European Institute of Technology - ICT Labs
University of Twente
Department of Computer Science

Master Thesis

Computer Science Master. Security and Privacy Major

Evaluation of HIMMO with Long Identifiers,

an Extension of the HIMMO Key Establishment Scheme

Student:
Cristian-Alexandru Staicu

University of Twente Supervisor:
dr. Andreas Peter

Industry Supervisor:
dr. Oscar-Garcia Morchon

August 2014

Abstract

Key establishment is one of the most important building blocks used in modern cryptography. Its main aim is to allow two or more parties to agree on a key in a secure way. Key predistribution schemes (KPS) are a subclass of solutions for key establishment that require a trusted third party (TTP) to distribute keys or key material to every node in the system upon node registration. This key material is further used to derive the actual key, using some additional information such as the identity of the other participant. Polynomial-based KPSs are an interesting type that are efficient in terms of computation, memory and bandwidth. The main limitation of such schemes is the low number of nodes that need to collude in order to derive additional keys or even the TTP's key material. Protecting against such attacks is traditionally done by increasing the degree of the used polynomial, impacting heavily the performance of the system. HIMMO key establishment scheme is overcoming this limitation, therefore achieving a good collusion resistance at low costs.

The initial version of the HIMMO scheme constrained the user to use the same length for the keys and for the identifiers of the nodes. This constraint has a negative impact on scalability and severely limits the number of scenarios. In this master thesis we analyze a proposed solution to overcome this limitation, namely *HIMMO with Long Identifiers* (HIMMO-LI). The main objective of our work was to adapt the existing optimizations proposed for HIMMO, to the case of HIMMO-LI. We wanted efficient algorithms that require minimum modifications and can be implemented in practice. Another aspect to focus on is the performance of lattice-based collusion attacks on HIMMO-LI as a result of the decoupling of the key length and identifiers length. At last, an additional problem that the current master thesis addresses is how to compare the performance and the security of HIMMO-LI with the other key establishment schemes.

We designed algorithms for adapting two existing optimizations to the long identifiers case: *More is Less* and *HIMMO Transformation* and we analyzed their theoretical performance. We implemented these algorithms in Java, C and ATmega assembly, and created an automated testing framework which we used for running thousands of experiments. First, we verified the theoretical expressions for the performance of the two algorithms. We also propose the use of a novel way of relating the dimension of a lattice used in a collusion attack to NIST security bits. Using this result and a few very simple protocols employing elliptic curves or identity-based cryptography, we compared the performance of HIMMO with other similar solutions, showing its main advantages and disadvantages.

The main result of our work is a more complete view on the actual performance of the HIMMO key establishment scheme, including a modification that makes the system more scalable. We also show that increasing the resistance against collusion attacks can be done in two ways in this key establishment scheme: by increasing either the degree of the used polynomial or the ratio between the identifier size and the key size. Another important finding is that it is more efficient to compute a pairwise key using multiple parallel HIMMO systems with smaller key size than one with a larger key. When compared with similar solutions, we found that HIMMO-LI can be efficiently used for lightweight devices, showing a better performance both in CPU time and RAM memory. However, this last result is heavily dependent on the validity of our way of relating the performance of lattice-based collusion attacks to NIST security bits.

Keywords: key predistribution, symmetric cryptography, polynomial scheme, lattice attacks.

Contents

Chapter 1

Introduction

1.1 Key Establishment

Key establishment is "any process whereby a shared secret key becomes available to two or more parties, for subsequent cryptographic use" [24]. Agreeing on a key is a mandatory step in every application that involves encryption, no matter if we are talking about symmetric or asymmetric encryption. The variety of such applications is immense, including mobile applications, corporate software, banking systems, wireless sensor networks or biomedical devices. This diversity led to the creation of hundreds of key establishment schemes (KESs), ranging from very simple, intuitive ones described and analyzed only in theory to the most complex ones that were successfully incorporated in the devices that surrounds us nowadays. The huge number of such protocols makes an exhaustive presentation impossible, so the current master thesis aims to present only a specific category of KESs. For a more complete, but a little out-dated survey on the existing KESs the reader is referred to [9]. In the remaining part of this section, we will highlight the main criteria used in the literature to differentiate between the available key establishment techniques, and present the desired properties of such constructions.

The first distinction of key establishment schemes proposed in the literature is a categorization depending on the way through which the participants gain access to the secret key. The first such category is represented by *key transport* schemes, where only one party creates the key and then delivers it securely to the other(s). Another way to perform the key establishment is to have each party generate a share of the key and transmit it to the others – this process is called *key agreement*, thus the shared key will be a function of all the generated shares. *Key predistribution* systems were introduced by Matsumoto and Imai in [23] and they require the distribution of some key material to all the participants before the actual key establishment starts. After this, every two or more participants can compute a shared key by using their key material and the identity of the other participant(s).

Another common differentiation is based on the number of parties that want to agree on a shared key. We speak about *two-party schemes* and *multi-party* or *conference schemes*. Traditionally, the two-party ones were more studied, since this scenario was the most frequently encountered in practice, but with the rise of ubiquitous computing, the conference key establishment protocols gained more and more popularity. However, the current master thesis analyzes mostly the two-party case in which the key agreement is performed between pairs of devices.

Another aspect in which the key establishment schemes differ is the nature of the initial trust relationship between the participants before the key agreement starts. We encounter situations with *direct* trust between the participants (existing long term keys, certificates etc.) or *indirect* trust, through a third party (key translation server - KTC, trusted third party - TTP, key distribution server - KDC, authentication server - AS). The involvement of the participants is another distinguishing factor when characterizing a key establishment scheme. We can have *online* or *interactive* key establishment in which a conversation between the participants is required and *offline* key establishment where the

key can be computed individually by each participant using only the information he already possesses.

There are certain properties that we require for a KES: forward secrecy, key confirmation, implicit and explicit key authentication, key control, freshness of the key, effectiveness, efficiency, collusion resistance and if possible a formal proof of these properties to guarantee the correctness of the method. We do not claim that this list of characteristics is complete, but rather relevant for the KESs that we are concentrating on. The list was mainly inspired by [9] and [24] and their way to characterize key establishment schemes.

Forward secrecy is the property of a key establishment process which guarantees the confidentiality of a key even after the long term information used to generate it, was compromised. This property is usually achieved by introducing some secret random elements in the key construction.

Key confirmation is the property of a protocol to ensure that one or both parties are in possession of the agreed key. This is performed usually through a challenge-response step or through the encryption of some temporary information (e.g. a timestamp) after the key was established.

Key authentication is a property that ensures that no other party aside from a specifically identified second party can gain access to the secret key.

Freshness is the property of a key that guarantees that the key is newly generated. On the contrary, a key is not fresh if it was reused by one of the parties. The freshness of the key is important in order to prevent the reuse of compromised keys. There are two main techniques to achieve key freshness: either by using nonces or by using timestamps. The timestamp approach has the advantage of minimizing the number of required online messages in a protocol, but it requires that the two participants have their clocks synchronized [27], a property that is not always easy to achieve in practice.

Key control refers to the source of information from which the key is derived. In some cases the key is generated by using key material from only one source and then we say that only one party has control on the key generation, while in other cases both of them contribute to the generated key with random material, therefore they both have control on the key. It is important for both parties to have key control in order to guarantee that a malicious participant cannot force the establishment of a compromised key of his choice during the key establishment.

Effectiveness is the property of a key establishment protocol which is necessary to successfully establish a key after the protocol steps were carried out. This concept is very important for random key establishment protocols in which the success rate of a given protocol is a decisive factor for its practical use.

Efficiency has two dimensions in the context of key establishment protocols: *communication efficiency* and *computational efficiency*. The communication efficiency refers to the number and the size of the messages that participants send in case of online key establishment protocols. A widely used convention is to call a protocol *x-pass*, where x is the number of messages exchanged. On the other hand, computational efficiency refers to the number and the complexity of the operations that are needed to be performed by the participants during the protocol. In case of protocols using asymmetric encryption or bilinear maps, this property is extremely important to consider while analyzing and designing them.

Collusion resistance is the property of key establishment schemes, traditionally of the distributed ones, that protects against the collaboration of multiple users in order to derive the keys used by different user(s). We call a scheme k-collusion resistant if it protects against k collusive users, but it fails to do so for $k + 1$.

The proof of correctness is a crucial property of a key establishment protocol and it is usually constructed using computational complexity techniques, by showing a reduction from the problem of breaking the protocol to another problem believed to be hard [11]. The first such technique was proposed in [4] and is still one of the most widely used. Another popular method was introduced in [10], and afterwards, naturally, a large number of techniques appeared in the last decades.

1.2 Key Establishment Schemes in the Internet of Things Era

In the last years, there is a tendency upon interconnecting the devices that surround us using the Internet. This phenomena is called *Internet of Things* (IoT) and it has unique requirements, fundamentally different from the traditional personal computers paradigm. The embedded hardware used have limited resources in computation power, memory, bandwidth. The devices need to operate in real time and to process huge amount of data. Another important aspect is the huge number of the devices used in an IoT application, which results in long size for identifiers. Security and privacy plays an even more important role in this context since we are talking about physical devices that can easily harm people or the environment if controlled by an adversary. All these particularities impact the choice of key establishment scheme for a given IoT application. Traditional solutions proved to be almost impossible to deploy in practice due to their high demand for computation resources. We will discuss further, chronologically, the possible key establishment solutions to be deployed in IoT scenarios.

From a historical point of view, **symmetric cryptography** was the first one to be used on a large scale in the modern computers. For this case, the initial solution for key establishment was key transport, but this can not be used in our scenario considering the security requirements of modern applications. Next in line were the server-based key establishment solutions, like Kerberos, which involved interactions of the participants with a central server. In the IoT case this is unacceptable since most of the communication is over distributed adhoc protocols. Besides this, deploying a central entity to mediate between such a big number of devices would introduce a bottleneck in the system. The next development for the traditional paradigm was the Diffie-Hellman protocol which was proved to offer unconditional confidentiality. It is to this day one of the building blocks or source of inspiration for the modern KESs. However, it has a couple of limitations, mainly related to the lack of authentication built in the scheme. This makes the protocol vulnerable to a man-in-the-middle attack, allowing an attacker to masquerade as a legitimate user. This is unacceptable for IoT and therefore an additional layer of security needs to be introduced in order to authenticate the principals. This is usually done using asymmetric cryptography and digital signatures, but implementing this solution on lightweight devices is extremely expensive.

The next option that can be used is a fully **asymmetric solution**: deploying certificates infrastructure and perform simple key establishment over the provided secure channel. This is a pretty standard solution in case of desktops, but not so well-suited for IoT since it involves the distribution and the maintenance of the certificates and also heavy computations required by the public key algorithms. A similar solution is **identity-based cryptography** which does not require the distribution of the public keys in the network since the identities plays this role. However, the current solutions for implementing such systems involve pairings which are extremely computation-expensive mathematical tools.

A more suitable solution for our IoT scenario seems to be **key predistribution** which is a hybrid that involves elements from most of the previously presented schemes. It implies the existence of a central authority with a passive role that only distributes key material at node registration, instead of getting involved actively in the key establishment process. This setup of the system is very similar to the public key infrastructure and it also guarantees key authentication. However, key predistribution is much more resource-efficient than the asymmetric cryptography solutions, consisting of polynomial evaluations most of the time. Some of the schemes are also identity-based, having the strengths of these constructions as well, but paying a small cost for it. Two major limitations of these constructions are: the low collusion resistance property and the lack of forward secrecy. HIMMO, the scheme we are working with in the current master thesis and presented in Chapter 3, aims at solving the first limitation out of these, while the second one can be addressed at a protocol level, by negotiating a session key using the communication channel provided by the KPS. In the rest of this work, we will mostly discuss the key predistribution case which we think will have major applications in the nearby future, considering the current tendencies.

1.3 Contributions

The main contribution of our work is to introduce HIMMO-LI, an extension for the HIMMO key establishment scheme, designed for better scalability and security. We adapt a series of existing algorithms for fast key computation on the node's side to our case and we analyze their performance. We implement these algorithms and we measure their performance in terms of time, RAM and FLASH memory. We relate these results to additional parameters such as the number of parallel HIMMO systems, the performance of a collusion attack, ratio between identifiers and key size etc. Using this, we draw some conclusions on how HIMMO-LI should be used in practice to achieve different goals. Finally, we propose a way of relating the lattice dimension of the collusion attack against a node to the number of security bits. Using this and some very simple key establishment protocols, we make a comparison between HIMMO and other similar schemes. We do this only at a theoretical level, using the results reported in the literature for other schemes on the same CPU.

1.4 Organization of the Thesis

Chapter 2 presents some key predistribution schemes, a subcategory of key establishment procedures that the HIMMO scheme is also part of. In addition, it introduces some theoretical concepts to be used further. We present HIMMO key establishment scheme and its long identifiers extension (HIMMO-LI) that is the main subject of our work in **Chapter 3**. We adapt a series of algorithms to our case and we analyze their performance. This chapter also gives a brief description of the possible collusion attacks that can be mounted against HIMMO-LI. **Chapter 4** describes our implementation of the designed algorithms in three different programming languages: C, Java and assembly. In **Chapter 5**, we introduce our experiments' framework designed to run high volume tests and to collect different parameters. We present our empirical findings on the performance of the algorithms we designed for HIMMO-LI. Lastly, we introduce a way of relating the lattice dimension of a collusion attack to the number of security bits. **Chapter 6** contains our comparison of HIMMO-LI with other similar schemes from a performance point of view. Finally, we conclude in **Chapter 7** and identify possible future directions of research.

Chapter 2

Background

In this chapter we will introduce the traditional key predistribution schemes and their limitations as a preparation for the HIMMO scheme which will be described in Section 3.1. In addition, we will introduce notions on elliptic curves and identity-based cryptography used in the comparison realized in Chapter 6 and also some simple lattice concepts to be used in the discussion upon security of HIMMO-LI.

2.1 Traditional Key Predistribution Schemes

In this section we introduce three of the most important key predistribution schemes, using a generic framework to describe them introduced by Matsumoto and Imai in [23]. Every such scheme consists of three phases which will be independently highlighted: initialization of the TTP, node registration and operational phase.

2.1.1 General Structure

We will start with describing the general structure of a key predistribution scheme (KPS). Such a system was first introduced in [23] as a method for key distribution that consists of the following simple steps: (I) the server generates the keys - or a set of algorithms for generating these keys - for each participant / group, (II) the server distributes the corresponding keys / algorithms to all the participants, (III) the participants choose / generate the shared keys using the identity of the other participant(s). The authors of [23] proposed a generic way for computing the shared key in a conference with e members:

$$X^{(i)}(\xi_1, \xi_2, ...\xi_{e-1}) = f(y_i, \xi_1, \xi_2, ...\xi_{e-1}), \tag{2.1}$$

where $\xi_1, \xi_2, ..., \xi_{e-1}$ are identities placeholders and $X^{(i)}$ is the algorithm that the server delivers to the participant with the identity y_i. This algorithms is further used by the node to compute the shared key. In most of the cases, f is a n-variate symmetric function and the algorithm $X^{(i)}$ is an evaluation of this symmetric function using the identity of the participant and possibly some additional information. In the symmetric polynomial case, the algorithm represents a polynomial with fewer variables which will further be evaluated to compute shared keys using the identities of the other participants. For the case that we will mostly consider in this master thesis, in which only two participants i and j want to agree on a key of size M, the previous equation becomes:

$$K_{i,j} = K_{j,i} = X^{(i)}(\xi_j) = X^{(j)}(\xi_i). \tag{2.2}$$

While describing the following schemes, we will use the notations introduced in this section and highlight each element from the presented framework as we encounter them.

2.1.2 Blom's Scheme

In [5], Rolf Blom introduced two famous practical key establishment schemes, one based on maximum distance separable (MDS) codes and another based on polynomials. They were so influential that they created a new subcategory of KESs and one of the schemes, which works with MDS codes and is described in the end of this section, carries his name, thus it will be further referred as Blom's scheme. The other one was refined by Blundo et al. and became the Blundo scheme which will be described in detail in Section 2.1.3. In its initial form, Blom's scheme can be described as follows:

- **Initialization:** In a system that can contain at most $N = b^l$ users, each user receives an address ξ_i in the interval $0, ..., N-1$ which can be expressed as a vector $\xi_i = (\xi_{i0}, \xi_{i1}, ..., \xi_{i(l-1)})$ containing a b radix number.
- **Node Registration:** We define l symmetric functions on subsets of GF(2^M) (M is the size of the key in bits) which can be visualized as a $b \times b$ matrix, where each function associates a b-bit value to another b-bit value:
$$f_m(\xi_i, \xi_j) = f_m(\xi_j, \xi_i); \quad \xi_i, \xi_j \in \{0, 1, ..., b-1\}^l, 1 \leq m \leq l. \tag{2.3}$$
Since this $b \times b$ matrix is symmetric, we can distribute the corresponding column or row to each participant in the following way: if $\xi_{im} = x$, user i will receive the $(x+1)^{\text{th}}$ row of the table corresponding to f_m. So every user will receive lbM bits, where l is the number of functions, b the number of inputs for each values and M the number of bits of the output of each function evaluation.
- **Key Agreement:** The shared key between user i and j is:
$$K_{i,j} = \sum_{m=0}^{l-1} f_m(\xi_{im}, \xi_{jm}). \tag{2.4}$$
Since user i has his shares $f_m(\xi_{im}, \cdot)$, representing a row in the symmetric function tables, he will be able to compute all the terms of the sums in Equation 2.4. The algorithm in the Matsumoto and Imai framework, i.e. the left side of Equation 2.1, is in this case the set of all the portions of the symmetric functions f_m evaluated in his identity $f_m(\xi_{im}, \cdot)$.

