays stable at 5V. In *Figure 6.4* you can see the x-axis which indicates the amount of seconds and on the y-axis the amount of current flow (in Ampere) can be seen. The first label (**0.2167**) in the diagram shows the peak current consumption in amperes to send a statement. The second label (**0.1191**) in the diagram shows the average value of the current consumption in Amperes when generating and sending 40 statements. The last label in the diagram shows the number of seconds (**365.2**) to send 40 statements. According to measurements, the generation and transmission of 40 statements takes 365.2 seconds. This means that the average value for the generation of a statement and transmission can be calculated as 9130ms (365200ms/40) (see *Table 6.6*).The calculated average power consumption is 0.5955W. One time execution of the loop takes therefore 5.437Ws.

Table 6.6: Key findings with USB-meter in loop

The maximum current consumption at 5V is: **0.2167A.**
The average current consumption at 5V is: **0.1191A.**
The average time is: **9130ms**
The average power consumption is: 5V x 0.1191A = **0.5955W**
The average energy consumption is: 0.5955W x 9.13 = **5.437Ws**

6.3.2 Time Calculation and comparison with the measurement data

In *Table 6.7* we compare the average time with the time determined by the millis function. We can see the time calculation for a signing node and 5 edges. The only thing that must be calculated is the execution time of transactions2Buffer because it is the only variable value. The other values were adopted from the time measurement of each function in *Table 6.5*. The average time for generating a statement and sending 9130ms is therefore pretty much the value determined with the millis function.

Table 6.7: Time calculation for 1 Signing Node and 5 Edges

Function	**Time (ms)**
transactions2Buffer für (1 Signing Node +5 Edge)	56
createHashForSignatur	9
sign_ED25519	6096
Base64.encode statement (1 Signing Node + 5 Edge)	2
sendHologramMessage	3000
Total	9163

6.3.3 Current Measurement in Setup

The data in *Figure 6.5* was recorded during setup. It performed the following activities:

- Connection with the LTE Shield
- Public and private key generation
- Generating the hash of the creator

Table 6.8: Key findings with USB-meter in setup

The average current consumption at 5V is: **0.0869A**
The average time is: **8250ms**
The average power consumption is: 5V x 0.0869A = **0.4345W**
The average energy consumption is: 0.4345W x 8.25 = **3.585Ws**

Interestingly, in the setup the average current consumption was only 0.0869A. The reason for that is because the LTE shield is in idle state and does not consume much power. All the activities in the setup endured 8.25 seconds. The average power consumption is 0.4345W and, therefore, the average energy consumption amounts to 3.585Ws (see *Table 6.8*).

