Evaluating the U-LP Cryptosystem in Practice

Evaluierung des U-LP Verschlüsselungsverfahrens in der Praxis
Bachelor-Thesis von Jannik Vieten aus Neuwied
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Florian Göpfert

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Kryptographie und Computeralgebra

Evaluating the U-LP Cryptosystem in Practice
Evaluierung des U-LP Verschlüsselungsverfahrens in der Praxis

Vorgelegte Bachelor-Thesis von Jannik Vieten aus Neuwied

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Florian Göpfert

Tag der Einreichung:

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jannik Vieten, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Datum Unterschrift

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Date Signature

Zusammenfassung

Seit vor einiger Zeit gezeigt wurde, dass (bisher theoretisch existente) Quantencomputer die meisten der heute gängigen asymmetrischen Verschlüsselungsverfahren brechen können, arbeiten viele Forscher an der Entwicklung neuer Verfahren, die nicht so stark von einem Durchbruch im Quantencomputing beeinflusst werden würden. Eines dieser Verfahren ist U-LP, das erste beweisbar sichere Verschlüsselungsverfahren, welches auf dem *learning with errors* (LWE) Problem basiert, und das den Fehler und das Geheimnis von einer Gleichverteilung generiert. In dieser Arbeit wird eine universelle Implementierung von U-LP in C präsentiert, die eine Bibliothek bildet, welche Funktionen zur Schlüsselerzeugung, Verschlüsselung, und Entschlüsselung bereitstellt (sowohl für die Standardvariante von U-LP als auch für die ring-LWE Variante). Die Verwendungszwecke für diese Implementierung liegen aktuell hauptsächlich im Test- und Evaluationsbereich. Die Eignung von Programmiersprachen für das Schreiben einer solchen Bibliothek wird diskutiert, die Programmierschnittstelle wird vorgestellt, und einige interessante Aspekte werden erläutert. Anschließend wird evaluiert, wie die Wahl von bestimmten Parametern das Laufzeitverhalten und die Größe der Datenstrukturen beeinflusst. Es wird U-LP mit LP verglichen, dem Verfahren, an das U-LP angelehnt ist. Schlussendlich wird gezeigt, wie U-LP durch die Verwendung der ring-LWE Variante und dem Einsatz von Parallelisierung optimiert werden kann.

Abstract

Since it was shown that (yet hypothetical) quantum-computers can break most of today's asymmetric encryption schemes, many researchers are working on new cryptosystems, which would not be critically affected by a breakthrough in quantum computing. One of these is U-LP, which is the first provably secure encryption scheme based on the *learning with errors* (LWE) problem, where noise and secret are drawn from a uniform distribution. In this thesis, I present a universal implementation of U-LP in the C programming language, providing a library which offers key generation, encryption, and decryption capabilities for the standard variant of U-LP, as well as for the ring-LWE variant. The use cases for this library are at the moment mainly testing and evaluation purposes. In this thesis, I discuss the suitability of programming languages for writing such a library, describe the API, and write about issues of the implementation, which might be of interest. Afterwards, I evaluate how specific parameter choices influence the runtime behavior and the size of the data structures. Beside that, I name some difficulties which arise when using U-LP in practice. I compare U-LP with LP, the encryption scheme from which U-LP is adapted. Finally I show and discuss how U-LP can be optimized and accelerated by using the ring-LWE variant and parallelization.

Contents

List of Tables

List of Code Listings

1 Introduction

Regarding the history of cryptography, the usage of encryption changed fundamentally in the last 30 years. Early cryptosystems where originally developed for military usage, intended to assure the confidentiality of the command chain. With the rise of interconnected consumer products and a large market for online businesses, more and more encryption schemes were developed and standardized for everyday use, aimed to protect against different kinds of threats. This includes but is not limited to data theft, financial fraud and governmental wiretapping. While symmetric schemes like AES or Twofish are used for the data encryption itself, the often less efficient asymmetric schemes are suitable to solve the key exchange problem by encrypting a symmetric session key. Classical cryptosystems like RSA, which rely on the hardness of the factorization problem, or those based on the computational Diffie-Hellman problem, are well established in Internet standards, but require an ongoing increase of parameter sizes due to the development of more powerful computing devices. To overcome the high computational load, caused by large parameters of classical schemes, elliptic-curve cryptography was introduced, bringing lower encryption overhead by comparable security levels. Unfortunately, even elliptic-curve cryptography is based on the discrete logarithm problem, which, as well as the factorization problem, is solvable with quantum computers in polynomial time, as stated by Shor [Sho97]. Today, it is still unclear, if it will become possible to build sufficiently large quantum computers to break nowadays common asymmetric encryption algorithms. However, several research groups are steadily working on such machines. Developing public-key encryption schemes which are (probably) not affected by quantum computers is reasonable for the case they succeed. Several classes of such algorithms already exist, for example hash-based, code-based, lattice-based, and multivariate-quadratic-equations cryptography. For a broad overview of the so called *post-quantum cryptography* see the book from Bernstein, Buchmann, and Dahmen [BBD09]. Anyhow, most of the existing encryption schemes are rather inefficient, so further research and development is still necessary.