The problem with this scheme is that a very small number of participants are able to collude in order to create a new key by combining their knowledge on the $f_m(\xi_i, \xi_j)$ function. To overcome this problem Blom proposes the use of MDS codes for assigning the identities to users in a system. A k-dimensional subspace of $GF(q)^n$ in which every non-zero vector has at least l non-zero coordinates is called a maximum distance separable (MDS) code with minimum distance l and length n denoted by (n, l). These codes have the property that two codewords (in our case identities) have at most $l-1$ elements in common. Thus, the number of users that need to collude in order to generate an additional key, which operation is equivalent to reconstructing the matrix itself, is $\lceil n/(l-1) \rceil$. However, the MDS codes have the disadvantage that they do not exist for all values of n, b and l, so for a system with n users there are limited combinations that can be used in practice.

In [6], a new way of viewing the scheme was introduced, using matrices and focusing on coding theory. This new way became increasingly more popular than the initial one. For building a system with n users in which a minimum of k users need to collude for breaking the scheme, we need to carry out the following steps:

- **Initialization:** A public generation matrix G is selected and delivered to all the participants. Then the trusted authority selects an (n, k) MDS code over $GF(q)$ with q a prime number and constructs a secret symmetric matrix D of dimension $n \times n$ in $GF(q)$ using this code.
- **Node Registration and Key Agreement:** The keys to be used by the participants can be calculated by means of the following equation:
$$K = (DG)^T D. \tag{2.5}$$
The trusted authority delivers the i-th row of D to user i, so that he will be able to compute all his keys corresponding to other users, but nothing more. Due to the minimum distance guaranteed by the MDS code, two rows have at least $(k-1)$ different elements. So a minimum of k users need to cooperate in order to reconstruct the matrix K.

2.1.3 Blundo's Scheme

In [7] the problem of distributing keys using a key distribution center is formalized using a probabilistic model and certain properties about such systems are identified. The paper also refines the polynomial scheme proposed by Blom creating probably the most analyzed and improved scheme in the field which we will further refer to as Blundo's scheme. It was originally designed for conferences with k participants, but in this master thesis we will present the simplified version, the case where only two-party keys are allowed:

- **Initialization:** Let $f(X, Y)$ be a symmetric bivariate polynomial $f(X, Y) = f(Y, X)$ over $GF(q)$ with $q > n$, where n is the number of users in the system. The polynomial is chosen by a trusted third party and it is kept secret.
- **Node Registration:** Shares of the polynomial f are distributed to each user i:
$$X^{(i)}(Y) = f(\xi_i, Y). \tag{2.6}$$
- **Key Agreement:** If users i and j want to agree on a shared key, they compute it using each other's identity by employing the following relation:
$$K_{i,j} = K_{j,i} = X^{(i)}(\xi_j) = X^{(j)}(\xi_i) = f(\xi_i, \xi_j). \tag{2.7}$$

The authors proved that for a polynomial f of degree k, the scheme is *unconditionally k-secure*, hence $k + 1$ nodes need to cooperate to reconstruct the initial polynomial. This can be easily done using Lagrange's polynomial interpolation and the polynomial shares of the k collusive nodes.

Blundo's scheme has the advantage of being easier to understand, since it is based only on simple polynomial evaluations. It is also extremely efficient to implement in hardware due to the simplicity of the required operations for performing the key agreement. However, the k-security property is a characteristic that made it not suitable for practical scenarios. We will discuss this aspect in detail in Section 2.1.5.

2.1.4 Zhang Scheme

In [34] a *random perturbation scheme* was proposed which tries to improve the k-security property of Blundo's scheme by introducing a polynomial ϕ_i in the generation of the key material for each user. The scheme is still predistribution based, therefore it enables pairwise key agreement. They define ϕ_i a *perturbation polynomial regarding r* as a polynomial with *limited infection* over $GF(q)$ meaning that when evaluated, it generates maximum r bits. The scheme structure is the following:

- **Initialization:** The trusted authority picks a symmetric bivariate polynomial $f(X, Y)$ of degree k, a set of perturbation polynomials ϕ_i regarding r and a hash function h.
- **Node Registration:** The authority computes and predistributes shares of the polynomial to the participants:
$$X^{(i)}(Y) = f(\xi_i, Y) + \phi_i(Y). \tag{2.8}$$
Due to the fact that the participants receive only the sum of the two polynomials, they cannot determine the coefficients of the two polynomials f and ϕ.
- **Key Agreement:** If participant i wants to establish a key with j, it computes the common key:
$$K_{i,j} = \frac{X^{(i)}(\xi_j)}{2^r}. \tag{2.9}$$
As ic can be seen in Equation 2.9, when computing the key, the last r bits that were directly affected by the perturbation polynomial are discarded and the key is formed by using the remaining bits. However, due to the initial addition of the r bits of the perturbation polynomials on both sides, the key can differ and an additional correction step is required.

- **Correction:** User i computes the hashed value of the key he computed, $h(K_{i,j})$, and sends it to user j. Meanwhile, j also computes his version of the shared key, but also two corrected values of the key:

$$K_{j,i} = \frac{X^{(j)}(\xi_i)}{2^r}; \quad K^+_{j,i} = K_{i,j} + 2^r; \quad K^-_{j,i} = K_{j,i} - 2^r. \tag{2.10}$$

  Participant j then computes the hashes of these values and compares them with the one received from user i, picking the one that matches. Intuitively, the effect of the perturbation polynomial can be either a carry to the r-th bit or a borrow from that bit. This is why the shared key has to be one of these three values. The correction step has the disadvantage of requiring an additional message to be sent as part of the protocol, hence the involvement of another participant in this case. This makes it impossible to generate the key passively as in case of the two previously described schemes.

  In [34], a successful and famous scheme was introduced, inspiring case studies in various fields, i.e. Wireless Sensor Networks (WSNs) or Vehicular ad hoc Networks (VANETs). This method has the speed advantage of Blundo's polynomial scheme and supposedly an increased security compared to Blundo's. Unfortunately, in [1] an error correction based attack was presented. It requires the compromise of $3k$ nodes to retrieve the key, where k is the degree of the bivariate polynomial. This attack proved the noisy approach of Zhang's scheme to be invalid and raised serious questions about the usefulness of Blundo-based schemes in practice.

2.1.5 Collusion Attacks against Traditional KPSs

Since the nodes generate keys in the case of KPSs using their key material, they expose themselves to so called *collusion attacks*. This means that a number of users conspire (collude) together in order to reveal the key material of a given node. This can be done, by putting together the information that each of them know about that node, usually the pairwise key. In the case of polynomial schemes, this attack can be easily mounted using an interpolation algorithm, therefore the number of nodes that need to collude is equal to the degree of the nodes key material. So, a simple solution to protect against such attacks is to increase this parameter, but the degree of the polynomial is also a measure of the computation time required for generating the key. Therefore, there is a tradeoff between the two forces. In the case of IoT, for which KPSs are suitable, deploying nodes is extremely cheap and most of the time a huge number of participants are involved. Thus, an attacker can mount an attack relatively cheap that involves the deployment of a big number of malicious nodes which will be used to attack a specific target node. We have to mention that a similar attack can be performed against the root key material of the TTP in most of the schemes, but for our work, due to its simplicity, we only consider the case when another node is attacked. To conclude, the security of the traditional schemes depends on two opposing forces: the number of malicious nodes that an attacker can deploy as part of an attack and, performance of the system. This however, is not a fair fight, since increasing the polynomial degree results in a linear increase in the number of nodes to deploy (security of the system), but most of the time in a polynomial increase of the computation time as well (performance). This is one of the main reasons why KPSs are not widely deployed yet.

2.2 Other Theoretical Concepts

This section provides the reader with preliminary concepts for further understanding the present work. Firstly, Diffie-Hellman and digital signature algorithms using elliptic curves are described which will be used in the comparison in Chapter 6. Then, a basic scheme for identity-based encryption is presented that is essential to understand the same comparison in Chapter 6. Finally, some lattice concepts are introduced to be used in the reduction from a collusion attack to a lattice problem, presented in Section 3.3.

2.2.1 Elliptic Curve Cryptography

An elliptic curve is the set of all solutions for an equation of the following form:

$$y^2 = x^3 + ax + b, \tag{2.11}$$

together with a point O, which is called the *point at infinity*. We can define the addition on an elliptic curve in the following way: having two points P and Q, the result of the addition is the third point of intersection of the line PQ with the elliptic curve. The point at infinity is the neutral element for this operation, therefore $P + O = P$. The elliptic curve together with the newly defined operation forms an abelian group. In practice, we are interested in elliptic curves over a finite field $\mathbb{Z}_p^*$, with p a prime number. The discrete logarithm problem in this group (ECDLP) is defined as follows: having two points P and Q and knowing that $Q = k \cdot P$, find the natural number k that satisfies this property. There are certain classes of elliptic curves, recommended by NIST, for which the ECDLP is a hard problem. These curves are intensively used in cryptography for replacing the group $\mathbb{Z}_p^*$, previously used in a large number of schemes. The benefit of doing so is a much smaller key size: 1024-bit RSA key is equivalent with a 160-bit elliptic curve key. We will further discuss two constructions based on elliptic curves: Elliptic Curve Diffie-Hellman and Elliptic Curve Digital Signature Algorithm.

Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman (ECDH) is an anonymous key agreement protocol that allows two parties to establish a shared secret over an insecure channel [2]. This algorithm is an adaptation of the initial Diffie-Hellman design to the group of an elliptic curve over a finite field. The two principals agree on a public elliptic curve and a non-zero large order point on that curve G. Principal A picks a random nonce n_A, computes $P_A = n_A \cdot G$ and sends it to B. Principal B performs similar steps: chooses a random nonce n_B, computes $P_B = n_B \cdot G$ and sends it to A. Now the two principals can compute the shared point $P = n_A \cdot P_B = n_B \cdot P_A$ using the private nonce and the point received from each other. The security of ECDH depends on the assumption that the ECDLP is hard for the used elliptic curve, therefore it is important to pick a curve for which this holds.

Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) is an algorithm for generating signatures for arbitrary long messages in order to authenticate their origin. Before any message can be signed, all the participants agree on an elliptic curve and a generator of the group, G, with order n. Every participant that wants to sign messages needs a symmetric key (d, Q) such that $Q = d \cdot G$. The steps for generating and verifying signatures are the following:

Signing messages using ECDSA

1. Calculate a hash, e, of the message.
2. Select a random k from $[1, n-1]$.
3. Calculate a point on the curve $(x, y) = k \cdot G$.
4. Compute $r = x \bmod n$. If $r = 0$ repeat from 3.
5. Generate $s = k^{-1}(e + rd) \bmod n$. If $s = 0$ go to step 3. Publish signature (r, s).

Verifying ECDSA signatures

1. Check that r and s are from $[1, n-1]$.
2. Calculate a hash, e, of the message.
3. Compute $w = s^{-1} \bmod n$.

4. Calculate $u_1 = zw \bmod n$ and $u_2 = rw \bmod n$.
5. Generate the point $(x, y) = u_1 \cdot G + u_2 \cdot Q$.
6. The signature is valid if and only if $r = x \bmod n$.

The signature size is four times the security level, for example for a 80 bits level the signature is 320-bit long.

2.2.2 Identity-Based Cryptography

Identity-based cryptography is a type of public key cryptography in which the public key is represented by a string that uniquely identifies the owner. Such a system was first proposed by Adi Shamir in [31], by introducing an ID-based signature scheme. However, no practical identity-based encryption (IBE) scheme was proposed till recently. The existing solutions are based on a mathematical concept called bilinear map which we will define below.

A **pairing** or **bilinear map** is a map $e : G_1 \times G_2 \to G_3$, where G_1 and G_2 are additive groups with generators P and Q, while G_3 is a multiplicative group, all of them having order p. The bilinear map needs to satisfy the following properties:

- $\forall\, a, b \in \mathbb{Z}_p^* :\ e(P^a, Q^b) = e(P, Q)^{ab}$,
- $e(P, Q) \neq 1$,
- e has to be easily computable.

One of the most successful IBE pairing scheme is Boneh-Franklin, introduced in [8], which consists of four phases:

Setup The central authority picks:
1. two groups G_1, G_2 and the corresponding pairing,
2. a random $s \in \mathbb{Z}_q^*$, the master key,
3. $s \cdot P$, the public key, where P is a high order point in G_1.
4. the message and the ciphertext.

Extract The central authority generates the private key for a user:
1. $d_{\mathrm{id}} = s \cdot \mathrm{id}$ the private key of the user.

Encryption For encrypting a message m for a node with a given identity id:
1. choose $r \in \mathbb{Z}_q^*$,
2. compute $g_{\mathrm{id}} = e(\mathrm{id}, s \cdot P)$,
3. the ciphertext is $c = (r \cdot P, m \oplus H_2(g_{\mathrm{id}}^r))$.

Decryption For decrypting a ciphertext $c = (u, v)$ a node uses its private key as following:
1. $m = v \oplus e(d_{\mathrm{id}}, u)$.

2.2.3 Lattices

Having n linearly independent vectors $b_1, b_2, ..., b_n$, we can define a lattice as:

$$\delta = \mathcal{L}(b_1, b_2, ..., b_n) = \left\{ \sum x_i b_i \,|\, x_i \in \mathbb{Z} \right\} = \{ Bx \,|\, x \in \mathbb{Z}^n \}, \tag{2.12}$$

where B is called the base of the lattice. If B is a matrix of size $m \times n$, then n is the rank of the lattice and m its dimension. The determinant of a lattice is $\sqrt{det(B^T B)}$. A given lattice can be represented in multiple ways, since more sets of vectors can generate the same lattice, therefore some base are more interesting than others. One classical lattice problem is the Close Vector Problem (CVP) for which there exist a lot of algorithms, but all of them exponential:

CVP: Given an approximation factor γ, a lattice basis $B \in \mathbb{Z}^{m \times n}$ and a vector $t \in \mathbb{Z}^m$, find $v \in \mathcal{L}(B)$ such that $\|v - t\| \leq \gamma \cdot \mathrm{dist}(t, \mathcal{L}(B))$. With $\|x\|$ we denote the Euclidian norm of x.

Chapter 3

Extension of the HIMMO Scheme

In this section we will describe the HIMMO scheme together with its optimizations for speed on the node's side: *More is Less* and *HIMMO Transformation*. We will then introduce our extension HIMMO with long identifiers (HIMMO-LI) and construct algorithms to adapt these optimizations for this case. We will also analyze, from a theoretical point of view, the resource consumption of these algorithms in terms of computation time, RAM and FLASH memory. In the end, we will show a reduction from a collusion attack against a node in the case of HIMMO to a lattice closest vector problem and discuss the success factors of such an attack.

3.1 HIMMO Scheme

HIMMO Key Agreement Scheme [16] aims to achieve the same objectives as Zhang's scheme descried in Section 2.1.4: the efficiency of Blundo with higher collusion resistance. While describing the scheme we will use the same notation utilized by the authors: $\langle x \rangle_n$ to denote the remainder of the division of x by n. The scheme is based on two supposedly hard mathematical problems:

- *Mixing Modular Problem* (**MMO**): Let $m, q_1, q_2, ..., q_m, N$ be some positive integers, $f_1, f_2, ..., f_m$ be polynomials of degree at most $\alpha \geq 2$. Having $h(\xi) = \left\langle \sum_{i=1}^{m} \langle f_i(\xi) \rangle_{q_i} \right\rangle_N \; \forall\, \xi \in \mathbb{Z}$, the MMO problem is to construct a polynomial time algorithm that recovers $f_1, f_2, ..., f_m$ given m, α, N and $k \geq 1$ pairs $(\xi_i, h(\xi_i))$.
- *Hiding Information Problem* (**HI**): Let $f(X) \in \mathbb{Z}_N[X]$ be a polynomial of degree at most α and let $f(\xi)_b = \langle \langle f_b(\xi) \rangle_N \rangle_{2^b}$ be the last b bits of the evaluated polynomial modulo N. The HI problem is to construct a polynomial time algorithm that can reconstruct f based on $k \geq 1$ different pairs $(\xi, f_b(\xi))$.

The MMO problem is an instance of the Noisy Polynomial Interpolation Problem, introduced in [32] and shown to be a hard for sparse polynomials of degree at least $N(\log N)^{-1/2}$. The second problem, HI, was analyzed in [15] and it was concluded that no polynomial time algorithm exists that can solve this problem in a reasonable time on the existing hardware.