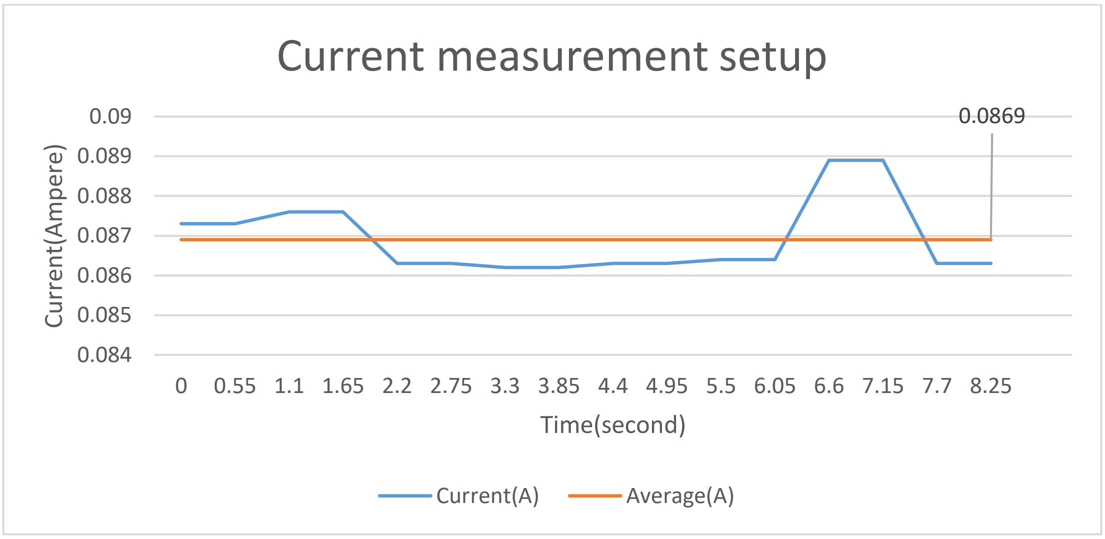

Figure 6.5: Current measurement during setup

6.3.4 Current Measurement while Idle

In *Figure 6.6* the current measurement was recorded while the system was idle. In the idle state Arduino is only connected to the LTE shield with no tasks to be completed. The idle state was measured to calculate the total amount of current consumption for a battery driven Arduino mega device in a real-world system in future. The average current consumption for the idle state is: **0.0857A**

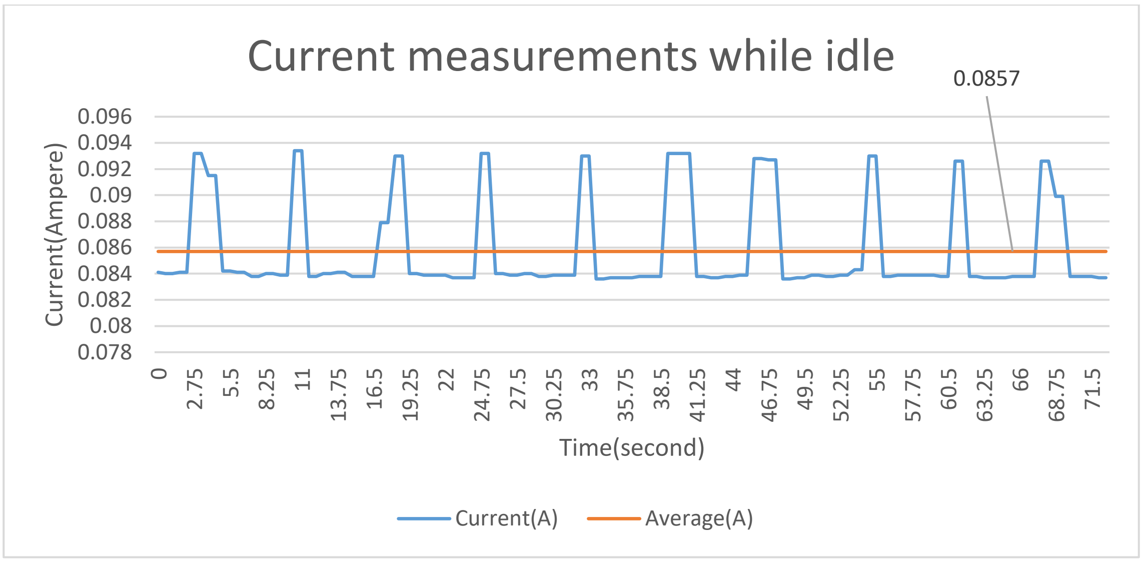

Figure 6.6: Current measurement while idle

7 Conclusion and Future Work

In this section we conclude our work and summarize our results. In addition, several issues that could be explored in future work are discussed to further improve the energy consumption.

7.1 Conclusion

In this thesis our main objective of implementing the Veritaa framework on an Arduino device was achieved after switching the Arduino uno with the more advanced device Arduino mega. The reason why Arduino uno was not compatible with the Veritaa framework is because the libraries Ed25519, SHA3, Base64 and SparkFun library use more flash memory than the Arduino uno board has, which is 32KB. Furthermore, Arduino devices have limited resources and standard RSA as a public key signature algorithm is not recommended to use even after improving the algorithm significantly with various methods. The solution is to use lightweight cryptographic algorithms like ed25519. Regarding to the Veritaa framework we discovered that we could only send a signing node and 5 edges as a statement. The reason is because there is a limit of 1024 bytes from the hologram cloud. Furthermore, the speed for the different functions that are needed in order to send statements to the hologram server were determined and the energy consumption was evaluated. The average energy consumption to send 1 statement takes 5.437Ws. The average energy consumption to generate a statement is only 3.585Ws due to the LTE shield being idle while Arduino generates a statement.