One lattice-based approach is the LP scheme by Lindner and Peikert as proposed in [LP11]. It relies on the hardness of the *learning with errors* (LWE) problem and uses discrete Gaussian sampling for noise and secret generation. Learning with errors is basically about computing the inner products of a fixed secret vector and several random vectors, and adding some noise to this inner products afterwards. The problem is to recover the secret vector, given access to the chosen random vectors and the above mentioned results. While LP is relatively efficient, dealing with a discrete Gaussian distribution in practice is sometimes cumbersome. To overcome these difficulties, Cabarcas, Göpfert, and Weiden adapted the LP scheme and presented a security proof for noise and secret, sampled uniformly at random [CGW14]. The proof is based on the work of Micciancio and Peikert [MP13]. The so called U-LP cryptosystem is subject of this thesis which is structured as follows: At first, in Section 2, I provide some background on lattice-based cryptography, the learning with errors problem, and the encryption schemes LP and U-LP. The ring-LWE variants of LP and U-LP are also described. In Section 3, my implementation of the U-LP cryptosystem is presented, including the rationale for the programming language used. After that, in Section 4, I compare U-LP and LP, highlight bottlenecks concerning runtime behavior and structure sizes, and show how the ring-LWE variant and parallelization can improve the situation. I conclude in Section 5.

2 Background

U-LP, as well as LP, belongs to the lattice-based encryption schemes and is based on the learning with errors problem. In this Section, I give a short overview about lattice-based cryptography, explain the LWE problem, and show how to make use of it in LP and U-LP.

2.1 Lattice-Based Cryptography

The term lattice-based cryptography refers to a set of encryption schemes based on lattices. In group theory, a lattice is a discrete subgroup of $\mathbb{R}^n$. Using a basis $B := (b_1, b_2, ..., b_n)$ where $b_i \in \mathbb{R}^n$ are linear independent vectors (similar to the basis of a vector space), a lattice is defined as follows:

$$L := \left\{ \sum_{i=1}^{n} k_i b_i \;\middle|\; k_i \in \mathbb{Z} \right\}$$

For use in cryptography it is handy to deal with integer numbers, so it is common to demand $b_i \in \mathbb{Z}$. Furthermore calculations are usually done modulo an integer q to obtain a finite set of representatives for an infinite number of vectors in the lattice.

Hard problems over lattices make this algebraic structure valuable in cryptography. Those problems are for example the *shortest vector problem* (SVP): find the shortest non-zero vector in the lattice, or the *closest vector problem* (CVP): given a vector not in the lattice, find the lattice element with the least distance to the given vector.

2.2 Learning with Errors

Another hard lattice problem is learning with errors [Reg09]: Let $n > 0$ be the dimension of the lattice, $q \geq 2$ the (usually prime) modulus, and χ a probability distribution on $\mathbb{Z}_q$. For a fixed (and secret) vector $s \in \mathbb{Z}_q^n$ denote with $A_{s,\chi}$ the probability distribution gained by choosing a vector $a \in \mathbb{Z}_q^n$ uniformly at random, choosing error $e \in \mathbb{Z}_q$ according to χ and outputting pairs $(a, \langle a, s\rangle + e)$, where $\langle a, s\rangle$ denotes the inner product of two vectors. All computations are done modulo q. The (search) LWE problem is to reconstruct secret s, given access to an arbitrary number of samples from $A_{s,\chi}$. Without error e, this would be easy, since the remaining part becomes a system of linear equations, which is trivially solvable via Gaussian elimination. But with error, there is no efficient way of doing this known today, even with hypothetical quantum computers. In fact, it turns out that LWE is hard as long as the hardest instance of standard lattice problems is hard [Reg10]. This renders LWE attractive for use in cryptography.

Aside from the search LWE problem, there is another variant, called the decision LWE problem. Instead of recovering s, it is about distinguishing samples drawn from the LWE distribution $A_{s,\chi}$ from those sampled uniformly at random over $\mathbb{Z}_q^n \times \mathbb{Z}_q$. Regev proofed that search and decision version of LWE are in most cases equivalent [Reg10].

When building cryptosystems atop of LWE, the key-sizes seem to become inordinately large (matrices of size $O(n^2)$). To overcome this issue, the *ring-LWE* variant is willingly adopted, which leads to smaller key-sizes, exploiting the fact that there is no known security impact on LWE, when the samples are of some structured form. In ring-LWE, the group $\mathbb{Z}_q^n$ is replaced by the polynomial ring $\mathscr{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. In this variant, let $n > 0$ be a power of two, $q \geq 2$ the prime modulus, and χ a probability distribution on $\mathscr{R}_q$. For a fixed (and secret) polynomial $s \in \mathscr{R}_q$ denote with $A_{s,\chi}$ the probability distribution gained by choosing a polynomial

$a \in \mathcal{R}_q$ uniformly at random, choosing error with small coefficients $e \in \mathcal{R}_q$ according to χ and outputting pairs $(a, a \cdot s + e)$. Here, $a \cdot s$ denotes a simple multiplication of two polynomials. Similar to the standard variant, described above, the problem is to find secret s, given access to an arbitrary number of samples from $A_{s,\chi}$.