The actual scheme consist of the following steps:

- **Initialization:** The TTP selects four integers: m (the number of bivariate polynomials), $\alpha \geq 2$ (the degree of the polynomials), b (the size of the identifiers / keys) and N (the public modulus, an $(\alpha+2)b$ bits long number). In addition, the TTP selects m positive integers $q_1, ..., q_m$ of form $q_i = N - 2^b\beta_i$ with $1 \leq \beta_i \leq 2^b - 1$ and a hash function h. The choice of the target domain for h is dependent on the security requirements of the scenario in which HIMMO is used. It also picks m bivariate polynomials $f_1(X, Y) \in \mathbb{Z}_{q_1}[X, Y], ..., f_m(X, Y) \in \mathbb{Z}_{q_m}[X, Y]$. The authors expanded these polynomials using the following notation:

$$f_i(X,Y) = \sum_{j=0}^{\alpha} f_{i,j}(Y)X^j \text{ with } f_{i,j} \in \mathbb{Z}_{q_i}[Y]. \tag{3.1}$$

An easier way to visualize these polynomials is to use a matrix notation:

$$f(X,Y) = \begin{bmatrix} f_1(X,Y) \\ f_2(X,Y) \\ \vdots \\ f_m(X,Y) \end{bmatrix} \begin{bmatrix} 1 \\ X^1 \\ \vdots \\ X^\alpha \end{bmatrix} = \begin{bmatrix} f_{1,1}(Y) & f_{1,2}(Y) & \dots & f_{1,\alpha}(Y) \\ f_{2,1}(Y) & f_{2,2}(Y) & \dots & f_{2,\alpha}(Y) \\ \vdots & \vdots & \ddots & \vdots \\ f_{m,1}(Y) & f_{m,2}(Y) & \dots & f_{m,\alpha}(Y) \end{bmatrix} \begin{bmatrix} 1 \\ X^1 \\ \vdots \\ X^\alpha \end{bmatrix}. \tag{3.2}$$

- **Node Registration:** For each node $1 \leq \xi_i \leq 2^b - 1$ that wants to register, the TTP selects $\alpha+1$ integers $\epsilon_{i,j} \leq 2^{(\alpha+1-j)b-2}$ called noise, with $j \in 0, 1, ..., \alpha$. Afterwards it computes the key material shares and distributes them to the node:

$$\mathrm{KM}_{i,j} = \left\langle \sum_{k=1}^{m} \langle f_{k,j}(\xi_i) \rangle_{q_i} + 2^b \epsilon_{i,j} \right\rangle_N . \tag{3.3}$$

  If we relate to the matrix notation presented earlier, each key material share is generated using a column in the polynomial matrix and the corresponding noise. It is important to observe that each polynomial share is evaluated in its corresponding ring $\mathbb{Z}_{q_i}$ and the whole operation in $\mathbb{Z}_N$. The algorithm in the left side of Equation 2.1 simplifies in this case to $X^{(i)} = \begin{bmatrix} \mathrm{KM}_{i,0} & \mathrm{KM}_{i,1} & ... & \mathrm{KM}_{i,\alpha} \end{bmatrix}$.

- **Key Agreement:** If node i wants to agree on a key with node j, it has to compute first:

$$K_{i,j} = \left\langle \langle \mathrm{KM}_i(\xi_j) \rangle_N \right\rangle_{2^b} = \left\langle \left\langle \sum_{k=0}^{\alpha} \mathrm{KM}_{i,k}(\xi_j)^k \right\rangle_N \right\rangle_{2^b} . \tag{3.4}$$

  User i computes a hashed value of the key that he computed, $h(K_{i,j})$, and sends it to user j. User j computes his key $K_{j,i}$ in an analogous way using Equation 3.4. According to [16], $K_{i,j} \in \{\langle K_{j,i} + kN \rangle_{2^b} | -\Delta \leq k \leq \Delta\}$, where $\Delta = 3m + \alpha + 1$. So, user j has to search this set for the key that matches the hash value sent by i.

The preliminary security analysis performed so far in [15] and [16], seem to hint that for some parameters, the HIMMO scheme is a fully collusion resistant Blundo-based scheme suitable for constrained devices. However, it has one major limitation: its security relies on a complex relation between (α, b), b being both the key size and the size of the identifiers. In this context, if we want more nodes in the system we have to increase both the key size and the degree of polynomials.

3.1.1 Node's Side Optimizations

Considering that the TTP can be a node with more computation power than the normal nodes, it is important to optimize the operational phase of HIMMO instead of the node registration. This is also because the key establishment is performed more often than the registration of a node. The operational phase, that we want to optimize, is a simple polynomial evaluation modulo N. The first optimization that can be done at the node's side is to reduce the number of key material blocks that are used to update the key at each step. Due to the HI problem, only the last b bits of the key are used, therefore not all the blocks in the key material influence these bits. In fact only the last $(j+1)B$ bits of the key material KM_j are used, so the others should not even be stored in the node's side.

More is Less was proposed in [25] and is an optimization designed for speeding up the key computation. Analyzing the pairwise key establishment, we can observe that the most expensive

computation by far is the mod N operation which is performed $\alpha+1$ times. The idea of the optimization is to transform this operation in an addition, by fixing a special form for N. If we choose $N = 2^{(\alpha+2)b}-1$ we have the following equality:

$$\left\langle 2^{(\alpha+2)b} \right\rangle_N = 1. \tag{3.5}$$

This means that the bits with a higher rank than $(\alpha+2)b$ will be shifted with $(\alpha+2)b$ bits and added to the key. By exploiting this fact, we can compute multiplications in the following way:

$$\langle A \cdot B \rangle_N = \langle R \rangle_N = R_0 + R_1, \tag{3.6}$$

where R_0 are the least significant $(\alpha+2)b$ bits, while the R_1 are bits with higher rank than that. Algorithm 1, exploits this trick with the special form of N to speed up the polynomial evaluation modulo N for the key computation between two nodes. As it can be seen, the only modular operation left, is one modulo power of two, which only involves, on a binary CPU, selecting specific bits from a memory location. The two parts of the key that influence the key bits are decoupled due to the property expressed in Equation 3.6: the least significant bits and the bits starting at position $(\alpha+2)b$. In consequence, we need to store their effect in two separate variables (*temp* and *key*).

Algorithm 1: More is Less [25]. Notations: α - polynomial degree, b - key size, η' - the identity of the other node, $KM_{\eta,j}$ - j$^{\text{th}}$ key material.

Data: $\alpha, b, \eta', \mathrm{KM}_{\eta,j}$ where $j \in 0, 1, ..., \alpha$
Result: $\langle \langle \sum_{j=0}^{\alpha} \mathrm{KM}_{\eta,j}\, \eta'^{j} \rangle_N \rangle_{2^b}$

$key = \langle \mathrm{KM}_{\eta,\alpha} \rangle_{2^b}$
$temp = \langle \mathrm{KM}_{\eta,\alpha} \rangle_{2^{(\alpha+2)B}}$
for $j = \alpha \to 1$ **do**
  $temp = (temp \gg b) \cdot \eta'$
  $temp = \langle temp \rangle_{2^{(j+2)b}}$
  $temp = temp + \mathrm{KM}_{\eta,j} \gg b$
  $key = \langle key \cdot \eta' \rangle_{2^b}$
  $key = \langle key + \langle \mathrm{KM}_{\eta,j} \rangle_{2^b} \rangle_{2^b}$
  $key = key + temp \gg (j+1)b$
end

The second optimization was also presented in [25], is called ***HIMMO Transformation*** and we reproduced it in Algorithm 2 with small modifications. The idea is to simplify the algorithm and speed it up by grouping together the two parts of the key material that influences the final key. This is done using a shift with αB bits, which will result in shifting all the useful bits in nearby memory location, allowing the node to treat them uniformly. The shift of the key material is performed by the TTP before sending it to the node. In Figure 3.1, we present graphically this transformation.

Algorithm 2: HIMMO Transformation [25]. Notations: α - polynomial degree, b - key size, η' - the identity of the other node, $KM'_{\eta,j}$ - j$^{\text{th}}$ component of the transformed key material.

Data: $\alpha, b, \eta', \mathrm{KM'}_{\eta,j}$ where $j \in 0, 1, ..., \alpha$
Result: $\langle \langle \sum_{j=0}^{\alpha} \mathrm{KM}_{\eta,j}\, \eta'^{j} \rangle_N / 2^{\alpha b} \rangle_{2^b}$

$key = \mathrm{KM'}_{\eta,\alpha}$
for $j = \alpha \to 1$ **do**
  $key = key * \eta'$
  $key = key \gg b$
  $key = key + \mathrm{KM'}_{\eta,j}$
end
$key = key \gg b$

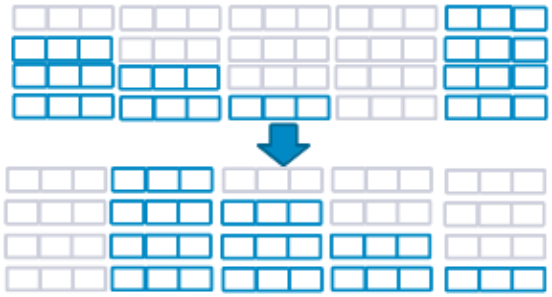

Figure 3.1: The transformation of the key material. Most significant bits are on the left and key materials are numbered from α downto 0 from up to down. The top part of the figure represents the key material for*More is Less*, and the bottom part the transformed key material. The gray bits can be zeroed with no effect on the key computation, therefore are not saved on the node's side

.

3.2 HIMMO with Long Identifiers

In this section we propose our extension, *HIMMO with Long Identifiers*, a modification of the HIMMO scheme that introduces an additional parameter B, the size of the identifiers, in order to decouple the key size from the identifiers size. Such a modification has profound implication on the security considerations presented in [16], which will be further discussed and analyzed during the experiments in Chapter 5. The addition of another parameter has an impact on the ease of understanding as well, and also on the possibility of comparing different HIMMO instances. As a consequence, we will have more variety, leading to questions like:

- Should we pick a system with both identifier and key size of 128 bits or is it better to choose 128 bits for the size of the identifier with two system instances of 64 key bits each?
- Can we compare such choices from a security point of view?
- What is the performance of the modified scheme?

After presenting the extension, we will analyze the theoretical performance impact for the nodes. We will mainly work on the two already presented optimizations for HIMMO that use a specific form of N.

3.2.1 Motivation

As it can be seen in the design of the HIMMO scheme 3.1, it has only one parameter b to denote the key size and the identifiers size. This has a negative impact on the performance of the system in many ways. First of all, a higher b means that more information from the computed polynomial mixing is available to the nodes. However, this is compensated in the HIMMO design by working with larger blocks b, therefore making the attacker's life harder since he needs to interpolate larger values. On the other hand, working with bigger values has a consequence on both the execution time (quadratic) and on the memory consumption (linear) as described in [16]. Therefore, a solution for achieving smaller computation times would be to have more instances of HIMMO with small b values. However, this is not possible in all the cases since for certain systems we want to have large b due to the possibly huge number of participants. In conclusion, we would like to have a system that allows **long identifiers**, but small key size. This is also the case in the IoT scenario described in section 1.2, since one of the requirements there is to enable the addition of a huge number of devices in the network. This property should not impact the decision on the key size in any way.

If we reflect on the HIMMO design, the decoupling of the amount of disclosed information (key size, b bits) from the amount of information used in the computations ($(\alpha+1)B+b$ bits polynomial coefficients) will also have an impact on the information theoretic security of the scheme. This is due to the HI problem on which HIMMO relies. In the case of traditional key predistribution schemes discussed in Section 2.1, increasing the security is equivalent to increasing the polynomial degree and in consequence the information that the attacker needs to reconstruct. In the case of HIMMO, the attacker has only b bits of the polynomial evaluation and therefore it is intuitive that increasing security can also be done by raising the ratio between the secret information (linearly dependent on the identifiers size) and the available information to the attacker (equal to the key size).

The current work aims at implementing a solution for decoupling the key size from the identifiers size and also offer a detailed view on the performance of such a system. Since HIMMO is still in the prototyping phase, a drastic change as the one described in the current work can be possible since it does not require any modification of existing hardware.

3.2.2 Proposed Solution

We propose a modified scheme, HIMMO-LI, which is slightly different from the original HIMMO:

- **Initialization:** The TTP selects four integers: m (the number of bivariate polynomials), $\alpha \geq 2$ (the degree of the polynomials), B **(the size of the identifiers)**, b (the size of the key) and N (the public modulus, an $(\alpha+1)B+b$ bits long number). In addition, the TTP selects m positive integers $q_1, ..., q_m$ of form $q_i = N - 2^b\beta_i$ with $1 \leq \beta_i \leq 2^B - 1$ and a hash function h. It also picks m bivariate polynomials $f_1(X,Y) \in \mathbb{Z}_{q_1}[X,Y], ..., f_m(X,Y) \in \mathbb{Z}_{q_m}[X,Y]$:

$$f_i(X,Y) = \sum_{j=0}^{\alpha} f_{i,j}(Y)X^j \text{ with } f_{i,j} \in \mathbb{Z}_{q_i}[Y]. \tag{3.7}$$

- **Node Registration:** For each node $1 \leq \xi_i \leq 2^B - 1$ that wants to register, the TTP selects $\alpha+1$ integers $\epsilon_{i,j}$ called noise, computes the key material shares and distributes them to the node:

$$\mathrm{KM}_{i,j} = \left\langle \sum_{k=1}^{m} \langle f_{k,j}(\xi_i) \rangle_{q_i} + 2^b\epsilon_{i,j} \right\rangle_N . \tag{3.8}$$

- **Key Agreement:** If node i wants to agree on a key with node j, it has to compute first:

$$K_{i,j} = \left\langle \langle \mathrm{KM}_i(\xi_j) \rangle_N \right\rangle_{2^b} = \left\langle \left\langle \sum_{k=0}^{\alpha} \mathrm{KM}_{i,k}(\xi_j)^k \right\rangle_N \right\rangle_{2^b} . \tag{3.9}$$

On the TTP side, the long identifiers modifications only impact the length of N and of the moduli q_i in the Equation 3.8. Same holds on the node's side, where Equation 3.9 remains unchanged and only the operands size and the modulus' length changes. The correctness and assumptions for HIMMO-LI holds since they are the same as in the case of HIMMO, we only changed the length of the coefficients for the polynomials and the number of bits we output as key.

On the other hand, the algorithms described earlier, for specific forms of N, change in fundamental ways. Both of them have a new form for the modulus $N = 2^{(\alpha+1)B+b} - 1$. For the case of *More is Less* illustrated in the Algorithm 3, the nodes need to treat separately the shift of the first block of the temporary result and the subsequent ones. This is due to the structure of the key material which is $(\alpha+1)B+b$ long. Since the bits used to update the *key* variable are stored in the key material as last bits, a shift with b bits is performed at line 10. Also the modular operations are mod b since this is the size of the key variable where the result will be stored.

Algorithm 3: More is Less with Long Identifiers. We used the same notations as for Algorithm 1, but with an additional parameter, B - the size of the identifiers.

Data: $\alpha, b, B, \eta', \mathrm{KM}_{\eta,j}$ where $j \in 0, 1, ..., \alpha$
Result: $\langle\langle\sum_{j=0}^{\alpha} \mathrm{KM}_{\eta,j}\, \eta'^{j}\rangle_N\rangle_{2^b}$

$key = \langle \mathrm{KM}_{\eta,\alpha} \rangle_{2^b}$
$temp = \langle \mathrm{KM}_{\eta,\alpha} \rangle_{2^{(\alpha+2)B}}$
for $j = \alpha \to 1$ **do**
  if $j = \alpha - 1$ **then**
    $temp = (temp \gg b) \cdot \eta'$
  else
    $temp = (temp \gg B) \cdot \eta'$
  end
  $temp = \langle temp \rangle_{2^{(j+2)B}}$
  $temp = temp + \mathrm{KM}_{\eta,j} \gg b$
  $key = \langle key \cdot \eta' \rangle_{2^b}$
  $key = \langle key + \langle \mathrm{KM}_{\eta,j} \rangle_{2^b} \rangle_{2^b}$
  $key = key + temp \gg (j+1)B$
end

Algorithm 4: HIMMO Transformation with Long Identifiers. Notations: α - polynomial degree, b - key size, B - identifiers size, η' - the identity of the other node, $\mathrm{KM}'_{\eta,j}$ - transformed key material.

Data: $\alpha, b, B, \eta', \mathrm{KM'}_{\eta,j}$ where $j \in 0, 1, ..., \alpha$
Result: $\langle\langle\sum_{j=0}^{\alpha} \mathrm{KM}_{\eta,j}\, \eta'^{j}\rangle_N / 2^{\alpha B}\rangle_{2^b}$

$key = \mathrm{KM'}_{\eta,\alpha}$
for $j = \alpha \to 1$ **do**
  $key = key * \eta'$
  $key = key \gg B$
  $key = key + \mathrm{KM'}_{\eta,j}$
end
$key = key \gg B$

For the case of *HIMMO Transformation* illustrated in Algorithm 4, the main implication is the way in which the key material is transformed. Since the blocks that need to be shifted are B-bit long, we need to use this parameter when aligning the useful blocks nearby. The *key* variable is $(\alpha+1)B+b-jB$ bits long, where j is a counter for the loop, since we only need to store as few bits as possible that include the final b bits which will be outputted.