The contribution of this work has a significant importance to the IoT field. The reason for this fact is that in future when real systems are constructed with millions of nodes in the Veritaa network we can already make estimations about the energy consumption of the future system. Embedded systems are compatible with the Veritaa framework too since Arduino mega had all the necessary resources to create statements and send them. This work is the steppingstone to much bigger projects when real systems are going to become tested with real data.

Of course, there are also some weaknesses with the framework. By adding the hashes, the overhead for transmitting a sensor value is increased. But with security there is always going to be some inefficiencies and Veritaa is a very fast system overall. It is one of the best systems to secure embedded systems that need to have authenticity and integrity of the data that has been measured or gained from the outside world.

7.2 Future Work

We should keep in mind that Arduino boards have several sleep modes, which should be used whenever possible. In a microchip AVR, which is an 8-bit microcontroller, regulated network, there are many things to consider when trying to reduce the power consumption. Sleep modes should be used as much as possible, and the sleep mode should be chosen to perform as few of the functions of the system as possible. All unwanted features should be deactivated. In a future work these sleep modes can be used to further optimize the energy consumption [33].

A problem that occurred which could be solved if we had more time, is the limit of 1024 bytes to send to the hologram server. This problem can be further analyzed. An idea is to split the JSON message into several pieces and then reconstruct it at the receiver.