2.3 LP

In [LP11], Lindner and Peikert proposed their provably secure LP encryption scheme, which is based on the LWE problem and samples error from a discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ with standard deviation σ. In the following, n denotes the security parameter, l the message length, and q the modulus. Additionally, a pair of error-tolerant encoding/decoding functions encode: $\mathbb{Z}_2^l \to \mathbb{Z}_q^l$ and decode: $\mathbb{Z}_q^l \to \mathbb{Z}_2^l$ is necessary, such that decode(encode(m) + e) = m, for all messages $m \in \mathbb{Z}_2^l$ and $||e||_\infty < \lfloor q/4 \rfloor$. Such functions are part of the implementation and are described in Section 3.6.

Key Generation: Sample matrix $A \in \mathbb{Z}_q^{n \times n}$ uniformly at random, choose $E \leftarrow D_{\mathbb{Z},\sigma}^{l \times n}$ and $S \leftarrow D_{\mathbb{Z},\sigma}^{l \times n}$. Calculate $P = E - S \cdot A \in \mathbb{Z}_q^{l \times n}$. The public key is (A, P), the private key is S.
Encryption: To encrypt an l bit message m, choose $e_1 \leftarrow D_{\mathbb{Z},\sigma}^n$, $e_2 \leftarrow D_{\mathbb{Z},\sigma}^n$, $e_3 \leftarrow D_{\mathbb{Z},\sigma}^l$, and set $m' = \text{encode}(m)$. The ciphertext $c = (c_1, c_2) \in \mathbb{Z}_q^{n \times 1} \times \mathbb{Z}_q^{l \times 1}$ is computed as follows: $c_1 = A \cdot e_1 + e_2$ and $c_2 = P \cdot e_1 + e_3 + m'$.
Decryption: For decryption, compute and return decode($S \cdot c_1 + c_2$) $\in \mathbb{Z}_2^l$.

Aside of that, a ring-LWE version of LP is also described in [LP11]. The parameters n and q are specified as above, but instead of a message length l, the message m has to consist of n bit. The group $\mathbb{Z}_q^n$ is replaced by the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1 \rangle$, where a polynomial $a = \sum_{i=0}^{n-1} a_i x^i$ is represented by its coefficient vector $(a_0, ..., a_{n-1})^T$. Therefore, the encoding/decoding functions must not necessarily change. With χ denote a probability distribution over $\mathcal{R}_q$, for example a discrete Gaussian. A discrete Gaussian over $\mathcal{R}_q$ can be obtained by sampling the coefficients of a polynomial from $D_{\mathbb{Z},\sigma}$.

Ring Key Generation: Sample $a \in \mathcal{R}_q$ uniformly at random and choose $e \leftarrow \chi$, and $s \leftarrow \chi$. Calculate $p = e - s \cdot a \in \mathcal{R}_q$. The public key is (a, p), the private key is s.
Ring Encryption: To encrypt an n bit message m, choose $e_1 \leftarrow \chi$, $e_2 \leftarrow \chi$, and $e_3 \leftarrow \chi$. Set $m' = \text{encode}(m)$. The ciphertext $c = (c_1, c_2) \in \mathcal{R}_q^2$ is computed as follows: $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + m'$.
Ring Decryption: For decryption, compute and return decode($s \cdot c_1 + c_2$) $\in \mathbb{Z}_2^n$.

2.4 U-LP

In [CGW14], Cabarcas, Göpfert, and Weiden presented an adapted version of LP, called U-LP, which gathers noise (error), and secret from a uniform distribution, instead of a discrete Gaussian. This allows a simpler implementation, precludes decryption failures and gives hope for more efficient operations due to the simpler sampling. U-LP is worst-case secure regarding standard lattice problems. Unfortunately, the security proof requires to choose larger parameters for U-LP, so a performance decrease is measurable as shown in Section 4.2. In the following, dimension n, message m of length l, modulus q, and the encoding/decoding functions are defined

as in LP. With $\mathcal{U}_z$ denote the uniform distribution modulo z. Additionally s_k and s_e are the error bounds for key generation and encryption.

Key Generation: Sample $A \leftarrow \mathcal{U}_q^{n\times n}$, $E \leftarrow \mathcal{U}_{s_k}^{l\times n}$, and $S \leftarrow \mathcal{U}_{s_k}^{l\times n}$. Compute $P = E - S \cdot A \in \mathbb{Z}_q^{l\times n}$. The public key is (A, P), the private key is S.
Encryption: To encrypt an l bit message m, sample $e_1 \leftarrow \mathcal{U}_{s_e}^n$, $e_2 \leftarrow \mathcal{U}_{s_e}^n$, $e_3 \leftarrow \mathcal{U}_{s_e}^l$, and set $m' = \text{encode}(m) \in \mathbb{Z}_q^l$. The ciphertext $c = (c_1, c_2)$ is computed as follows: $c_1 = A \cdot e_1 + e_2$ and $c_2 = P \cdot e_1 + e_3 + m'$.
Decryption: For decryption, compute and return $\text{decode}(S \cdot c_1 + c_2) \in \mathbb{Z}_2^l$.

Equivalently to LP, the authors of U-LP mentioned a ring based analogue for U-LP in [CGW14], which leads to notably smaller key sizes and performance increases. Again, the group $\mathbb{Z}_q^n$ is replaced by the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where a polynomial $a = \sum_{i=0}^{n-1} a_i x^i$ is represented by its coefficient vector $(a_0, ..., a_{n-1})^T$. Also, message m has to be of size l bit.