3.2.3 Theoretical Analysis

The security of HIMMO-LI depends on the amount of information that needs to be obtained by the attacker in order to compromise a node and the amount of information he has available from a node he controls. As stated in Section 2.1.5, the number of collusive nodes that need to conspire in order to reveal the key material of a node, is a good indication of a KPS. In case of HIMMO-LI, from an information theoretical point of view, this value is:

$$c = \frac{B}{b}\left[\frac{\alpha(\alpha+1)}{2} + \alpha + 1\right]. \tag{3.10}$$

However, this is a theoretical value, that is very hard to reach in practice, since combining parts of different polynomial points together does not give a valid point to an attacker. Thus, a smarter attack than simple polynomial interpolation needs to be implemented in practice. For our analysis though, this is a good enough estimation since it represents a lower bound for the number of nodes that are needed for a successful attack.

As stated in Section 1.2, a key predistribution scheme needs to be very fast and memory efficient since its main applications are in the Internet of Things sphere. Therefore, in the rest of this chapter, we will analyze the time, RAM and FLASH consumption of the two proposed algorithms.

More is Less

The key material that needs to be stored in each device decreases with j and it has a stairs-like structure (see [25] for details). For each instance of HIMMO we have $\alpha + 1$ components for the key material. Each of them is $b + (j + 1)B$ long where j varies as in Algorithm 3. This yields a total key material size of $\left(\frac{(\alpha^2+3\alpha+1)B}{2} + \alpha b\right) n$ bits for each node.

The required RAM memory for such a procedure is dictated only by the two variables *key* and *temp* in Algorithm 3 whose intermediate values needs to be stored at each step. Also the identifier of the other node requires storage in the RAM memory. We can assume that it is transferred into the register when needed byte by byte from a different location e.g. flash memory, network. However, this is not a good practice since it will increase the algorithm's running time. Hence, the amount of memory required is $4b + (\alpha + 1)B$. If the number of HIMMO instances are executed in parallel on the same processor, then the memory requirements will be higher, $b + n((\alpha + 1)B + 3b)$.

When estimating the computation time, we do not consider the modulo to power of 2 operations and the shifts since they are performed in constant, negligible time (addition to a pointer). The expensive operations are in our case the additions and the multiplications (lines 5, 7, 10, 11, 12, 13 in Algorithm 3). Since the implementation of the algorithm will involve emulating long numbers using operands of the size allowed by the processor, the size of the CPU word (CPUW) variable is an important component of the time estimation.

If we consider the multiplication as the dominant factor the computation time is:

$$\mathcal{O}\left(n\frac{\alpha b^2 + B\sum_{j=1}^{\alpha}((j+2)B+b)}{\text{CPUW}^2}\right) \tag{3.11}$$

$$= \mathcal{O}\left(n\frac{\alpha b^2 + \alpha bB + 2\alpha B^2 + \frac{B^2\alpha(\alpha+1)}{2}}{\text{CPUW}^2}\right) \tag{3.12}$$

$$= \mathcal{O}\left(n\frac{\alpha^2 B^2 + \alpha b^2 + \alpha B^2}{\text{CPUW}^2}\right) \tag{3.13}$$

HIMMO Transformation

The FLASH memory requirements should not change since we store the same bits for the key material, due to the significantly lower code size, the overall flash memory consumption will be lower than in the case of Algorithm 3. The RAM memory consumption should also be about half, compared with *More is Less*, since we now store one big memory area (*key*) instead of two (*key* and *temp*).

The time consumption should also be significantly lower:

$$\mathcal{O}\left(nB\frac{\sum_{j=1}^{\alpha}((j+1)B+b)}{\text{CPUW}^2}\right) \tag{3.14}$$

$$= \mathcal{O}\left(n\frac{2\alpha B^2 + 2\alpha bB + \alpha(\alpha+1)B^2}{2\text{CPUW}^2}\right) \tag{3.15}$$

Comparing 3.15 with 3.13, we can observe that the performance gain is $\frac{\alpha b^2 + \alpha B^2}{\text{CPUW}^2}$. However, this factor gets dominated by $\alpha^2 B^2$ one, when we increase B or α.

3.3 Collusion Attacks on HIMMO-LI

Using the method in Appendix B in [16] we will show the reduction of a collusion attack against a node in case of HIMMO-LI to a lattice problem. In this attack we assume that c nodes collude in order to obtain the key material polynomial of another node x. As it can be seen in the design of the scheme, this polynomial has $\alpha + 1$ coefficients each being $(\alpha+1)B + b$ bits long.

Using the identities of the c colluding nodes, we introduce the Vandermonde matrix V of size $c \times (\alpha + 1)$:

$$V = \begin{bmatrix} 1 & \eta_1 & \eta_1^2 & \dots & \eta_1^\alpha \\ 1 & \eta_2 & \eta_2^2 & \dots & \eta_2^\alpha \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \eta_c & \eta_c^2 & \dots & \eta_c^\alpha \end{bmatrix} \tag{3.16}$$

We note the coefficients of the key material with $r_k \in \mathbb{Z}_N, 0 \leq k \leq \alpha$ and we define the column vector $r = (r_0, r_1, \dots r_k)^T$. We also notate with $h = (h_x(\eta_1), h_x(\eta_2), \dots, h_x(\eta_c))^T$ the shared keys computed by the colluding nodes using the identity of the attacked nodes $(h_x(\eta_i) = h_i(\eta_x), 0 \leq i \leq c)$. Now, we can express our attacks using the newly introduced notations: given h the shared keys computed by the colluding nodes with the attacked node, find the integer components r of the key material and

$$h = \langle\langle Vr\rangle_N\rangle_b = Vr - N\left\lfloor\frac{Vr}{N}\right\rfloor - b\left\lfloor\frac{Vr - N\left\lfloor\frac{Vr}{N}\right\rfloor}{b}\right\rfloor. \tag{3.17}$$

For expanding the second part of the previous equation, we used the property $\lambda = \langle a\rangle_b = a - b\lfloor\frac{a}{b}\rfloor$ valid for any rational number. For any positive integers a, b the computed value λ is unique such that

$|(a/b)-\lambda-(a-1)/2b| < 1/2$. In the case when a and λ are vectors, the equations should be valid for each pair of elements (a_k, λ_k). Therefore, the unified condition becomes $\max_k |a_k/b - \lambda_k - (b-1)/2b| < 1/2$.

Using this newly introduced property, our attack can be defined as follows: given h, find r and λ_1, λ_2 such that:

$$h = Vr - N\lambda_1 - b\lambda_2 \tag{3.18}$$

Applying the previous constraints on λ_1 and λ_2, we obtain:

$$\left\| \frac{Vr}{N} - \lambda_1 - \frac{(N-1)e_c}{2N} \right\|_\infty < \frac{1}{2}, \tag{3.19}$$

where e_c is a column vector of length c and

$$\left\| \frac{Vr - N\lambda_1}{b} - \lambda_2 - \frac{(b-1)e_c}{2b} \right\|_\infty < \frac{1}{2}. \tag{3.20}$$

With $\|x\|_\infty$ we denoted $\max |x_i|$, the maximum element of x. Knowing that the coefficients of r are elements in $\mathbb{Z}_N$, we can also express the following constraint:

$$\left\| \frac{r}{N} - \frac{(N-1)e_{\alpha+1}}{2N} \right\|_\infty < \frac{1}{2}. \tag{3.21}$$

Now we define a column vector x and a matrix A:

$$x^T = (-\lambda_1^T - \lambda_2^T r)^T, \tag{3.22}$$

$$A = (NI_c \; bI_c \; V), \tag{3.23}$$

where I_c is the identity matrix of dimension $c \times c$. The vector x is of length $2c+\alpha+1$, while matrix A's dimension is $c \times (2c+\alpha+1)$. Now Equation 3.18 can be expressed in a much simpler way:

$$h = Ax. \tag{3.24}$$

In order to be able to express the constraints in Equations 3.19, 3.20 and 3.21 in a condensed way, we need to define a matrix B of size $(2c+\alpha+1) \times (2c+\alpha+1)$ and a column vector u of size $2c+\alpha+1$:

$$B = \begin{bmatrix} I_c & 0 & \frac{V}{N} \\ \frac{N}{b} I_c & I_c & \frac{V}{b} \\ 0 & 0 & \frac{1}{N} \end{bmatrix}, \tag{3.25}$$

$$u^T = \begin{bmatrix} \frac{N-1}{2N} e_c^T & \frac{b-1}{2b} e_c^T & \frac{N-1}{2N} e_{\alpha+1}^T \end{bmatrix}. \tag{3.26}$$

Now, all the constraints expressed earlier are described by the following equation:

$$\|Bx - u\|_\infty < \frac{1}{2}. \tag{3.27}$$

We showed so far that attacking a node in HIMMO-LI is equivalent to solving Equation 3.24, when the constraint of Equation 3.27 is satisfied. Let now x_0 be a solution of Equation 3.24. Every other integer solution x can be written as $x = x_0 + w$, where $Aw = 0$. Now, we can express our problem in terms of lattices:

$$\|y - (u - Bx_0)\|_\infty < \frac{1}{2}, y \in \mathcal{L}, \tag{3.28}$$

$$\mathcal{L} = \{Bw | w \in \mathbb{Z}^{2c+\alpha+1}, Aw = 0\}. \tag{3.29}$$

This lattice's dimension is $c + \alpha + 1$**.** The described problem is the well known CVP problem presented in Section 2.2.3 where we want to find a vector y that has less than $1/2$ to a target vector $u - Bx_0$.

The implications of the HIMMO-LI changes on a lattice attack on the root key material are the same as the presented case for attacking only a node. This is the case mainly because the reduction is analogous, obtaining a lattice of dimension $m(c+\alpha+1)$ as can be seen in [16], where m is the number of polynomials used by the TTP. The main effect of our modifications is on the number of colluding nodes, c which appear in both the attacks. In this work we consider the attack against a node since it is simpler and it requires a lattice with smaller dimension.

3.4 Main Results

In this section, we proposed two algorithms optimized for time to be used for the case of HIMMO-LI. Both of them have a quadratic FLASH demand and computation time, but a linear RAM need. *HIMMO Transformation* performs slightly better in terms of time than *More is Less* due to its simplicity. In Chapter 5, we will verify empirically these results on a lightweight device.

Replacing the result in Section 3.2.3, in the lattice dimension obtained in Section 3.3, we have:

$$d = \frac{B}{b}\left(\frac{\alpha(\alpha+1)}{2} + \alpha + 1\right) + \alpha + 1, \tag{3.30}$$

which is a good indication of the performance of the best attack that can be mounted against HIMMO. From the discussion above, we can see that the security of the system depends mainly on α, the degree of the polynomial used, and on the $B\backslash b$ ratio. In Chapter 5, we analyze the lattice dimension growth and we present a way of translating from this dimension to standard security levels.

Chapter 4

Implementation

We implemented our algorithms in three different programming languages at various levels of abstraction. The Java implementation was the simplest and was used mainly for creating a testing framework and as a tool for assisting in debugging the other two implementations. The C version was designed to be used on AVR microcontrollers and it uses custom libraries to interact with the hardware (interrupts, ports, eeprom etc.). The ASM implementation had the lowest level of abstraction and was implemented using the instruction set of the ATmega 128L processor. When we implemented the three versions we kept their intended usage in mind:

- the Java code to be used as learning tool and for assisting in the debugging process due to its simplicity,
- the assembly code to be used in evaluating the performance of HIMMO-LI since it has the best results,
- and the C code we planned to use in small prototype applications due to its relatively low cost of porting it to a different instruction set.

The Java code is the only one to include a trusted third party (TTP), while the other two implement only the nodes' operational phase. The C and Java versions were written from scratch, while the assembly version is an enhancement of an existing HIMMO code. In the following part of this chapter we will briefly describe each of the implementations and we will present the biggest challenges encountered on the way.

4.1 Java Implementation: TTP and Nodes

We implemented both the TTP side and the nodes side for HIMMO and HIMMO-LI. Even though, there existed a HIMMO implementation in Java, we rewrote the code from scratch in order to focus the design on the generation of key materials in different output formats. We did so, for better understanding the internals of the HIMMO scheme as well. Our implementation uses the BigInteger class provided by the standard java libraries. Our initial phase of the TTP also uses a custom implementation for polynomials operations. We implemented three different algorithms on the nodes side: naive HIMMO using polynomials, More is Less and HIMMO Transformation. All the implementations are capable of generating computation traces for certain key materials. Such an output consists of intermediate values $key, temp$, represented as hexadecimal numbers, computed after each update with a key material entry. These values are used for debugging the other, lower level implementations. The TTP implementation is capable of generating key materials for the nodes either in Java BigInteger format or in compiler-ready C header files or assembly memory files.

4.2 C Implementation: Nodes

For implementing HIMMO in embedded C, we needed to find a lightweight library for big numbers which can run with minimum overhead on an 8-bit microcontroller. Since neither of the available solutions fit all our needs, we decided to build a simple library whose API we present in Listing 4.1. It emulates long numbers using arrays of bytes. Before any operations can be performed, the *aux* variable needs to be initialized with a specific size in bits. This is done in order to avoid multiple dynamic allocations for temporary variables used in computations which are extremely expensive. This detail prevents the use of our implementation in a multi-threaded scenario, but for our 8-bit microcontroller it was not a must. The first block of available functions, which are used for initialization, are dealing with memory allocations and are very simple with $\mathcal{O}(l)$ efficiency, where l is the size of the number in bytes. The shift operations have the same complexity, but they are faster since they involve only modifications of the previously allocated memory areas. The last group of services, the search / status functions, are even faster since they do not involve any update of memory areas, only the inspection of stored values. The current performance of addition / subtraction, multiplication and modulo operations are respectively $\mathcal{O}(\max(l1, l2))$, $\mathcal{O}(l1 \cdot l2)$, $\mathcal{O}(l1 \cdot l2)$. For an addition / subtraction we perform bytewise operations with carry transport, while for modulo and multiplication we implemented the schoolbook versions which imply operations between every two pair of bytes of the two operands.

Listing 4.1: The API of the custom C big numbers library

```
typedef struct {
       uint16_t size_bits;
       uint16_t size_bytes;
       uint8_t *bytes;
} long_number;

void init_aux(int no_bits);
long_number * clone(long_number *);
long_number * init_long_number(int);
long_number * int_to_long_number(int , int);
long_number * array_to_long_number(uint8_t[] , int );
void free_ln(long_number *no);

void shift_left_one_bit(long_number *);
void shift_right_one_bit(long_number *);

void add_in_situ(long_number *, long_number *);
void sub_in_situ(long_number *, long_number *);
void multiply_in_situ(long_number *, long_number *);
void mod_in_situ(long_number *, long_number *);
void mod_pow2_in_situ(long_number *no, int pow);

int get_msb_position(long_number *no);
bool grater_than(long_number *, long_number *);
bool is_zero( long_number *a );
```

The heavy computation part of the library are the arithmetic operations. In the current phase of the implementation we focused on correctness and not on efficiency, but this should be the place for future improvements, since the current code, although correct, it is quite slow.

Using our library, we implemented HIMMO-LI, on the node size, for embedded devices. We used AVR library to access ports, timers and EEPROM memory services, hence we tightly connected our implementation to this family of processors. We also used asserts for verifying the correctness of the computations.

The most interesting aspect encountered during this phase was finding a way to include the key material. Even when it is statically defined in a header file and marked with *const*, a char array is still allocated in the RAM memory area. Considering, that this resource is scarce compare to the FLASH, we would like to force the compiler to allocate this key material in the program area. Doing so requires the annotation of the array with PROGMEM and using special library calls for manipulating these memory locations. We created separate initialization functions for dealing with this issue and we modified the Java TTP to generate key materials that follow these guidelines. This problem was discovered while trying to run test cases with large input. When inspecting the memory, we realized this particularity of the compiler.

4.3 ASM Implementation: Nodes

Our ASM implementation is practically an enhancement of the already existing HIMMO source code. We started our work by reverse engineering the existing implementation and by getting used with the instruction set and the resources of the microprocessor. After getting one or two scenarios to run using the old spaghetti code, we quickly realized that we need to refactor the code using function calls instead of GOTO instructions. Doing so was not a straightforward task since the stack pointer's default location is overlapping the data area in ATmega128L. As an effect, calling any function results in random values being written in the Program Counter upon function return. Identifying this problem was extremely difficult, but fixing it was very easy and it just consisted of defining the base stack pointer value as being the end of the RAM memory.