8 Arduino Sketch of the Veritaa framework

```
#include <Ed25519.h>
#include <SHA3.h>
#include <Base64.h>

// Click here to get the library:
// http://librarymanager/All#SparkFun_LTE_Shield_Arduino_Library
#include <SparkFun_LTE_Shield_Arduino_Library.h>

const uint8_t COUNT_EDGE = 5;
const uint8_t COUNT_STATEMENT = 40;

uint32_t transactionsCounter;

// key generation with Ed25519
uint8_t privateKey[32];
uint8_t publicKey[32];

uint8_t lastObjectHash[32];
uint8_t creatorHash[32];

typedef enum : uint8_t { Ed_25519, P_521, RSA } algorithms;

typedef enum : uint8_t {
  ISSUES, VALIDATES, AUDITS, REVOKES,
  APPROVES, TRUSTS, REQ_SUBSIGNING,
  GRANT_SUBSIGNING
} edge_types;

typedef struct Payload {
  int temperature;
  uint64_t timestamp;
} payload;

// Create a SoftwareSerial object to pass to the LTE_Shield library
SoftwareSerial lteSerial(10, 9);
// Create a LTE_Shield object to use throughout the sketch
LTE_Shield lte;

// Plug in your Hologram device key here:
String HOLOGRAM_DEVICE_KEY = "SS9peQQq";
//String HOLOGRAM_DEVICE_KEY = "Mdy,L0Ju";

// These values should remain the same:
const char HOLOGRAM_URL[] = "cloudsocket.hologram.io";
const unsigned int HOLOGRAM_PORT = 9999;


// generates the object hash from a signing node
// and returns it via parameter hash
void createObjectHashSigningNode(
  uint8_t* hash,
  uint16_t nodeNameLength,
  unsigned char* nodeName,
  uint8_t type,
  uint8_t nodeType,
  uint8_t publicKeyLength,
  uint8_t* publicKey,
  uint16_t validationLinkLength,
  unsigned char* validationLink)
{
  SHA3_256 sha3_256;
  sha3_256.reset();
  size_t dataLength = 1 + 1 + nodeNameLength
                           + publicKeyLength
                           + validationLinkLength;

  uint8_t data[dataLength];
  uint8_t start = 0;

  memcpy(&data[start], &(type), 1);
  start += 1;

  memcpy(&data[start], &(nodeType), 1);
  start += 1;
  memcpy(&data[start], nodeName, nodeNameLength);
  start += nodeNameLength;
  memcpy(&data[start], publicKey, publicKeyLength);
  start += publicKeyLength;
  memcpy(&data[start], validationLink, validationLinkLength);
  start += validationLinkLength;
  sha3_256.update(data, dataLength);
  sha3_256.finalize(hash, 32);
}


// copies the signing node data to buffer
size_t signingNode2Buffer(
  unsigned char* buffer,
  size_t start,
  uint8_t* previousTransaction,
  uint64_t inceptionDate,
  uint64_t expiryDate,
  uint16_t nodeNameLength,
  unsigned char* nodeName,
  algorithms algorithm,
  uint8_t publicKeyLength,
  uint8_t* publicKey,
  uint16_t validationLinkLength,
  unsigned char* validationLink)
{
  uint16_t transactionLength = 90 - sizeof(uint16_t)
                                  + nodeNameLength
                                  + publicKeyLength
                                  + validationLinkLength;

  memcpy(&buffer[start], &(transactionLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], previousTransaction, 32);
  start += 32;

  memcpy(&buffer[start], &(inceptionDate), sizeof(inceptionDate));
  start += sizeof(inceptionDate);
  memcpy(&buffer[start], &(expiryDate), sizeof(expiryDate));
  start += sizeof(expiryDate);

  uint8_t type = 0;
  memcpy(&buffer[start], &(type), sizeof(type));
  start += sizeof(type);

  uint8_t nodeType = 0;
  memcpy(&buffer[start], &(nodeType), sizeof(nodeType));
  start += sizeof(nodeType);

  uint8_t objectHash[32];
  createObjectHashSigningNode(objectHash,
                              nodeNameLength,
                              nodeName,
                              type,
                              nodeType,
                              publicKeyLength,
                              publicKey,
                              validationLinkLength,
                              validationLink);

  // Update lastObjectHash
  memcpy(lastObjectHash, objectHash, 32);

  memcpy(&buffer[start], objectHash, sizeof(objectHash));
  start += sizeof(objectHash);

  memcpy(&buffer[start], &(nodeNameLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], nodeName, nodeNameLength);
  start += nodeNameLength;

  memcpy(&buffer[start], &(algorithm), sizeof(algorithm));
  start += sizeof(algorithm);

  memcpy(&buffer[start], &(publicKeyLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], publicKey, publicKeyLength);
  start += publicKeyLength;

  memcpy(&buffer[start], &(validationLinkLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], validationLink, validationLinkLength);
  start += validationLinkLength;
  return start;
}

// generates the object hash from a edge and
// returns it via parameter hash
void createObjectHashEdge(uint8_t* hash,
  uint16_t payloadLength,
  uint8_t* payload)
{
  SHA3_256 sha3_256;
  sha3_256.reset();
  sha3_256.update(payload, payloadLength);
  sha3_256.finalize(hash, 32);
}


// generates the hash from a creator name and
// returns it via parameter hash
void createCreatorHash(uint8_t* hash,
  uint16_t creatorLength,
  uint8_t* creator)
{
  SHA3_256 sha3_256;
  sha3_256.reset();
  sha3_256.update(creator, creatorLength);
  sha3_256.finalize(hash, 32);
}


// copies the signing node data to buffer
size_t edge2Buffer(
  unsigned char* buffer,
  size_t start,
  uint8_t* previousTransaction,
  uint64_t inceptionDate,
  uint64_t expiryDate,
  edge_types edgeType,
  uint16_t payloadLength,
  uint8_t* payload)
{
  uint8_t  transactionLength = 86 - sizeof(uint16_t) + payloadLength;

  memcpy(&buffer[start], &(transactionLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], previousTransaction, 32);
  start += 32;

  memcpy(&buffer[start], &(inceptionDate), sizeof(inceptionDate));
  start += sizeof(inceptionDate);

  memcpy(&buffer[start], &(expiryDate), sizeof(expiryDate));