Ring Key Generation: Sample $a \leftarrow \mathcal{U}_q^n$, $e \leftarrow \mathcal{U}_{s_k}^n$, and $s \leftarrow \mathcal{U}_{s_k}^n$. Calculate $p = e - s \cdot a \in \mathcal{R}_q$. The public key is (a, p), the private key is s.
Ring Encryption: To encrypt n bit message m, sample $e_1 \leftarrow \mathcal{U}_{s_e}^n$, $e_2 \leftarrow \mathcal{U}_{s_e}^n$, and $e_3 \leftarrow \mathcal{U}_{s_e}^n$. Set $m' = \text{encode}(m)$. The ciphertext $c = (c_1, c_2)$, is computed as follows: $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + m'$.
Ring Decryption: For decryption, compute and return $\text{decode}(s \cdot c_1 + c_2) \in \mathbb{Z}_2^n$.

3 Implementing U-LP

During this thesis, I implemented the U-LP cryptosystem (as well as LP) in C, creating a generic library for using this cryptosystem in practice (which are at the moment primarily testing and evaluation purposes). This includes the normal variant as well as the ring-LWE variant of U-LP. The library is built with the *CMake* build system [cma], ensuring cross platform compatibility (which is at least Windows, Mac OS X and Linux in this context). The compilation process relies actually on the *gcc* compiler [GNU], but can be easily adapted to other compilers. As of this writing, the source code of *ulpcrypt* can be found on GitHub [git]. The library is released under the terms of the MIT license.

In this Section, I justify the choice of the programming language, giving pros and cons about different options. I define the scope of the ulpcrypt library and its API, and write about parts of the implementation, which might be of some interest. Afterwards, some quality assurance measures, which are taken, are described.

3.1 Choice of Programming Language

When writing a cryptographic library, the first thing to consider is the programming language. While there are hundreds of programming languages, many of them are not well suited for developing security sensitive software. The main difficulty lies in the conjunction of performance and safety guarantees, where the latter is often not taken into account. C for example, the most prominent language in this field, compiles to extremely efficient machine code, taking

advantage of highly optimized compilers and libraries.[1] On the other hand it is easy to unintentionally produce critical flaws in C, which can lead to security vulnerabilities such as buffer overflows or use-after-free bugs. Additionally, C comes with a very small standard library, which requires the programmer to either write a lot of common code from scratch, or rely on third party libraries. C++ tried to improve this situation, but introduced a lot of complex language constructs, which are easily used incorrectly as well. Modern scripting languages, that come with clean syntax and semantics, provide a huge standard library, and support the programmer with automatic memory management. But such languages like Python or Ruby, often tend to result in rather inefficient software. It is desired that cryptographic software computes their expensive calculations without high delay. Therefore, languages which compile to efficient machine code are probably more suitable than interpreted scripting languages. Java might look like a compromising alternative, but is only half way compiled, sometimes suffers from virtual-machine bugs and is not established as the de-facto standard in this area, yet. Additionally, automatic memory management such as garbage collection is often considered as a lack of control. For example, some might want to wipe the memory after holding sensitive data, but one can argue that modern operating systems anyway zero out memory pages before passing them to other processes.

Finally, I decided to write the U-LP library in C. Since U-LP is not ready for the use in real world applications, the interest is mainly from academic nature. The goals which must be achieved are comparability and (to some degree) compatibility to similar libraries. These are typically written in C or C++, too. Additionally, performance is an important issue here, which straightly leads to C as the language of choice.

Nevertheless, for the future it would be valuable to have a language which assists the developer in avoiding critical failures and still produces efficient code, suitable for the use in cryptography and other security related areas (even down to operating system level code). The Rust programming language [rus], driven by the Mozilla Foundation, is designed as a system programming language, providing strong safety guarantees, although this language is very young and still not stable, yet. Another promising candidate might be Google's Go [Goo]. Time will show if that kind of language become prevalent.

3.2 The Scope of the Library

The main goal of this library, called *ulpcrypt*, is to provide an easy to use, universal implementation of the U-LP cryptosystem (for the normal variant as well as for the ring-LWE variant). It should be possible to utilize it in common use cases such as hybrid encryption, when building a secure channel over a network. U-LP, as well as most other post quantum encryption schemes, is mainly of academic interest yet, due to its limited performance. So the primary purpose is the evaluation and comparison to other encryption schemes. Since U-LP is the first provably secure LWE cryptosystem based on uniform error distribution, it might be of special interest. While I took measures to stem common problems like buffer overflows (see Section 3.7), far advanced threats like side-channel attacks are currently out-of-scope of this implementation. These become important, when cryptosystems are targeted for in use protocols.

[1] GMP for example, the GNU Multiple Precision Arithmetic Library, offers assembly level optimized support for arbitrary precision number types.