Two other interesting **bugs discovered** during the reverse engineering process were related to the generation of the key material and with the carry propagation during operations with long numbers. The key generation problem appeared when converting from the BigInteger format that the TTP uses in Java to the assembly memory format. The translation was done using the printing as hexadecimals provided by java.string and subsequently translate two hex digits at a time in a byte. This transformation involved changing from MSB format to LSB and hence lose the leading zeros from the BigInteger representation. This resulted in bad key material structure and was solved by adding more logic to the TTP to include the leading zeros in the string representation of the long number before translating it to assembly. The second problem was causing about 5% of the key agreements to be computed incorrectly. The problem was that the carry was being propagated only to the next byte after an addition was completed. The case when the update with the carry was causing an overflow was not treated. To solve this, we simply updated with the carry in a loop. One other minor problem of the old HIMMO implementation was that it was assuming that both the key and the identifiers are longer than one byte. This is not always the case since we can have simple applications with maximum 256 nodes. We included additional code to deal with this issue as well.

For adding the **modifications for long identifiers**, we proceeded by introducing a new parameter in the configuration file, the identifiers size. Afterwards we updated all the memory locations size using this parameter, as explained in Section 3.1.1. We then modified all the loops and shifts to consider this new dimension and deal with the new form of key material illustrated in Figure 4.1. As detailed in [25], some of the bits are not considered in the key establishment process (the ones marked with zero in the figure), that is why they are not included in the key material delivered by the TTP. Our job was to include code that treats the last b bits separately (the blues ones in the figure), since that block can be smaller than the others.

When implementing the computation time measurement, we needed to deal with overflow interrupt handling. Since we were working in assembly, we needed to emulate the saving on the stack of performed by higher level languages for local variables. Therefore, at the beginning of the interrupt handling routine we were pushing on the stack the values of the registers we were using in the routine and just before the return instruction we were popping them.

For the case of ***HIMMO Transformation***, described in Section 3.1.1, we implemented the node operational phase from scratch. It is very similar to the previously augmented HIMMO implementation

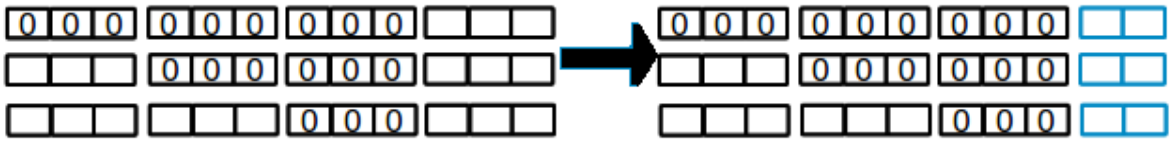

Figure 4.1: Key Material for HIMMO and HIMMO-LI

since both uses the special form of $N = 2^{(\alpha+1)B+b} - 1$, hence they only require additions, subtractions, multiplications and shift operations for long numbers. The shifts were the simplest among them and they required only the additions with a constant for memory addresses. For addition and subtraction, it requires only four 8-bit registers. In two of them the current bytes of the operands are loaded, in the third the result of the addition is temporarily stored, while the fourth stores the carry. After each step the corresponding result byte is stored in memory, while the carry is transported to the next iteration. This operation is very simple and its computation time is mostly influenced by the memory accesses. In the case of our 8-bit processor, we needed to make one access after each step of the computation, but in the case of a more powerful processor the number of accesses can be reduced by using a cache memory. For multiplying big numbers, the schoolbook method was used. It consists of multiplications of one byte of the multiplier with the multiplicand bytes one by one followed by an addition to the corresponding result memory location. The process is repeated for all the bytes of the multiplier, performing a byte shift operation after each step. This method has the advantage of simplicity, but it can be replaced by faster algorithms such as Karatsuba [19] or Toom-Cook [12] for obtaining an important reduction of the computation time.

Porting the current assembly code to a different CPU architecture would be a really challenging task, since we are relying on the particularities of the AVR controllers. However, porting it to a CPU from the same family, but with a larger CPU word should be relatively easy. The only modifications required would be with respect to the loop increment used.

Chapter 5

Evaluation

In the current chapter, we present the evaluation of the implementation of our HIMMO-LI algorithms. We compare their performance in three different programming languages and we also observe the benefit of using one of them. We then analyze the results with respect to different parameters such as: number of parallel systems, lattice dimension or security bits. We also introduce our framework used to perform the experiments, which can be further reused for similar purposes.

5.1 Setup. Board. Testing Framework.

In all our experiments we used a STK600 board with an ATmega128L microprocessor. This processor operates at 8 Mhz and has 128 Kbytes of flash available (program memory), 4 Kbytes of SRAM (data memory) and 4 Kbytes of EEPROM. For debugging, we used an AVR Dragon board and for programming we used the JTAG interface of the microprocessor. In the initial phase of our experiments, we used ATmelStudio to program the board manually and we observed the output on the 8 LEDs provided by the STK600 board. We analyzed the memory consumption values, both RAM and FLASH, in the compilation output provided by the IDE. For measuring the computation time values we used TCNT1, one of the hardware timers available in ATmega128L. This is a 16 bit register that can be configured with different prescaling values to obtain the desired accuracy for the measured time. We also captured the timer register overflow interrupt for incrementing an additional 16 bit variable. Considering the microprocessor's frequency, without using the prescaling functionality, we obtain an upper bound for our time measurements at around 536 seconds. In the cases when we wanted to measure experiments that take longer than this value, we used a higher prescaling value for the timer. Having in mind that the only available output in the STK board are the 8 LEDs and we wanted to output 32 bits (16 for overflow counter variable and 16 for the timer), we needed to run a certain experiment at least four times in order to observe the number of clocks, and therefore the time required for the computation. An additional concern was the correctness of the computation and therefore, we had to output the bits of the key as well on the available 8 bits. In conclusion, for 64 bit key, we needed to run the experiment 12 times, observe each time the binary 8 bit output on the LEDs, and store them carefully, in the right order. In the end we were comparing all the obtained bits of the key with the expected ones and convert the 32 bits for the number of cycles in decimal and then in milliseconds using the value for the CPU frequency. This was a laborious, time consuming and error prone process and we realized early that we need to automate it.

The first step towards full automation was to avoid the usage of the 8-bit LED output for printing the results. To overcome this problem we stored the obtained values for the key and for the number of cycles in consecutive EEPROM memory locations. We then, used the AVR dragon and AtmelStudio's debug mode to jump to the last instruction of the HIMMO computation and we observed the values stored in the non-volatile memory. However, this was not enough, since a human agent was still

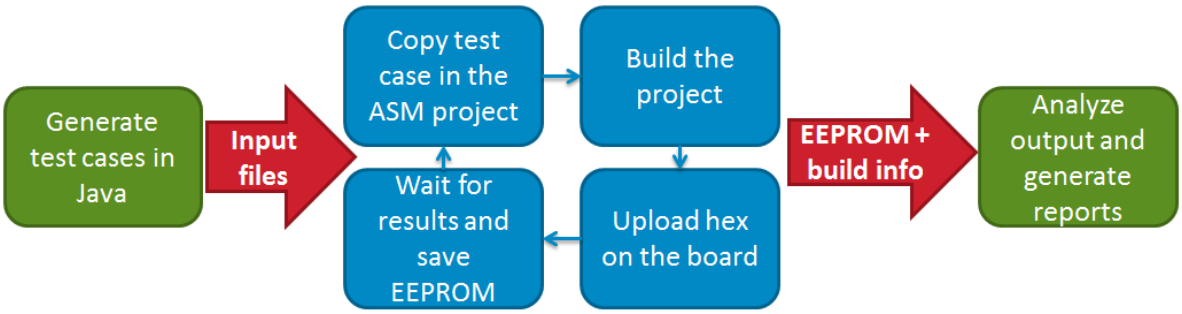

Figure 5.1: Testing and data collection framework

needed to copy the input files, run the experiments manually and then inspect the EEPROM memory to observe the correctness of the computation and the running time.

To overcome these limitations we created a batch script using the command line tools provided by the IDE. However, the scarce documentation of similar uses of these tools made the automation process very difficult. The result script, is plotted in Listing 5.1 and is capable of running a huge number of test scenarios without any human intervention. The script assumes that in the folder input exists a directory for each experiment and it contains the input files for the analyzed implementation. In Listing 5.1 we run assembly experiments, but we designed similar scripts for the C implementation as well. As it can be observed, the steps are roughly the following: copy the input files in the project, recompile the project and save the compiled output in a file corresponding to the current experiment, erase and program the board, wait a number of seconds and finally save a snapshot of the EEPROM memory in the output folder of the experiment.

Listing 5.1: Automation script used to run multiple experiments

```
1  for /l %%x in (1, 1, %%NO_TESTS) do (
2     mkdir output\%%x
3     copy input\%%x\config.asm HIMMO-ASM-LI\
4     copy input\%%x\working-params.asm HIMMO-ASM-LI\
5     atmelstudio.exe HIMMO-ASM-LI\HIMMO-ASM-LI.asmproj /build debug /out
          output\%%x\build_output.txt
6     echo Saved output of build to file output\%%x\build_output.txt
7     atprogram -t avrdragon -i jtag -d ATmega128 chiperase
8     atprogram.exe -t avrdragon -i jtag -d ATmega128 program -f
          "HIMMO-ASM-LI\Debug\HIMMO-ASM-LI.hex" --verify
9     ping 190.0.213.2 -n 1 -w 3000 > nul
10    atprogram.exe -t avrdragon -i jtag -d ATmega128 read -ee -s 1024 -f
          output\%%x\memory_output.txt --format hex
11 )
```

The discussed steps of the automation script represent only the data acquisition part of our testing framework. These steps are colored blue in Figure 5.1 which represents a high level view of the process. However, there are two more parts of the process that need to be discussed: the generation of the test cases and the analysis of the the output files. For the test case generation part, we used an existing well tested implementation of HIMMO in Java to generate key material for nodes in different HIMMO settings. This key material was further written in the experiment folder in three separate formats: two header files to be included in the C implementation, two memory files for assembly implementation and two Java input files. In this way, a certain experiment can be executed on three different implementations, making it very easy to compare the three versions, and also being extremely useful in the debugging process. As mentioned earlier, special attention was paid on outputting the

different key material formats in accordance to the syntax of the programming language for which they were targeted. Failing to do so would have crashed the compile step in the data acquisition script. In the generation process of the test cases, a comment is added in all the files, containing the expected computed key. This value is calculated using the Java implementation and is further used in the last step of the framework. This last step is also a Java program that analyzes all the output files and generates a report which includes the HIMMO instance's parameters, the result of the test case: pass / fail, the computation time and the FLASH and RAM memory used. This information was further used to generate all the figures in the current chapter.

Our framework proved to be very effective allowing the evaluation of hundreds of test cases within an hour. We adapted also the previously existing assembly implementation and the C version to the structure of the framework, making it easier to compare the performance of all of them.

5.2 Evaluation of HIMMO-LI

In the first experiment that we performed, we wanted to assess the performance of the HIMMO-LI compared with the original implementation. We wanted to observe that for the case $b = B$ the two implementations perform the same way and no significant performance penalties were introduced by the extension. We also wanted to verify the theoretical expressions computed for HIMMO-LI. For our measurements, we used the previously existing HIMMO assembly implementation and the one in which we fixed several bugs and we included the long identifiers extension.

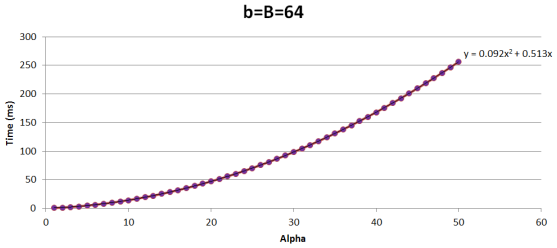

Figure 5.2: Quadratic dependency on the polynomial degree for the computation time, in the case $b = B = 64$

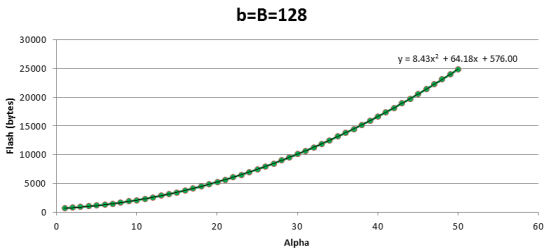

Figure 5.3: Quadratic dependency on the polynomial degree for flash memory, in the case $b = B = 128$

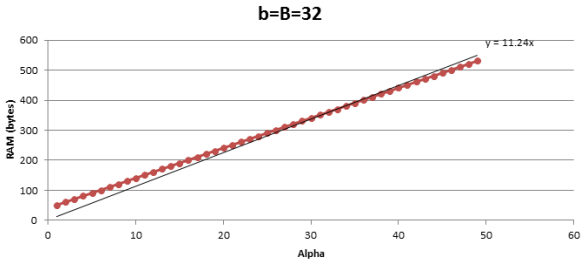

Figure 5.4: Linear dependency on the polynomial degree for RAM memory, in the case $b = B = 32$

In Figures 5.2, 5.3, 5.4 we observed the effect of α on the computation time, flash and RAM memory for HIMMO-LI by fixing $b = B$ for some standard values and varying α in the interval $[1, 50]$. As it can be observed, the results are consistent with the ones obtained in Chapter 3.1.1. We performed similar experiments and visualizations for $b = B = 8, 16, 32, 64, 128, 256$, observing smooth growth functions as the ones presented here. We also performed similar experiments to observe the effect of B and b on the performance of the algorithm. Some sample, such visualizations can be consulted

in Appendix A. In Figure 7.2 an interesting effect can be observed for small values of the FLASH memory: the assembler fills the key material with zeros in order to have an even number of bits. This is a particularity of the processor and it results in a stair-like plot, instead of a smooth linear function.

For comparing with the previous implementation, we had to modify it in order to fit in our data collection framework, but the modifications were only related to the way we collect the time measurements and to the way we output the collected values in the EEPROM memory. After performing these changes, we carried out 50 experiments, similar with the previously described ones, measuring the difference in CPU clocks, milliseconds, flash and RAM memory between the two implementations. Our results show no increase in the RAM memory consumption and a 36 bytes higher FLASH size due to the added code. Also, the computation time increased with an average of 0.19% for HIMMO-LI, which we consider negligible. Out of the 50 experiments, four of them failed, while for our new HIMMO-LI all the computations were performed as expected.

To conclude, our experiments confirmed the theoretical values for the computation time and memory consumption for HIMMO-LI. When compared with the previous implementation for HIMMO we observed only a slight increase in the code size and in computation time, but our implementation achieved 100% correctness as opposed to only 92% for the previous implementation. Finally, the success of all the tests performed during all the experiments presented in this chapter (more than 10,000) are a strong argument for the quality of our implementation.

5.3 HIMMO-LI Multiple Instances

In this experiment we aimed at achieving a 128-bit key using different configurations for HIMMO. Our main target was to give an intuition and visual representation for the trade off between a single big HIMMO system and multiple smaller ones. We noted with t the number of such identical systems used to achieve our target. Each system has the identifiers size equal with the key size ($b = B$).

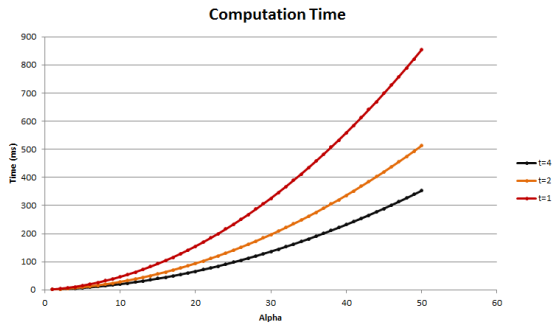

Figure 5.5: Required time for computing a 128-bit key using one, two or four parallel HIMMO systems

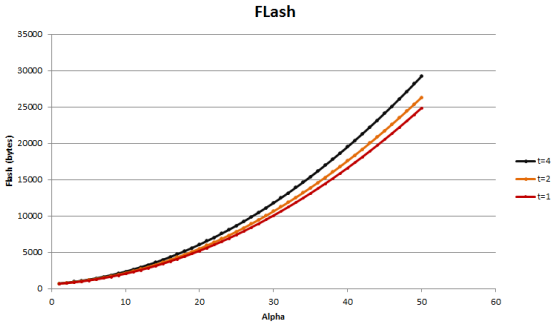

Figure 5.6: Flash consumption for computing a 128-bit key using one, two, or four parallel HIMMO systems

The results in Figures 5.5 and 5.6 are not surprising, considering the theoretical analysis. A raise in key size or identifiers size results in a quadratic increase in the computation time. This quadratic gain is much more significant than the linear one caused by multiple HIMMO systems in parallel. Therefore, from a computation point of view, it is more efficient to have multiple smaller systems than a big one. When analyzing the FLASH memory consumption, the situation is reversed: due to the linear dependency on both the key and the identifiers size, an increase in the memory requirements can be observed. This is because of the additional information that needs to be stored for each instance in addition to the key material. In a memory-optimized HIMMO instance, the flash requirements would be the same for all the three setups.

The conclusion of the current experiment is that smaller computation times can be achieved for

a certain key size, if more smaller keys are computed by parallel HIMMO-LI systems. However, this approach will increase the FLASH consumption compared with the single HIMMO-LI approach. No matter how appealing this gain in computation time is, it still needs to be analyzed from a security perspective as well. In a subsequent experiment we will check whether this optimization comes with a loss of security or not.