  start += sizeof(expiryDate);
  uint8_t type = 1;
  memcpy(&buffer[start], &(type), sizeof(type));
  start += sizeof(type);

  memcpy(&buffer[start], &(edgeType), sizeof(edgeType));
  start += sizeof(edgeType);

  uint8_t objectHash[32];

  createObjectHashEdge(objectHash,
                       payloadLength,
                       payload);

  // Update lastObjectHash
  memcpy(lastObjectHash, objectHash, 32);
  memcpy(&buffer[start], objectHash, sizeof(objectHash));
  start += sizeof(objectHash);

  memcpy(&buffer[start], &(payloadLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], payload, payloadLength);
  start += payloadLength;

  return start;
}


//Helper method for providing a transaction
size_t transactions2Buffer(unsigned char* buffer, size_t start) {

  payload payload = { -2, 1588180278 };
  uint8_t payloadData[sizeof(payload)];
  memcpy(payloadData, &payload, sizeof(payload));

  size_t lastPos = start;

  // Fill buffer with signing node
  lastPos = signingNode2Buffer(buffer,
                               lastPos,
                               lastObjectHash,
                               1588180278,
                               1609459200,
                               5, (unsigned char*)"Node1",
                               Ed_25519,
                               sizeof(privateKey), privateKey,
                               15, (unsigned char*)"validationlink1");


  for(size_t i = 0; i < COUNT_EDGE; i++)
  {
    // Fill buffer with edge
    lastPos = edge2Buffer(buffer,
                          lastPos,
                          lastObjectHash,
                          1588180278,
                          1609459200,
                          REQ_SUBSIGNING,
                          sizeof(payload),
                          payloadData);

  }

  return lastPos;
}


// Just needs the addresses to generate the private
// and derive the public key.
void getEd25519Keys(uint8_t* privateKey, uint8_t* publicKey) {
  Ed25519::generatePrivateKey(privateKey);
  Ed25519::derivePublicKey(publicKey, privateKey);
}
```

```
// Signs the message.
void sign_ED25519(uint8_t signature[64],
  const uint8_t  privateKey[32],
  const uint8_t publicKey[32],
  const void* hash, size_t len)
{
  Ed25519::sign(signature, privateKey, publicKey, hash, len);
}


// generates the hash from signature message and
// returns it via parameter hash
// signature message consists of the hash of the last object (edge)
// and the creator hash
void createHashForSignature(uint8_t* hash, uint8_t* signatureMessage)
{
  SHA3_256 sha3_256;
  sha3_256.reset();
  sha3_256.update(signatureMessage, 64);
  sha3_256.finalize(hash, 32);
}


void sendHologramMessage(String message)
{
  int socket = -1;
  String hologramMessage;

  // Construct a JSON-encoded Hologram message string:
  hologramMessage = "{\"k\":\"" + HOLOGRAM_DEVICE_KEY + "\",\"d\":\"" +
    message + "\"}";

  // Open a socket
  socket = lte.socketOpen(LTE_SHIELD_TCP);
  // On success, socketOpen will return a value between 0-5. On fail -1.
  if (socket >= 0) {
    // Use the socket to connect to the Hologram server
    //Serial.println("Connecting to socket: " + String(socket));
    if (lte.socketConnect(socket, HOLOGRAM_URL,
                          HOLOGRAM_PORT) == LTE_SHIELD_SUCCESS) {
      // Send our message to the server:
      if (lte.socketWrite(socket,
                          hologramMessage) == LTE_SHIELD_SUCCESS)
      {
        // On succesful write, close the socket.
        if (lte.socketClose(socket) == LTE_SHIELD_SUCCESS) {
          Serial.println("Socket " + String(socket) + " closed");
        }
      } else {
        Serial.println(F("Failed to write"));
      }
    }
  }
}



void setup()
{
  Serial.begin(9600);
  Serial.println(millis());

  // connection with LTE Shield
  if ( lte.begin(lteSerial, 9600) ) {
    Serial.println(F("LTE Shield connected!"));
  }

  // key generation with Ed25519
  getEd25519Keys(privateKey, publicKey);

  createCreatorHash(creatorHash, 8, (uint8_t*)"Creator1");

  // Initialize transaction counter
  transactionsCounter = 1;

  // Init lastObjectHash
  memset(lastObjectHash, 0, 32);

  Serial.println(millis());
}


void loop()
{
  // Measurements performed with COUNT_STATEMENT packages
  if (transactionsCounter > COUNT_STATEMENT)
  {
    Serial.println(millis());
    delay(10000);
    return;
  }

  // buffer for transactions
  unsigned char buffer[750];

  int start = 0;

  memcpy(&buffer[start], &(transactionsCounter), sizeof(uint32_t));
  start += sizeof(uint32_t);

  // generate a transaction with a signing node and several edges
  start = transactions2Buffer(buffer, start);

  memcpy(&buffer[start], creatorHash, 32);
  start += 32;

  // Signature over previous object hash (lastObjectHash)
  // and creator hash
  uint16_t signatureLength = 64;
  uint8_t signature[64];
  uint8_t signatureMessage[64];
  memcpy(signatureMessage, lastObjectHash, 32);
  memcpy(&signatureMessage[32], creatorHash, 32);
  uint8_t hash[32];
  createHashForSignature(hash, signatureMessage);
  sign_ED25519(signature, privateKey, publicKey, hash, 32);

  memcpy(&buffer[start], &(signatureLength), sizeof(uint16_t));
  start += sizeof(uint16_t);

  memcpy(&buffer[start], signature, signatureLength);
  start += signatureLength;

  // encode the binary data to Base64-string for sending
  int encodedLength = Base64.encodedLength(start);
  char message[encodedLength];
  Base64.encode(message, (char*)buffer, start);

  // Send the Message
  sendHologramMessage(message);
  lte.poll();
  Serial.println(start);
  Serial.println(encodedLength);
  transactionsCounter++;
}
```