3.3 Defining an Interface for ulpcrypt

The central components when working with a cryptosystem are the public and private keys as well as the plaintext and ciphertext structures. For the plaintext, it suffices to hold the data in a `uint8_t` array of size $\lceil \frac{l}{8} \rceil$, which consists of arbitrary bytes. Here, l denotes the message length in bit. For the ciphertext it is slightly more compound, because a ciphertext in U-LP consists of two vectors. Therefore the `ulp_ciphertext` structure covers two `uint64_t` arrays, holding c_1 and c_2. Additionally the dimension parameters n and l are part of this structure to keep track of the array sizes. This is also required for the key structures, since C has no length attribute for arrays and matrices. Beside that, `ulp_public_key` contains two `uint64_t` values for modulus q and encryption-error-bound s_e, and the `uint64_t` matrices A and P. The modulus is also stored in `ulp_private_key`, because this value is necessary during the decryption calculations and the public key must not necessarily be present at this time. Furthermore the matrix S is part of the private key structure. Such matrices are stored as one-dimensional consecutive arrays, to make use of locality (for possible performance tuning). To make memory allocation for such compound datatypes easier, the `ulpcrypt` library provides the functions `ulp_alloc_public_key`, `ulp_alloc_private_key`, and `ulp_alloc_ciphertext`, which take the dimension parameters n and l and return pointers to corresponding structures with appropriate array sizes. For memory deallocation, complementary functions `ulp_free_public_key`, `ulp_free_private_key`, and `ulp_free_ciphertext` are also provided.

The first thing one might do when using `ulpcrypt` is generating a key pair. To accomplish this goal, one needs to pass values for q, s_k, and s_e. In [CGW14], the authors noted how these parameters must be chosen to fulfill the requirements of the security proof. The function `ulp_generate_parameters` does this computation and generates the three values dependent on security parameter n and message length l. After that, one can feed all these five parameters into the `ulp_generate_key_pair` function. Additionally, it takes two references to public and private key struct pointers, which are filled during the function call. There is no need to create these structures manually beforehand using the alloc-functions stated above.

To encrypt a `uint8_t` array of plaintext data, one can use the `ulp_encrypt` function. Beside that array, the function takes a pointer to a `ulp_public_key` structure and a reference to a `ulp_ciphertext` pointer. The latter structure is created during the encryption process, there is no need to create this manually beforehand. Note that the length of the plaintext array has to match the parameter l in the `ulp_public_key` structure.

The function `ulp_decrypt` works as expected, taking pointers to the `ulp_ciphertext` and `ulp_private_key` structures, and a reference to a `uint8_t` pointer for the decrypted plain text data, which again, is generated during the decryption process. Similar to the constraints for encryption, parameters n and l of the `ulp_ciphertext` have to match those in the `ulp_private_key`.

The functions for the ring-LWE variant look similar to the above mentioned. The nomenclature differs in the word *ring*, between *ulp* and the function name, for example `ulp_ring_encrypt`. Note that there is no parameter l in the ring-LWE variant. Instead, message length must be equal to the security/dimension parameter n. Key components a, p, and s are vectors instead of matrices. a `ulp_ring_generate_parameters` function does not exist, because there is no security proof and suitable parameter estimation, yet.

The library compiles to a static, as well as to a shared version. To access the function declarations, it suffices to include the single header `ulpcrypt.h`. For a detailed API overview, see Appendix A.

3.4 Gathering Random Data

U-LP requires generating secret, samples, and noise randomly. For the use in cryptography, (pseudo) random number generators have to fulfill strong security properties. As stated in [GPR06], these are:

- Pseudorandomness: The generated numbers conform to a desired probability distribution.
- Forward security: If an attacker learns the internal state of the pseudo random number generator, he can not derive anything about the previously generated numbers.
- Backward security: If an attacker learns the internal state of the pseudo random number generator, he can not foresee anything about subsequent numbers.

Such generators, which hold these properties, are called *cryptographically secure pseudo-random number generators* (CSPRNG). The Linux operating system provides such CSPRNGs via the two virtual devices `/dev/random` and `/dev/urandom`. Its security was evaluated in [GPR06] and improved afterwards. The `/dev/random` device computes random numbers entirely based on the Kernel's entropy pool, which gathers noise from input devices, and system and network interrupts. If there is not enough entropy available to safely generate the next random number, `/dev/random` blocks until enough entropy is gathered. The `/dev/urandom` device, in contrast, initializes a pseudo-random number generator when there is not enough "real" entropy available. This allows `/dev/urandom` to continuously output random numbers. As far as it is known, this random device is suitable for the use in cryptography, too. In U-LP, a lot of random data is required (e. g. multiple matrices during key generation), so `/dev/random` is inappropriate due to its blocking behavior. It simply would last to long. Therefore I rely on `/dev/urandom` in this implementation for uniform random number generation on Linux and Mac OS X. Unfortunately, Windows does not offer such an easy to use random device. Instead it provides an API call to a system random number generator in C++. To make use of it in plain C, some dynamic library loads and casts are necessary, to make `RtlGenRandom` available.

3.5 Utility Functions

The different operations during the U-LP calculations require some utility functions. The most important operation in ulpcrypt is the multiplication of two 64 bit integers modulo another integer. This functionality is implemented in a function called `mulmod` and is the most time-consuming operation involved. This is evaluated in Section 4.1. Basically, it relies on the `unsigned __int128` datatype provided by the GCC compiler, for storing the intermediate result. For the case that this datatype is not available, ulpcrypt contains a less performant fallback implementation.