5.4 C versus Assembly Implementation

In this experiment we evaluated our C implementation by comparing it with the optimized assembly version. The main advantage of having a key agreement algorithm written in a high level programming language like C is the portability: it makes the process of porting the code on different devices much easier. After having a working C implementation, we generated, using the Java TTP, 10 test cases. Each of these tests was exported in two formats: one for the C implementation and another one for the assembly version. We then ran them independently on the controller and compared the results. Both the implementations passed all the tests, but the assembly version performed 160 times faster. This was surprising, since we expected a difference with one order of magnitude smaller. We identified the main cause for this poor performance, the quality of our big numbers library. This library is very naive and straight-forward, focused on correctness and maintenance instead of performance. For detailed results for the comparative tests, the reader is referred once again to Appendix A.

We conclude that our C implementation performs rather poor in terms of time, but it offers a better portability, giving the opportunity to easily implement small application prototypes using HIMMO. We also believe that by trading some code simplicity and putting some additional effort, the performance of the C implementation could be boosted significantly.

5.5 More is Less versus HIMMO Transformation

In this experiment we made a practical evaluation of HIMMO Transformation (described in Section 3.2.3). We changed our assembly implementation and the TTP to generate and work with the transformed key material. Afterwards we performed 200 test cases: 100 for the *More is Less* and 100 for *HIMMO Transformation*. The test cases corresponded to $b = B = 8$, $b = B = 16$ and $\alpha \in [1, 50]$. We measured the performance gain in terms of CPU clocks, time, flash and RAM. We obtained an average gain of 11% in time, 2.57% in flash and 7.25% in RAM. The CPU improvement is consistent with our theoretical findings $(\alpha b^2 + \alpha B^2) \backslash \text{CPUW}^2$. This is a consequence of treating both the parts of the key material together, instead of independently. In Figure 5.7, we illustrated the difference between the two implementations. Even though the gain grows linearly with α, the percentage growth is decreasing since the total computation time is dominated by the term with quadratic dependence on α.

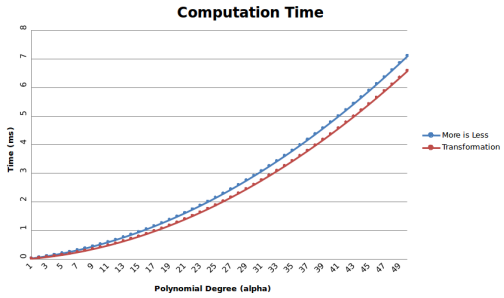

Figure 5.7: Computation time for *HIMMO More is Less* versus *HIMMO Transformation*

The conclusion of the current experiment is that *HIMMO Transformation* can be easily implemented in practice, boosting the computation time and both RAM and flash memory, simplifying

significantly the code. In Figures 5.8 and 5.9, we illustrated the gain in memory for this optimization. The increase in RAM is a result of reducing the number of temporary memory locations used by the algorithm, while the flash reduction is due to the smaller code size. Another cause for the reduction of the RAM memory requirements is that a B size block does not require storage on the node size anymore.

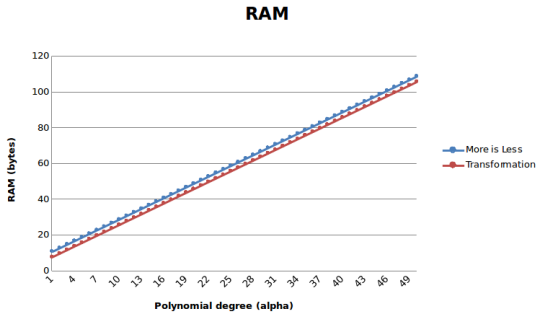

Figure 5.8: RAM memory requirements for *HIMMO More is Less* versus *HIMMO Transformation*

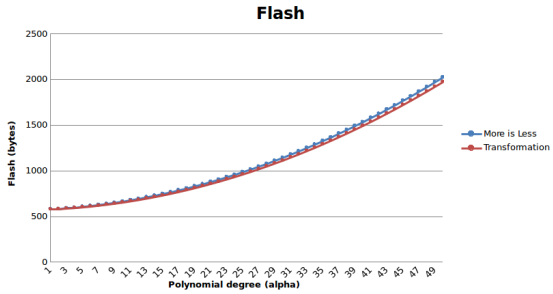

Figure 5.9: Flash memory requirements for *HIMMO More is Less* versus *HIMMO Transformation*

5.6 Lattice Dimension and Bits of Security

In our theoretical analysis from section 3.3, we deduced the expression for the lattice dimension for the case of HIMMO-LI and we showed that it includes a $B\backslash b$ ratio. A question that was left unanswered after computing this value is whether we can use this ratio to build systems with large difference between B and b and therefore, the same lattice dimension by using polynomials of smaller degree. This was a hard problem, since increasing the before mentioned ratio implies also an increase in the identifiers' size and/or decrease in the key size. This implies a penalty due to either the quadratic dependency on the identifiers size or due to the increased number of systems that need to work in parallel to achieve a longer key. Solving this trade-off was not obvious from a theoretical perspective, so we decided to verify it experimentally. Our setup included five types of systems: three with a ratio of one ($b = B = 40$, $b = B = 72$, $b = B = 136$), one with a ratio of two ($B = 136, b = 72$) and one with a ratio of four ($B = 136, b = 40$). We then varied the polynomial degree, α, which is another important parameter in the lattice dimension. We plotted the computation time for computing a 128 bit key function of attack lattice dimensions up to 1400 in Figure 5.10. Solving such lattices is way out of the power of the existing super computers [18]. As it can be seen, the faster solutions are the ones with low identifiers' size (since the performance depends quadratically on it), even though these setups use polynomials of higher degree in the computation of the key.

The conclusion of this experiment is that we can not exploit the $B\backslash b$ ratio in the attack lattice dimension expression to obtain systems that perform faster and guarantee the same level of security. This is mainly due to the fact that increasing the ratio means either a gain in the identifiers size or a reduction of the key size. Both of them have a greater impact on the computation time than the reduction of α. However, in the applications where we are forced to choose a large number of identifiers, we should consider shortening the key as a mean of increasing security.

Lattice dimension is a good way of estimating the security level, as shown by other schemes like NTRU which rely on similar considerations. For the case of NTRU, a lattice dimension grater than 500 is considered to guarantee a good security level [30]. We would also like to have a more standardized way to discuss about the security of different HIMMO systems. In [14], the number of security bits is estimated as:

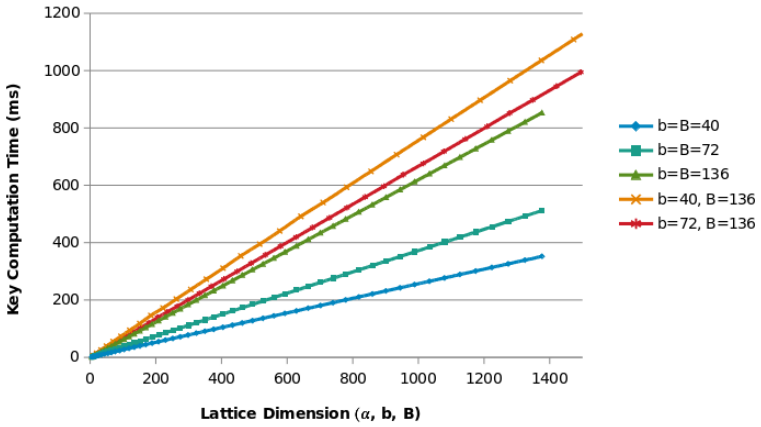

Figure 5.10: Computation time for calculating a 128-bit key by different HIMMO systems function of attack lattice dimension

$$\mathrm{SB} = \log\left(f(ld, el\_size)\right) + \log\left(f'(ld)\right). \tag{5.1}$$

We noted with ld the lattice dimension and function f has an order of growth of $\mathcal{O}(d^5 \log el\_size)$. The el_size in our case are the coefficients of the key material polynomial, being a $(\alpha+1)B+b$ bits long number. We think that approximating function f with its growth is a good enough estimation and gives the following expression for the number of security bits:

$$\mathrm{SB} = 5ld * 3\log\log\left(el\_size\right) + ld\log 1.099. \tag{5.2}$$

This approximation is a week one since we do not now the exact expression for f and the involved constants. However, for our purposes it serves well enough. We performed a set of experiments in which we estimated the performance of the HIMMO-LI algorithm for $B = b = SB$ setups. Therefore we identified the polynomial degree, for a certain $B = b$ system, for which the estimated number of security bits is equal to the key size 5.1. As shown in the previous experiment, the same level of security can be achieved by increasing the identifiers length instead of the key, however we did not consider this in the current scenario.

Key / Identifiers Size	Polynomial Degree
32	5
48	12
64	20
80	27
96	33
112	39
128	44
144	49
160	54

Table 5.1: Required polynomial degree for achieving a certain number of security bits

After identifying the polynomial degree for each key size, we performed some test key agreements using this value and observed the usual parameters: RAM, flash and computation time. NIST do not recommend using less than 80 bits of security, so the results for these values are purely illustrative, to

observe the growth rate. As it can be seen, achieving the security levels used in practice, a lightweight device will spend less than two seconds, which is considerably less than the existing key agreement schemes, as will be shown in the next chapter. The quadratic growth with respect to the newly introduced metric is not unexpected since our estimation has the significant factor dependent on B, b and α. The RAM consumption is also limited since it does not exceed 2.5 Kbytes which is easily affordable even for a constrained device. The problem comes from the FLASH consumption since its high reaches as much as 35 Kbytes, which is close to the limit of the processor we are working on.

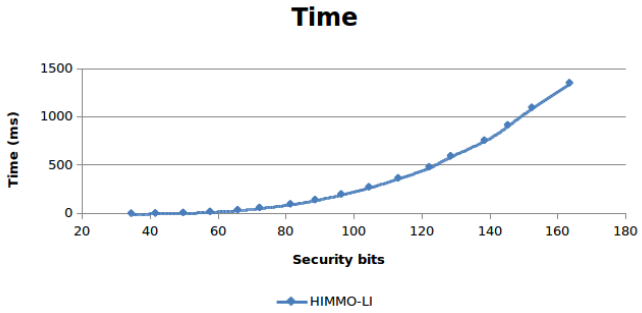

Figure 5.11: Computation time requirements for certain number of security bits

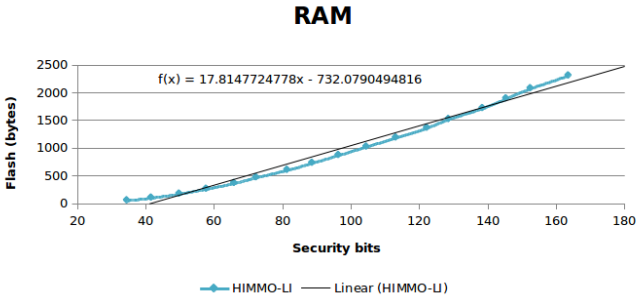

Figure 5.12: RAM memory requirements for certain number of security bits

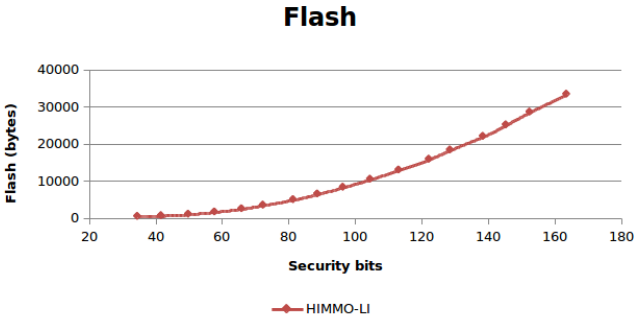

Figure 5.13: Flash memory requirements for certain number of security bits

Our findings of this last experiment confirm that HIMMO's main strengths are the low computation time and RAM memory required in the key agreement process. We will use the presented results in the next chapter to compare our system with similar existing solutions. However, we have to advice once more on the quality of our estimation, since it is only an initial proposal for relating lattice dimensions to security bits. Further research should be concentrated into obtaining a better approximation for the number of security bits expression in Equation 5.2

Chapter 6

Comparing HIMMO with Other Schemes

In this chapter, we first update the comparison between HIMMO and other schemes using our results. We denote with *HIMMO-LI-SB-XX* an instance of HIMMO corresponding to *XX* security bits, introduced in the last chapter. In Table 6.1 we updated the table from [16] with our results. Oliveira et al. [28] is a pairing scheme, Zhang et al. [34] is a predistribution scheme described in Section 2.1.4 and Liu et al. [21] is a random predistribution scheme based on the Blundo construction described Section 2.1.3. The results for these other three constructions are for 80 bits of security. Liu and Zhang schemes suffer from the low collusion resistance property that characterizes the polynomial schemes. In addition the Liu construction is probabilistic, establishing keys only with a certain probability.

Scheme	**CPU Cycles**	**Flash (bytes)**	**RAM (bytes)**
HIMMO-LI-SB-80	$7.8 * 10^5$	5196	621
HIMMO-LI-SB-112	$2.9 * 10^6$	13176	1205
HIMMO-LI-SB-144	$7.3 * 10^6$	25326	1909
Zhang et al. [34]	$9.6 * 10^5$	15005	325
Liu et al. [21]	$4 * 10^4$	416+656	20
Oliveira et al. [28]	$1.4 * 10^7$	38800	512

Table 6.1: Updated comparison between HIMMO and other related schemes

We also wanted to see how HIMMO-LI perform compared to standard encryption techniques (RSA, NTRU, ECC). For this, we used the results obtained on the same microcontroller in [26] and [17] in order to construct Table 6.2. The first four lines correspond to 80 NIST security bits, while the last

Scheme	**Time (seconds)**	**Flash (bytes)**	**RAM (bytes)**
HIMMO-LI-SB-80	0.098	5196	621
ECC-160	0.81	3682	282
RSA-1024	0.43+10.99	6292+1073	930+542
NTRU-251:3	0.032+0.078	970+786	804+618
HIMMO-LI-SB-112	0.367	13176	1205
ECC-224	2.19	4812	422
RSA-2048	1.94+83.26	2854+7736	1332+1853

Table 6.2: Comparison between HIMMO and standard encryption schemes

three to 112. As it can be observed, HIMMO is the fastest out of them, but it requires slightly more memories than ECC. The results for some of the schemes are represented in an $a + b$ format due to the different performances of the encryption (a) and decryption (b).

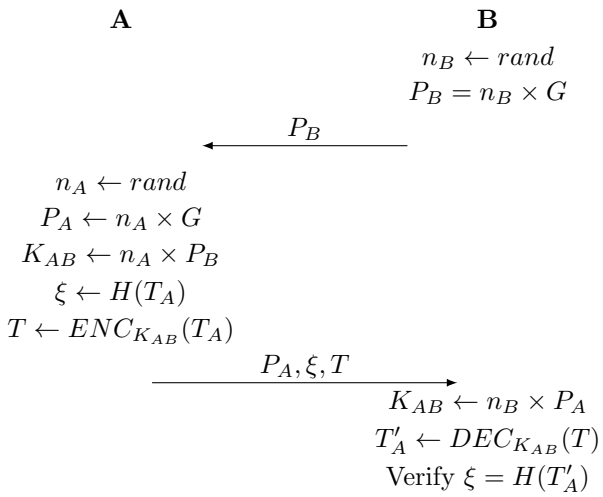

Figure 6.1: ECDH key establishment protocol. Notations: H - hash function, ENC/DEC - AES encryption and decryption, G - generator point on the curve, T_A - timestamp, K_{AB} - pairwise key, n_A, n_B - integer nonces, P_A, P_B - points on the curve.

Next, we will analyze the possibility of using HIMMO in a simple real life protocol. We imagined a two-party scenario in which two lightweight devices want to agree on a key. The security level is 80 bits for all the variants. The first such protocol is the schoolbook Elliptic Curve Diffie-Hellman (ECDH) which is shown in Figure 6.1. For encryption/decryption it uses a fast block cipher such as AES. In this protocol, one principal chooses a random nonce n_A, while the other picks n_B. They compute a point by adding the generator point repeatedly nonce times and send it to the other participant. After exchanging the two points they can apply their nonce independently in the same manner and obtain a common secret: a point on the curve. For key confirmation a fresh information is exchanged: in this case a timestamp. It is sent both in the form of a hashed value, and also encrypted with the shared key. In this way, by checking the decrypted value with the hash, one can draw the conclusion that the other participant is in possession of the key. However, we need to advise the reader that there is a clear distinction between key confirmation, presented in the Section 1.1, and principal authentication. This protocol, like most Diffie-Hellman ones, only offers the former one. This makes it vulnerable to a man-in-the-middle attack, allowing an adversary to rely the messages he receives from a legitimate user and pretend to be his messages.

In Figure 6.2, we sketch a simple solution for solving the limitation of the previous protocol, using digital signatures. The only difference between them is that the second one additionally offers node authentication. For enabling such a scenario, certificates need to be deployed in the network and also we need a way of transporting the public keys between the nodes. The authentication is done in a *MAC-then-Encrypt* fashion, that was presented in [3], and consists of signing a timestamp with the certificate in order to confirm the node's identity and then encrypt with the shared key and authenticate the key in this manner.