Bibliography

[1] Z. Labbi, M. Senhadji, A. Maarof and M. Belkasmi, "Lightweight Cryptographic for Securing," *International Journal of Innovative Technology and Exploring Engineering (IJITEE),* pp. 181-188, 4 February 2020.

[2] M. Xu, J. M. David and S. H. Kim, "The Fourth Industrial Revolution: Opportunities and Challenges," *International Journal of Financial Research,* pp. 90-95, 8 March 2018.

[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[4] J. Schaerer, S. Zumbrunn and T. Braun, "Veritaa - The Graph of Trust".

[5] M. Shirer, "International Data Corporation," 18 June 2019. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS45213219.

[6] "Mordor Intelligence," [Online]. Available: https://www.mordorintelligence.com/industry-reports/iot-sensor-market.

[7] S. Popov, "The Tangle," 30 April 2018. [Online]. Available: https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf.

[8] "Digital Signature Standard (DSS)," in *FEDERAL INFORMATION PROCESSING STANDARDS*, Gaithersburg, 2013.

[9] E. K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2," RSA Laboratories, 2016.

[10] A. S. TANENBAUM and D. J. WETHERALL, COMPUTER NETWORKS, Amsterdam, The Netherlands; Seattle, WA: Pearson Education, 2010, p. 795.

[11] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," in *Massachusetts Institute of Technology*, Cambridge, 1979.

[12] A. Menezes, P. v. Oorschot and S. Vanstone, "Digital Signatures," in *Handbook of Applied Cryptography*, United States, CRC Press, 1996, pp. 425-481.

[13] C.-H. Wu, J.-H. Hong and C.-W. Wu, "RSA Cryptosystem Design Based on the Chinese Remainder Theorem," in *Asia and South Pacific Design Automation Conference*, Asia, 2001.

[14] L. C. K. Hui and K. -.. Lam, "Fast square-and-multiply exponentiation for RSA," Electronics Letters, 1994.