For parameter generation, it is necessary to do prime number checks. One fast algorithm is the *Miller-Rabin primality test*. In this library, the deterministic variant of this test is implemented, which relies on a fixed set of bases instead of generating those randomly. This ensures a definitively correct result and eliminates the overhead of generating random bases. The deterministic

test is possible when only numbers below a concrete threshold are tested. This is the case here, since all numbers must fit into a 64 bit register.

During the primality check, the computation of $a^d \mod n$ must be performed. To do this efficiently, a function for exponentiation by squaring is implemented.

In the ring-LWE variant, multiplication is done in $\mathbb{Z}_q[x]/\langle x^n+1\rangle$. Multiplication in the ring, including the calculation of modulo x^n+1, is implemented in the function `poly_mulmod`. Instead of performing a complete polynomial division, the implementation exploits the fact that when x^n+1 is the modulus, it is $x^n \equiv -1$. Therefore division can be replaced by some subtractions.

3.6 Encoding / Decoding Functions

When encrypting a message, the plain text is given as a `uint8_t` array, i. e. a bitstring of size l. But to perform operations on the message in U-LP, it must be provided as an element of $\mathbb{Z}_q^l$, i. e. a `uint64_t` array of size l. This conversion is done by the function `encode` which takes a `uint8_t` array of size $\lceil l/8 \rceil$ and outputs a `uint64_t` array of size l. This is done by iterating over the input array, interpreting it as a bit array, and multiplying each bit with $\lfloor q/2 \rfloor$. The `decode` function reverts the encoding by iterating over the `uint64_t` array, converting each value between $q/4$ and $q-q/4$ to the bit 1, and each value outside this interval to the bit 0. This is a pair of error tolerant encoding/decoding functions as demanded in LP and U-LP. Note that there is no need to change these functions for the ring version, since a vector in $\mathbb{Z}_q[x]/\langle x^n+1\rangle$ is represented as an array of size n.

3.7 Quality Assurance

The ulpcrypt library should provide the functionality described in Section 3.3 and should also be able to perform this operations flawlessly, in terms of correctness. To assure this, a (semi-)automated test suite is built, which includes at least one test for each function provided by the library. The test suite is built around the *CTest* tool, which is part of the *CMake* build system.

The presented ulpcrypt is a cryptographic library and therefore a security sensitive software. Measures must be taken to avoid critical bugs which might weaken the system. Therefore, static code analysis tools are used, which can reveal flaws in the source code. One of this tools is *cppcheck* [cpp], which can for example find memory leaks and out-of-bounds accesses. Another tool with some kind of static code analysis is the gcc compiler itself.

Additionally the dynamic analysis tool *valgrind* [val] offers memory checking and the profiling of function calls. The latter is useful for the investigation of time consuming parts of the software, where performance tuning is especially worthwhile.

4 Evaluation

In this Section, the U-LP cryptosystem is evaluated. This includes the influence of the bit length when calculating operations in U-LP. Then, U-LP is compared to LP, regarding runtime behaviour and structure sizes. Afterwards, it is described how the usage of the ring-LWE variant and parallelization can optimize the encryption scheme.

The stated test runs are done on a 64-bit Linux machine (Fedora 20) with an Intel® Core™ i5-4200U CPU and 12 gigabyte main memory. The CPU has two physical cores and supports Hyper-Threading.

Bit Security	LP				U-LP			
	n	q	$\lceil \log_2(q) \rceil$	σ	n	q	$\lceil \log_2(q) \rceil$	s
85 – 87	256	378353	19	32	488	310027967972291	49	278420
116 – 118	320	590921	20	36	592	615698195236667	50	356922
228 – 229	512	1511821	21	46	888	2603483886956573	52	601141

Table 1: Typical parameters for LP and U-LP, including the bit length of modulus q, grouped by the intended level of security in bit. The values are based on the estimation in [CGW14].

number of multiplications	Times [ms]	
	32 bit	64 bit
10^8	39.71	40.71

Table 2: Comparison of the multiplication of 32 bit, and 64 bit numbers, where the data is already present and must not be loaded.

number of multiplications	Times [ms]	
	32 bit	64 bit
10^8	180.92	371.01

Table 3: Comparison of the multiplication of 32 bit, and 64 bit numbers, where the data must be loaded from memory.

4.1 Influence of the Bit Length

Well established asymmetric cryptosystems like RSA usually deal with numbers up to the length of 4096 bit to reach an acceptable level of security. U-LP and LP, in contrast, operate on matrices of much smaller numbers. Table 1 shows typical parameters for LP and U-LP, and what security level is implied. As one might see, the modulus q for LP is smaller than 32 bit and for U-LP is not bigger than 64 bit. Therefore, all the numbers involved in these cryptosystems, fit entirely into CPU registers of a modern 64 bit machine. This avoids calculations on arbitrary precision data structures, which usually come with a notable overhead. Nevertheless, U-LP operates on numbers twice as large as those of LP. The authors of [CGW14] stated, that multiplication in $\mathbb{Z}_q$ takes time proportional to $\log(q)$, which results in a performance drawback for U-LP. Considering the difference of the bit lengths in practice, it turns out that the comparison is more intricate. The multiplication of numbers on an *arithmetic logic unit* (ALU) should take a roughly constant time, as long as the numbers fit entirely into a CPU register. Modern CPUs, which are highly optimized, may result in a difference of one or two clock cycles, when recognizing that the operands are only 32 bit long. But in general, multiplication of 32 bit numbers on a 64 bit machine should not result in a remarkable performance gain. See Table 2 for an experiment. While CPU cores are extremely fast, memory access is comparably slow, and this is one matter which handicaps U-LP. The matrices used in U-LP must be read from memory and transferred to the CPU. Due to the 32/64 bit difference, these matrices are as twice as large as those in LP, and it takes nearly the doubled amount of time to transfer all this data. Table 3 shows the timing behavior when multiplying 32 bit, and 64 bit numbers, which must be loaded from memory.