A similar construction can be adapted to include identity based encryption schemes and is presented in Figure 6.3. The difference is that the key should not be computed any more and the encryption is performed using the identity as a key. The scenario requires the distribution of the secret key, but it has the advantage that it does not require public key delivery. The first interaction, the delivery of the identity, can be avoided in most of the applications if the topology of the network is known and

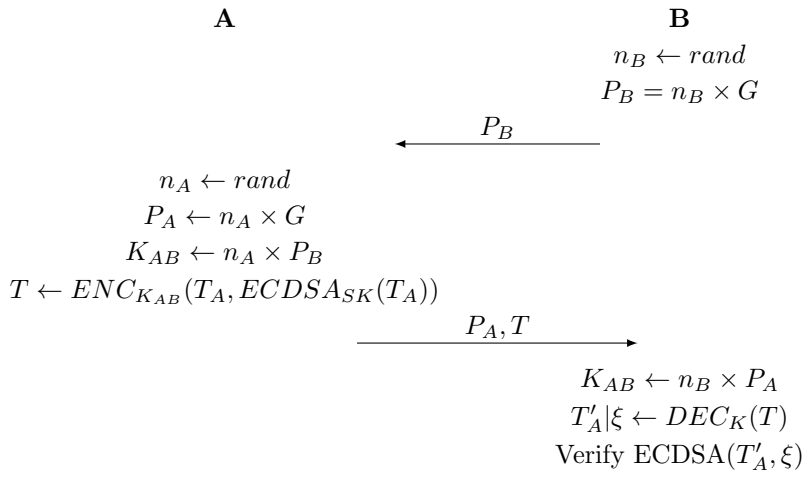

Figure 6.2: ECDH+ECDSA key establishment protocol. Notations: ECDSA - ECDSA signing algorithm, ENC/DEC - AES encryption and decryption, G - generator point on the curve, T_A - timestamp, K_{AB} - pairwise key, n_A, n_B - integer nonces, P_A, P_B - points on the curve.

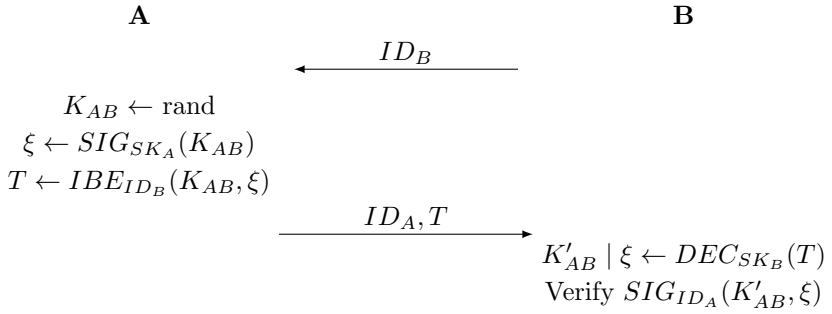

Figure 6.3: 2-pass key establishment protocol based on identity-based encryption (pairing). Notations: T_A - timestamp, ID_A, ID_B - identities of principals, H - hash function, K_{AB} - pairwise key, SIG - identity based signature algorithm, IBE/DEC - identity based encryption / decryption algorithm

the nodes know each other from deployment. However, there has to be a mechanism through which B triggers the key agreement. Transforming an identity based scheme in a signature construction is extremely simple and it reuses the same logic. In our toy protocol we noted with *IBE* an identity based scheme, possibly one using pairings and with *SIG* its corresponding signature scheme.

Our last protocol is the most interesting for the current work since it includes the HIMMO scheme depicted in Figure 6.4. The key confirmation is done in a MAC-and-Encrypt manner. Even though it looks very similar with the pairing solution, it is more efficient as it will be seen later since it uses a block cipher (AES) instead of the expensive identity based encryption. It provides both node and key authentication by using the predistributed keys. On the other hand, there are a couple of limitations that this protocol has. First, only one participant has control on the key and therefore can force the use of a compromised key. Second, it does not offer forward secrecy since the key generation in HIMMO does not use any information that identifies one specific session.

In Appendix B we present detailed computations for estimating the performance of each of the proposed protocols. We summarize the results of this work in Table 6.3. All the protocols require one round trip for key establishment in the form we proposed. However, as mentioned before, the identity based ones can be transformed in non interactive protocols if we consider that each pair of

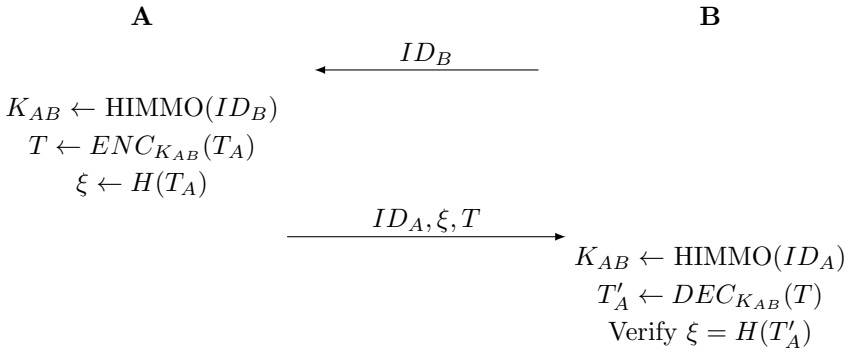

Figure 6.4: 2-pass key establishment protocol based on HIMMO-LI. Notations: H - hash function, ENC/DEC - AES encryption and decryption, T_A - timestamp, K_{AB} - pairwise key, HIMMO - HIMMO's registration phase.

nodes know each other's identity before we start the key agreement process. This is valid for HIMMO as well since the correction step can be avoided by delivering together with the key material some information (function, part of the keys) which will allow the computation of the key independently. With respect to the bandwidth, HIMMO seems to perform the best since it needs to exchange only two identities, an output of an AES encryption and a hash value. IBE and ECDH-ECDSA need to exchange signatures which are more expensive than the information exchanged in HIMMO (roughly two points on the elliptic curve).

HIMMO-LI also seems to perform the best with respect to the CPU time. The protocol consists of one AES encryption / decryption and two HIMMO key computations. ECDH consists of one AES encryption / decryption and two point multiplications. ECDH-ECDSA involves the same operations and in addition one digital signature generation and one verification. For IBE, we have one identity based encryption / decryption and one signature generation / verification.

Protocol	**Bandwidth**	**Round Trips**	**CPU Time (ms)**	**Security Properties**
ECDH	480	1	3960.70	key confirmation, forward secrecy
ECDH-ECDSA	704	1	11901.92	key confirmation, authentication, forward secrecy
HIMMO-AES	320	0/1	202.01	key confirmation, authentication
IBE	700+key size	0/1	>6213.6	key confirmation, authentication, forward secrecy

Table 6.3: Estimation of performance for our proposed protocols

The conclusion of the current chapter is that HIMMO could be successfully used in a real life scenario, having slightly better time and bandwidth consumption. We proposed a simple key agreement protocol that offers key confirmation and authentication which seems to work faster than the previous elliptic curve and identity based solutions. The main limitation of our scheme is the flash memory consumption, but this is becoming a less important requirement even for lightweight devices. The protocol we proposed does not provide forward secrecy, but it can be easily added by exchanging some random material between the two participants on the secure channel provided by the HIMMO scheme. Using this information and a one way function, a session key can be computed by the two principals. We did not add this element to our protocol since we wanted to have a simple construction and similar structure to the other proposed solutions. This simple adjustment will increase the number of round trips by one and slightly the bandwidth consumption.

Chapter 7

Conclusions. Limitations and Future Work

In this master thesis we presented *HIMMO with Long Identifiers* (HIMMO-LI), an extension of the HIMMO key establishment scheme that enhances the scalability of the original design. We adapt a series of algorithms designed for optimizing the computations of HIMMO and we present a theoretical analysis for them. We also deliver some considerations on the performance of the collusion attacks against a node in the case of HIMMO-LI.

We propose an automated framework for testing and benchmarking instances of HIMMO on ATmega128L microprocessor written either in C or in the assembly language. Using this, we performed a series of experiments with the aim of verifying existing assumptions and exploring different hypothesis. Our tests proves that the computation time and the flash consumption of HIMMO-LI has a quadratic dependency on the degree of the used polynomial and on the size of the identifiers and a linear dependency on the key size. We also found that the RAM memory depends linearly on the key size and on the degree of the polynomial, and quadratically on the identifiers size. Another important finding is that for minimizing the computation time one can use multiple parallel small HIMMO-LI systems, increasing slightly the FLASH and RAM memory consumption. We also showed that increasing the security of a HIMMO-LI system can be done, not just like in the case of other KPSs, by increasing the degree of the used polynomial, but also by increasing the ratio between the identifiers size and the key size. In certain scenarios one can take advantage of this observation, even though this second option is slightly more expensive. In the last part of the thesis we propose a rudimentary way of relating the dimension of the lattice used in the collusion attack against HIMMO-LI to the NIST security bits. Using this correspondence we compare the performance of HIMMO-LI with other key establishment scheme for the same level of security. Finally, we compare HIMMO-LI with the performance of standard encryption schemes as well.

One possible directions of extending the current work is implementing HIMMO-LI in a real life application and observe the nodes' interactions *in the wild*. So far, all the performance measurements were performed on isolated single running only our tests. Porting our results on different architectures can be an interesting idea as well. In this way, one can verify the difficulty of porting our code on other CPU instructions set. Testing our assembly code on another ATmega processor, with larger CPU word can be a good research direction also, since it will expose the *CPUW* factor in our theoretical analysis which was not observed in our experiments so far. Comparing the results at different CPU frequencies can be challenging as well.

An idea for further research is to take advantage of our observation that parallel smaller HIMMO-LI systems are faster than a single large one, by running tests in a parallel setup. Even the most lightweight devices start to have multiple parallel processing units which can be exploited for this purpose.

Speeding up the multiplications in the assembly implementation can have a great impact on the performance of the scheme. This can be done by using significantly faster algorithms than the classic paper and pencil version. Such methods are usually divide and conquer, inspired by Karatsuba [19]. An extreme way to optimize the key agreement process would be to implement HIMMO-LI on dedicated hardware. This can be done in the initial phases using reconfigurable hardware, in order to verify the feasibility of such an approach.

Tightening the gap between the performance of C and assembly implementations can be an interesting improvement too. Our C version is really slow, focusing on the code's simplicity, but having a fast HIMMO-LI written in C would be extremely useful, making it easy to port on other CPUs. The optimization should focus on the arithmetic operations since they dictate the overall performance of the key computation process. Another way of achieving portability and have good performance would be to port the code on a virtualization layer like LLVM. However this approach has the disadvantage of requiring the installation of additional software on the nodes, which is not desirable for lightweight devices that we worked with.

The protocols we proposed can as well be modified in various ways in further works. One way is to replace or relate them with standardized solutions, so that we are getting closer to what is used in practice. Also, a future version of the protocol we proposed for HIMMO-LI should guarantee forward secrecy. This is mainly because such a property is a requirement for most of the modern key establishment solutions, but also because in this way we take advantage of the low computation time of HIMMO.

To make our comparisons more accurate, future research should run the other key establishment schemes in our framework as well. In this way we avoid any possible measurement error that may have arose in any of the experiments we used the data from.

The most sensible point of our work is the link we proposed between the lattice dimension of a collusion attack and the number of security bits. It is only a naive estimation and future research must be performed in this direction. Nevertheless, we showed how such a link can be used to compare HIMMO with other key establishment schemes that have more clear security guarantees. The estimation can be improved by at least including some error tolerance in the formula to compensate for possible mistakes.

With all this in mind, we conclude that the current master thesis clarifies a lot of aspects regarding the performance of the HIMMO key establishment scheme, and of HIMMO-LI in particular. However, we do not give a complete image on the problem, but rather lay the foundation for a more sustained work, giving a lot of clear directions for research.

Appendix A

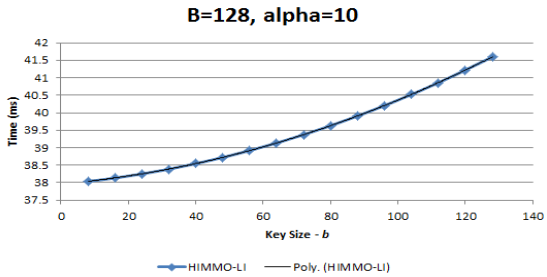

Figure 7.1: Quadratic dependency on key size for the computation time for $B = 128,\ \alpha = 10$

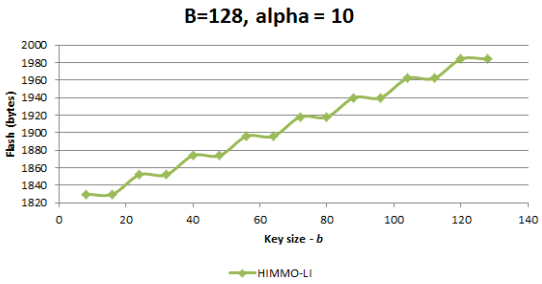

Figure 7.2: Linear dependency on key size for FLASH memory for $B = 128,\ \alpha = 10$

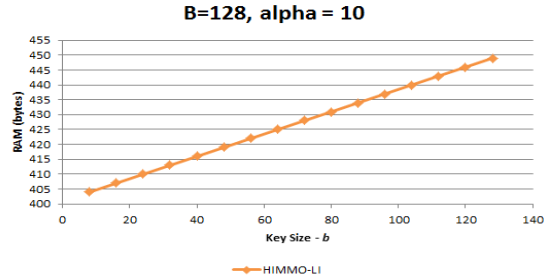

Figure 7.3: Linear dependency on key size for the RAM memory for $B = 128,\ \alpha = 10$

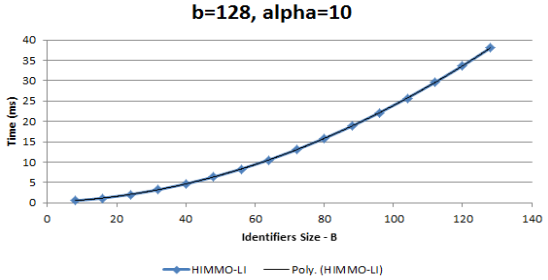

Figure 7.4: Quadratic dependency on identifiers size for computation time for $b = 128,\ \alpha = 10$

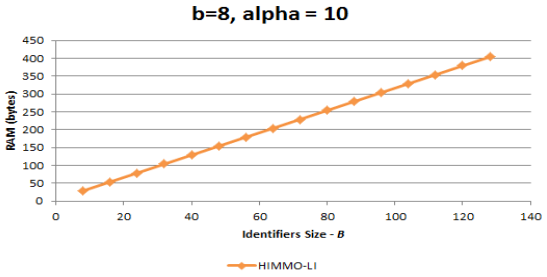

Figure 7.5: Linear dependency on identifiers size for the RAM memory for $b = 128,\ \alpha = 10$

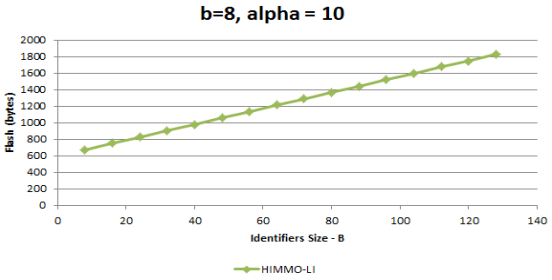

Figure 7.6: Linear dependency on identifiers size for FLASH memory for $b = 128,\ \alpha = 10$