[15] P. C. Paar, *Implementation of Cryptographic Schemes 1,* Ruhr University Bochum: Chair for Embedded Security, 2015.

[16] A. Fiat, "Batch RSA," in *Journal of Cryptology*, Israel, 1996.

[17] M. J. Hinek, "On the security of multi-prime RSA," in *Journal of Mathematical Cryptography*, Canada, 2008.

[18] T. Takagi, "Fast RSA-type cryptosystem modulo," in *Annual International Cryptology Conference*, Japan, 1998.

[19] Alshamma, S. A. Nagar and Saad, "High Speed Implementation of RSA Algorithm with Modified Keys Exchange," in *6th International Conference on Sciences of Electronics*, Sudan, 2012.

[20] S. Sepahvandi, M. Hosseinzadeh and K. N. a. A.jalali, "An Improved Exponentiation Algorithm for RSA Cryptosystem," in *International Conference on Research Challenges in Computer Science*, 2009.

[21] S. Nakov, "Practical Cryptography for Developers," 2018. [Online]. Available: https://cryptobook.nakov.com/digital-signatures/eddsa-and-ed25519.

[22] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering,* p. 77–89, 2012.

[23] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," January 2017. [Online]. Available: https://tools.ietf.org/html/rfc8032#page-13.

[24] A. Dorri, S. S. Kanhere and R. Jurdak, "Towards an Optimized BlockChain for IoT," in *IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, Pittsburgh, 2017.

[25] B. Liu, X. L. Yu, S. Chen, X. Xu and L. Zhu, "Blockchain based Data Integrity Service Framework for IoT data," in *IEEE International Conference on Web Services (ICWS)*, Honolulu, 2017.

[26] D.-P. Le, H. Meng, L. Su, S. L. Yeo and V. Thing, "BIFF: A Blockchain-based IoT Forensics Framework with Identity Privacy," in *IEEE Region 10 Conference*, Korea (South), 2018.

[27] A. Tekeoglu and N. Ahmed, "TangoChain: A Lightweight Distributed Ledger for Internet of Things Devices in Smart Cities," in *IEEE International Smart Cities Conference (ISC2)*, Morocco, 2019.

[28] L. Xu, L. Chen, Z. Gao, X. Fan, T. Suh and W. Shi, "DIoTA: Decentralized-Ledger-Based Framework for Data Authenticity Protection in IoT Systems," *IEEE Network,* pp. 38-46, January/February 2020.

[29] A. Tesei, L. D. Mauro, M. Falcitelli, S. Noto and P. Pagano, "IOTA-VPKI: A DLT-Based and Resource Efficient Vehicular Public Key Infrastructure," in *IEEE 88th Vehicular Technology Conference (VTC-Fall)*, Chicago, 2018 .

[30] M. Bhandary, M. Parmar and D. Ambawade, "A Blockchain Solution based on Directed Acyclic Graph for IoT Data Security using IoTA Tangle," in *5th International Conference on Communication and Electronics Systems (ICCES)*, India, 2020.

[31] C. Qu, M. Tao and R. Yuan, "A Hypergraph-Based Blockchain Model and Application in Internet of Things-Enabled Smart Homes," *Sensors,* 24 August 2018.

[32] "Hologram," [Online]. Available: https://support.hologram.io/hc/en-us/articles/360035212594-Updating-the-Cat-M1-R410-Nova-s-Firmware. [Accessed 15 August 2020].

[33] "Datasheet Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V," 2014, pp. 50-54.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname:	Serdil Mordeniz
Matrikelnummer:	16-108-086
Studiengang:	Computer Science Bachelor
	Bachelor [✓] Master [ ] Dissertation [ ]
Titel der Arbeit:	Veritaa: Signing Transactions on Arduino
LeiterIn der Arbeit:	Prof. Dr. Torsten Braun

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

3123 Belp, 31.08.2020

Ort/Datum

S. Mordeniz

Unterschrift