Unfortunately, the explanation above does not cover the whole subject of calculation in U-LP (and LP). Instead of simply multiplying the numbers, they are afterwards reduced by the modulo q (which is according to valgrind the most time consuming operation in U-LP). Doing so with 32 bit numbers is easy, since the result of a 32 bit multiplication consists of a 64 bit number which entirely fits into a CPU register. Then, the x86_64 instruction `div` computes the quotient and

number of mulmod	Times [ms]	
	32 bit	64 bit
10^8	320.81	1628.28

Table 4: Comparison of the modular multiplication, executed on 32 bit, and on 64 bit operands. Note that the operands are loaded from memory, so the overhead of copying data from memory into CPU registers must be taken into account.

the remainder simultaneously, so the intended result is available quickly (see Listing 1). Trying the same with 64 bit operands exceeds the build-in capabilities of the CPU. The product of a 64 bit multiplication is 128 bit long and is written into two registers, one holding the upper 64 bits, one holding the lower 64 bits. While the result is easily available, too, it is not possible to pass this compound number directly as an input to another instruction. Since the C standard does not define an integer datatype greater than 64 bit, yet, I rely on the unofficial but handy `unsigned __int128` datatype, provided by the gcc. This allows to work with the 128 bit product similar as with the 64 bit product. But as already mentioned, there is no x86_64 instruction for dividing a 128 bit number. Listing 2 shows what the compiler produces to make this calculation possible.

Listing 1: 32 bit multiplication with modulo

```
1 mov   %edi,%eax
2 mov   %esi,%esi
3 mov   %edx,%ecx
4 imul  %rsi,%rax
5 xor   %edx,%edx
6 div   %rcx
7 mov   %edx,%eax
8 retq
```

Listing 2: 64 bit multiplication with modulo

```
1  mov   %rdi,%rax
2  mov   %rdx,%rcx
3  push  %rbp
4  mul   %rsi
5  mov   %rdx,%rsi
6  mov   %rax,%rdi
7  mov   %rcx,%rdx
8  xor   %ecx,%ecx
9  callq <__umodti3>
10 pop   %rbp
11 retq
```

After the multiplication is performed, the integer arithmetic routine `__umodti3` is called. Such routines are provided by the compiler for the case that native operations are not supported by the hardware. The use of `__umodti3` results in a bigger piece of code, manually computing the remainder, keeping track of overflows, etc. This costs much more time than the simple `div` instruction. Table 4 shows a comparison between 32 bit and 64 bit multiplication with modulo (called mulmod here). One can see that the operation on 64 bit operands is approximately five times slower than those on 32 bit operands. We already lost the factor two by loading the doubled amount of data from memory, so what remains for the pure application of mulmod is the factor 2.5. Obviously the efficiency highly depends on the available hardware. Theoretic CPUs which support 128 bit registers could compute the modular multiplication on 64 bit operands as fast as on 32 bit operands. Unfortunately, while in principle, the *AVX2* extension brings support for integer instructions, it does not include division and modulo instructions.

Bit Security	Times U-LP [ms]			Times LP [ms]		
	Generation	Encryption	Decryption	Generation	Encryption	Decryption
85 – 87	3648.29	18.90	5.82	467.13	4.93	1.14
116 – 118	5564.58	26.05	7.25	659.96	5.90	1.44
228 – 229	12521.58	51.48	10.54	1524.14	10.09	2.85

Table 5: Runtime comparison of U-LP and LP, regarding key generation, encryption, and decryption. Security parameter n is chosen differently to achieve the desired level of security, while message length $l = 256$ is fixed (which is a reasonable number when having hybrid encryption in mind).

4.2 Comparison of U-LP and LP

The main difference between U-LP and LP is the probability distribution, used for sampling noise and secret. LP relies on a discrete Gaussian, while U-LP utilizes a uniform distribution, which allows a simpler implementation. The authors of [LP11] based their security estimation on average case hardness results. For a fair comparison, the worst case hardness results from [CGW14] should be used, which supply parameters for both, U-LP and LP. Unfortunately, as shown in Table 1, the uniform probability distribution has negative impact on the parameter sizes, leading to numbers greater than 32 bit. Exceeding this threshold means, that the overhead of 64 bit operands, described in Section 4.1 comes in, reducing the performance of U-LP. Additionally, larger random matrices must be generated and handled, which costs time. On the other hand, uniform sampling should be more efficient than sampling from a Gaussian distribution, which is derived from an underlying uniform distribution. Nevertheless, this depends highly on the implementation of the Gaussian random number generator. In this implementation, a simple Box-Muller transform is used. Table 5 shows the runtime behavior for U-LP and LP, regarding key generation, encryption, and decryption. Due to the different security estimations, we need bigger values for n in U-LP to achieve a similar level of security. This enlarges the size of the matrices again and slows down the operations in U-LP. One can see that key generation with U-LP is 7 to 9 times slower than with LP, while encryption and decryption is about 3.5 to 5 times slower.