Alpha	Result	Cycles	Time (ms	Flash (bytes)	RAM (bytes)	Result	Cycles	Time (ms)	Flash (bytes)	RAM (bytes)	DIFF CPU	Diff Flash	Diff RAM	CPU Percentage
1	PASSED	13991	1.748875	832	161	PASSED	13885	1.735625	796	161	106	36	0	0.7634137559
2	PASSED	31717	3.964625	912	193	PASSED	31521	3.940125	876	193	196	36	0	0.6218076838
3	PASSED	53859	6.732375	1008	225	PASSED	53573	6.696625	972	225	286	36	0	0.533851007
4	PASSED	80451	10.05638	1120	257	PASSED	80081	10.010125	1084	257	370	36	0	0.4620321924
5	PASSED	111425	13.92813	1248	289	PASSED	110965	13.870625	1212	289	460	36	0	0.4145451268
6	PASSED	146849	18.35613	1392	321	PASSED	146305	18.288125	1356	321	544	36	0	0.37182598
7	PASSED	186655	23.33188	1552	353	PASSED	186021	23.252625	1516	353	634	36	0	0.3408217352
8	PASSED	230911	28.86388	1728	385	PASSED	230193	28.774125	1692	385	718	36	0	0.311912178
9	PASSED	279583	34.94788	1920	417	PASSED	278781	34.847625	1884	417	802	36	0	0.2876810113
10	PASSED	332671	41.58388	2128	449	PASSED	331785	41.473125	2092	449	886	36	0	0.2670404027
11	PASSED	390141	48.76763	2352	481	PASSED	389165	48.645625	2316	481	976	36	0	0.2507933653
12	PASSED	452061	56.50763	2592	513	PASSED	451001	56.375125	2556	513	1060	36	0	0.2350327383
13	PASSED	518397	64.79963	2848	545	PASSED	517253	64.656625	2812	545	1144	36	0	0.2211683644
14	PASSED	589191	73.64888	3120	577	PASSED	587921	73.490125	3084	577	1270	36	0	0.216015417
15	PASSED	664396	83.0495	3408	609	PASSED	663045	82.880625	3372	609	1351	36	0	0.2037569094
16	PASSED	743983	92.99788	3712	641	PASSED	742545	92.818125	3676	641	1438	36	0	0.1936582968
17	PASSED	827935	103.4919	4032	673	PASSED	826461	103.307625	3996	673	1474	36	0	0.1783508236
18	PASSED	916351	114.5439	4368	705	PASSED	914793	114.349125	4332	705	1558	36	0	0.1703117536
19	PASSED	1009217	126.1521	4720	737	PASSED	1007581	125.947625	4684	737	1636	36	0	0.16236908
20	PASSED	1106465	138.3081	5088	769	PASSED	1104745	138.093125	5052	769	1720	36	0	0.1556920375
21	PASSED	1208163	151.0204	5472	801	PASSED	1206365	150.795625	5436	801	1798	36	0	0.1490427856
22	PASSED	1314277	164.2846	5872	833	PASSED	1312401	164.050125	5836	833	1876	36	0	0.1429441154
23	PASSED	1424773	178.0966	6288	865	PASSED	1422813	177.851625	6252	865	1960	36	0	0.1377552777
24	PASSED	1539719	192.4649	6720	897	PASSED	1537681	192.210125	6684	897	2038	36	0	0.1325372428
25	PASSED	1663177	207.8971	7168	929	FAILED	1656965	207.120625	7132	929	6212	36	0	0.3749023063
26	PASSED	1782859	222.8574	7632	961	PASSED	1780665	222.583125	7596	961	2194	36	0	0.1232123954
27	PASSED	1911134	238.8918	8112	993	PASSED	1908781	238.597625	8076	993	2353	36	0	0.1232723922
28	PASSED	2043747	255.4684	8608	1025	PASSED	2041313	255.164125	8572	1025	2434	36	0	0.1192369813
29	PASSED	2180785	272.5981	9120	1057	FAILED	2178261	272.282625	9084	1057	2524	36	0	0.1158722486
30	PASSED	2322221	290.2776	9648	1089	PASSED	2319625	289.953125	9612	1089	2596	36	0	0.1119146414
31	PASSED	2468082	308.5103	10192	1121	PASSED	2465405	308.175625	10156	1121	2677	36	0	0.1085825655
32	PASSED	2618359	327.2949	10752	1153	PASSED	2615601	326.950125	10716	1153	2758	36	0	0.1054442172
33	PASSED	2773086	346.6358	11328	1185	PASSED	2770253	346.281625	11292	1185	2833	36	0	0.1022650278
34	PASSED	2932093	366.5116	11920	1217	PASSED	2929281	366.160125	11884	1217	2812	36	0	0.095996253
35	PASSED	3095649	386.9561	12528	1249	PASSED	3092765	386.595625	12492	1249	2884	36	0	0.0932498913
36	PASSED	3269475	408.6844	13152	1281	PASSED	3260625	407.578125	13116	1281	8850	36	0	0.2714203565
37	PASSED	3435975	429.4969	13792	1313	PASSED	3432941	429.117625	13756	1313	3034	36	0	0.0883790313
38	PASSED	3612893	451.6116	14448	1345	PASSED	3609673	451.209125	14412	1345	3220	36	0	0.0892047562
39	PASSED	3794082	474.2603	15120	1377	PASSED	3790781	473.847625	15084	1377	3301	36	0	0.0870796809
40	PASSED	3979730	497.4663	15808	1409	PASSED	3976345	497.043125	15772	1409	3385	36	0	0.0851284282
41	PASSED	4169776	521.222	16512	1441	PASSED	4166325	520.790625	22324	1665	3451	-5812	-224	0.0828307921
42	PASSED	4364256	545.532	17232	1473	PASSED	4360721	545.090125	17196	1473	3535	36	0	0.0810645762
43	PASSED	4563005	570.3756	17968	1505	PASSED	4559533	569.941625	17932	1505	3472	36	0	0.0761481494
44	PASSED	4766305	595.7881	18720	1537	PASSED	4762761	595.345125	18684	1537	3544	36	0	0.0744106202
45	PASSED	4974021	621.7526	19488	1569	PASSED	4970405	621.300625	19452	1569	3616	36	0	0.0727506109
46	PASSED	5186187	648.2734	20272	1601	FAILED	5182505	647.813125	20236	1601	3682	36	0	0.0710467235
47	PASSED	5402735	675.3419	21072	1633	PASSED	5398981	674.872625	21036	1633	3754	36	0	0.0695316394
48	PASSED	5623699	702.9624	21888	1665	PASSED	5619873	702.484125	21852	1665	3826	36	0	0.0680798303
49	PASSED	5849113	731.1391	22720	1697	FAILED	5845221	730.652625	22684	1697	3892	36	0	0.0665843088
50	PASSED	6078909	759.8636	23568	1729	PASSED	6074945	759.368125	23532	1729	3964	36	0	0.0652516196
														0.1949443865

Figure 7.7: Comparative evaluation of HIMMO-LI and the previous HIMMO implementation

Result	b	B	Alpha	m	Cycles	Time	Flash			
ASM Implementation	32	32	4	2	7473	0.934125	784			
	40	40	7	2	23643	2.955375	950			
	48	48	10	2	56161	7.020125	1212			
	56	56	14	2	128627	16.07838	1742			
	64	64	18	2	252563	31.57038	2508			
	72	72	22	2	447465	55.93313	3594			
	80	80	26	2	735869	91.98363	5004			
	88	88	29	2	1074016	134.252	6476			
	96	96	32	2	1517373	189.6716	8208			
	104	104	35	2	2085857	260.7321	10298	DIFF Cycles	Diff Flash	Times slower
C Implementation	32	32	4	2	871389	108.9236	5010	863916	4226	116.6049779205
	40	40	7	2	3048045	381.0056	5226	3024402	4276	128.9195533562
	48	48	10	2	8570512	1071.314	5658	8514351	4446	152.6061145635
	56	56	14	2	19806330	2475.791	6548	19677703	4806	153.9826785978
	64	64	18	2	39725153	4965.644	7930	39472590	5422	157.2880944556
	72	72	22	2	71373718	8921.715	9840	70926253	6246	159.506817293
	80	80	26	2	1.3E+008	16266.24	12438	129394042	7434	176.83841961
	88	88	29	2	2.1E+008	25625.65	15110	203931159	8634	190.8772076021
	96	96	32	2	2.9E+008	36216.5	18342	288214644	10134	190.9431741569
	104	104	35	2	3.7E+008	45715.17	22198	363635514	11900	175.3338656485
Performance Penalties								112765457	6752.4	160.2900903204

Figure 7.8: Comparative evaluation between HIMMO-LI C implementation and the assembly version

Appendix B

For each of the four protocols, we estimate the bandwidth and computation time using the results available in the literature. Most of this data was collected on the same setup as ours, but sometimes the authors use a MICAz chip which is basically the same microcontroller, but used together with a radio chip. We express the bandwidth as the total number of bytes exchanged by the participants in the process. The time is expressed in seconds and it refers only to the computation time, therefore the transmission and message conversion time is ignored.

ECDH Protocol

$$\begin{aligned} \text{BWIDTH} &= 2 \cdot \text{EC\_POINT} + \text{BLOCK\_SIZE} + \text{HASH\_SIZE} && (7.1)\\ &= 2 \cdot 160 + 128 + 32 = 480 && (7.2) \end{aligned}$$

$$\begin{aligned} \text{CPUTIME} &= \text{ECDH\_INIT} + \text{ECDH} + \text{AES\_ENC} + \text{AES\_DEC} && (7.3)\\ &= 1838.37 + 2117.43 + 1.993 + 2.901 = 3960.69 && (7.4) \end{aligned}$$

The values for AES encryption decryption on ATmega128L were taken from [29], which is currently one of the fastest software implementation on 8 bits. The elliptic curve values were taken from [20]. The computation and verification of the hash function was ignored since it is negligible compared with the elliptic curve operations. The hash can also be replaced by a simple checksum, depending on the security requirements of the application.

ECDH-ECDSA Protocol

$$\begin{aligned} \text{BWIDTH} &= 2 \cdot \text{EC\_POINT} + \text{BLOCK\_SIZE} \cdot \text{COUNT} && (7.5)\\ &= 2 \cdot 160 + 128 \cdot 3 = 704 && (7.6) \end{aligned}$$

$$\text{CPUTIME} = \text{ECDH\_INIT} + \text{ECDSA\_INIT} + \text{ECDH} + \text{SIG} + \text{SIG\_VF} + \text{AES\_ENC} + \text{AES\_DEC} \quad (7.7)$$

$$= 1838.37 + 3493.36 + 2117.43 + 2001.62 + 2436.46 + 3 \cdot 1.993 + 3 \cdot 2.901 = 11901.92 \quad (7.8)$$

The time values are taken, as in the case of the previous protocol, from [29] and [20], but also from [13] for the digital signature's time. One ECSDA signature consists of two points on the curve, in our case 160 bits long each. The signature together with the timestamp has a size of 384 bits, hence we need to use the AES encryption / decryption three times.

HIMMO Protocol

$$\text{BWIDTH} = 2 \cdot \text{ID\_SIZE} + \text{BLOCK\_SIZE} + \text{HASH\_SIZE} \tag{7.9}$$
$$= 2 \cdot 80 + 128 + 32 = 320 \tag{7.10}$$

$$\text{CPUTIME} = 2 \cdot \text{HIMMO\_KE} + 2 \cdot \text{HASH} + \text{AES\_ENC} + \text{AES\_DEC} \tag{7.11}$$
$$= 2 \cdot 98.560 + 2.901 + 1.993 = 202.01 \tag{7.12}$$

The cost of the proposed HIMMO protocol was estimated easily using our results from Section 5.6 and the ones in [29]. Again, the hashing time was ignored, since a simple checksum can be implemented instead. The computation time corresponds to HIMMO-LI-SB-80 with a polynomial of degree 27.

IBE Protocol

$$\text{BWIDTH} = 2 \cdot \text{ID\_SIZE} + \text{IBE\_ENC\_SIZE} \tag{7.13}$$
$$= 2 \cdot 80 + 270 + 80 = 510 \tag{7.14}$$

$$\text{CPUTIME} = \text{SIG} + \text{VF\_SIG} + \text{IBE\_ENC} + \text{IBE\_DEC} \tag{7.15}$$
$$> \text{POINT\_MULT} + \text{POINT\_MULT} + \text{PAIRING} + \text{POINT\_MULT} + \text{PAIRING} + \text{PAIRING} \tag{7.16}$$
$$= 2066 + 4.02 + 4.02 + 4.02 + 2066 + 2066 = 6213.6 \tag{7.17}$$

In [33], a short signature scheme for pairing cryptography is proposed. It consists of only one point in the first group of the pairing. In our case this is 270 bits ([28]) and together with the key which is 80 bits as well gives a total size for the encrypted message of 350 bits, since the size of the encryption is equal to the size of the message in a Boneh-Franklin scheme which we consider in the current protocol. Together with two IDs, it gives a total bandwidth of 510 bits.

Estimating the computation time for the IBE protocol is very challenging and here we give only a lower bound to show that it is significantly slower than HIMMO-LI. According to [33], signing is equivalent to one inversion and one point multiplication, while the verification consist of a pairing, a point addition and a point multiplication. We consider only the pairing and the multiplication since they are more computational significant. A Boneh-Franklin encryption requires one point multiplication and one pairing, while a decryption implies only one pairing. We consider the computation time for multiplication 4.02 obtained in [28] and the pairing as 2066 from [22].

Bibliography

[1] Martin Albrecht, Craig Gentry, Shai Halevi, and Jonathan Katz. Attacking cryptographic schemes based on perturbation polynomials. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 1–10. ACM, 2009.

[2] Elaine Barker, Don Johnson, and Miles Smid. Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. *NIST Special Publication*, pages 800–56A, 2007.

[3] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT 2000*, pages 531–545. Springer, 2000.

[4] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - CRYPTO'93*, pages 232–249. Springer, 1994.

[5] Rolf Blom. Non-public key distribution. In *Crypto*, pages 231–236, 1982.

[6] Rolf Blom. An optimal class of symmetric key generation systems. In *Advances in Cryptology*, pages 335–338. Springer, 1985.

[7] Carlo Blundo, Alfredo De Santis, Amir Herzberg, Shay Kutten, Ugo Vaccaro, and Moti Yung. Perfectly-secure key distribution for dynamic conferences. In *Advances in Cryptology - CRYPTO'92*, pages 471–486. Springer, 1993.

[8] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Advances in Cryptology—CRYPTO 2001*, pages 213–229. Springer, 2001.

[9] Colin A Boyd and Anish Mathuria. *Protocols for key establishment and authentication*. Springer-Verlag New York, Inc., 2003.

[10] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology - EUROCRYPT 2001*, pages 453–474. Springer, 2001.

[11] Kim-Kwang Raymond Choo. *Secure Key Establishment*. Springer Science+Business Media, 2009.

[12] Stephen A Cook and Stål O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, pages 291–314, 1969.

[13] Benedikt Driessen, Axel Poschmann, and Christof Paar. Comparison of innovative signature algorithms for WSNs. In *Proceedings of the first ACM conference on Wireless network security*, pages 30–35. ACM, 2008.

[14] Nicolas Gama and Phong Q Nguyen. Predicting lattice reduction. In *Advances in Cryptology–EUROCRYPT 2008*, pages 31–51. Springer, 2008.

[15] Oscar Garcia-Morchon, Ronald Rietman, Igor E Shparlinski, and Ludo Tolhuizen. Interpolation and approximation of polynomials in finite fields over a short interval from noisy values. *arXiv preprint arXiv:1401.1331*, 2014.

[16] Oscar Garcia-Morchon, Ronald Rietman, Ludo Tolhuizen, Domingo Gomez-Perez, Jaime Gutierrez, and Santos Merino del Pozo. An ultra-lightweight ID-based pairwise key establishment scheme aiming at full collusion resistance. *ePrint Archive, Report*, 618, 2012.

[17] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004.

[18] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *Coding and Cryptology*, pages 159–190. Springer, 2011.

[19] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[20] An Liu and Peng Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*, pages 245–256. IEEE, 2008.

[21] Donggang Liu, Peng Ning, and Rongfang Li. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):41–77, 2005.

[22] Julio López, D Aranha, D Câmara, R Dahab, L Oliveira, and C Lopes. Fast implementation of elliptic curve cryptography and pairing computation for sensor networks. In *The 13th Workshop on Elliptic Curve Cryptography (ECC 2009), Available at http://ecc. math. ucalgary. ca/sites/ecc. math. ucalgary. ca/files/u5/Lopez_ECC2009. pdf.*

[23] Tsutomu Matsumoto and Hideki Imai. On the key predistribution system: A practical solution to the key distribution problem. In *Crypto*, volume 87, pages 185–193. Springer, 1987.

[24] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2010.

[25] Santos Merino del Pozo. *Efficient implementation of symmetric-key generation systems*. PhD thesis, Universidad de Cantabria, 2013.

[26] Mariano Monteverde Giacomino. NTRU software implementation for constrained devices. 2011.

[27] B Clifford Neuman and Stuart G Stubblebine. A note on the use of timestamps as nonces. *ACM SIGOPS Operating Systems Review*, 27(2):10–14, 1993.

[28] Leonardo B Oliveira, Diego F Aranha, Conrado PL Gouvêa, Michael Scott, Danilo F Câmara, Julio López, and Ricardo Dahab. Tinypbc: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. *Computer Communications*, 34(3):485–493, 2011.

[29] Dag Arne Osvik, Joppe W Bos, Deian Stefan, and David Canright. Fast software AES encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.

[30] Daniel Rosenberg. NTRUE*ncrypt and Lattice Attacks*. PhD thesis, Royal Institute of Technology, 2010.

[31] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Advances in cryptology*, pages 47–53. Springer, 1985.

[32] Igor Shparlinski and Arne Winterhof. Noisy interpolation of sparse polynomials in finite fields. *Applicable Algebra in Engineering, Communication and Computing*, 16(5):307–317, 2005.

[33] Fangguo Zhang, Reihaneh Safavi-Naini, and Willy Susilo. An efficient signature scheme from bilinear pairings and its applications. In *Public Key Cryptography–PKC 2004*, pages 277–290. Springer, 2004.

[34] Wensheng Zhang, Minh Tran, Sencun Zhu, and Guohong Cao. A random perturbation-based scheme for pairwise key establishment in sensor networks. In *Proceedings of the 8th ACM international symposium on Mobile ad hoc networking and computing*, pages 90–99. ACM, 2007.