Another worthwhile comparison is about the key and ciphertext sizes. Table 6 shows the size of the data structures, when instantiating the cryptosystems with reasonable security parameters and an adequate message length (which is 256 here). While the size of the ciphertexts is only about a couple of kilobytes, the keys quickly reach the size of a few megabytes in U-LP and several hundred kilobytes in LP. The public key size increases most notably up to the factor 5.5 from LP to U-LP, while the increase factor of the private key size lies between 3.4 and 3.8. The ciphertext growth scatters around the factor 2.9.

4.3 Optimization by Using the Ring Variant

The long computation time and the large data structures make the U-LP cryptosystem difficult to use in practice. Using the ring-LWE variant as described in Section 2.4 can probably improve this situation. Instead of matrices, one-dimensional vectors are used which immediately leads to smaller keys. Having said this, one must note that there is no message length parameter l in this modified scheme. Instead it is possible to encrypt plaintext messages up to the length of the security parameter n. Unfortunately common message lengths such as 128 or 256 are way too

Bit Security	Sizes U-LP [kB]			Sizes LP [kB]		
	Public Key	Private Key	Ciphertext	Public Key	Private Key	Ciphertext
85 – 87	2904.60	999.44	5.96	524.30	262.16	2.06
116 – 118	4016.16	1212.44	6.80	737.30	327.70	2.32
228 – 229	8127.00	1818.64	9.16	1572.88	524.30	3.08

Table 6: Comparison of key sizes and ciphertext sizes of U-LP and LP. n is chosen according to Table 1, l is set to 256.

n	Times ring U-LP [ms]		
	Key Generation	Encryption	Decryption
256	7.28	7.82	2.94
488	18.76	24.01	10.77
592	24.64	36.68	15.71
888	40.94	76.81	35.42

Table 7: Runtime of the U-LP ring-LWE variant, regarding key generation, encryption, and decryption.

short for an adequate level of security. Therefore, one has to choose larger messages when using this system. Having hybrid encryption in mind, a possible scenario could be to concatenate two (or more) symmetric keys, (for example one for encryption, one for MAC), append some extra padding, and encrypt this enlarged message afterwards.

Table 7 shows the runtime for key generation, encryption, and decryption of the U-LP ring-LWE variant. As expected, the performance gain for the key generation is enormous. Encryption and decryption, in contrast, is only negligibly faster for a message length of 256. In fact, when increasing message length in ring U-LP to values of n as used in standard U-LP, encryption and decryption become even a bit slower. This might be due to the more costly modular multiplication for polynomials. When looking at the data structure sizes in Table 8, a remarkable decrease of the key sizes is observable. But this does not hold for the ciphertext. The ciphertext size might slightly increase, because in standard U-LP, it consists of $n + l$ entries, where in ring U-LP it consists of $2 * n$ entries (and n is usually larger than l). Additionally, the ciphertext is not much smaller than the keys, as it is in standard U-LP, because of the vectors instead of matrices as the underlying structures. But this should not be an issue.

While the advantages of faster key generation and smaller key structures are obvious, the drawback of slower encryption and decryption might diminish this benefit. However, the comparison should be taken carefully anyway. There is no proof or an estimation of the security properties of the ring U-LP variant yet, so it is unclear, if the juxtaposition of the cryptosystems instantiated with equal n is meaningful. Additionally, optimizations for the ring variant might be found which could improve the situation.

4.4 Optimization by Parallelization

A promising technique for the optimization of U-LP is the parallelization of several core routines. U-LP and LP, as well as numerous other lattice based algorithms seem to be perfectly appropriate for parallelization, due to the intense use of matrices and vectors. Operations on those data structures often involve the calculation of several independent partial results which

n	Sizes ring U-LP [kB]		
	Public Key	Private Key	Ciphertext
256	4.12	2.06	4.10
488	7.83	3.92	7.81
592	9.49	4.75	9.48
888	14.23	7.12	14.21

Table 8: Key sizes and ciphertext sizes of the U-LP ring-LWE variant.

n	l	Times [s]	
		1 Thread	4 Threads
888	512	24.56	9.29

Table 9: Runtime comparison between the singled-threaded and the multi-threaded version of U-LP. Measured is the overall runtime of key generation, and the encryption and decryption of a plain text afterwards. The cryptosystem is instantiated with $n = 888$ and $l = 512$.

might attract the use of parallelization techniques. As observable in Table 6, key generation is the worst performing part of U-LP. So a first approach would be to speed this process up, by parallelizing calculations such as the matrix multiplication $S \times A$. While this highly depends on the degree of parallelization, even with a simple OpenMP [ope] tuning is a speedup between 2 and 3 measurable. This is shown in Table 9. Note that the test run is done on two physical CPU cores which support Hyper-Threading. There are more portions of the code which allow parallelization, but when looking at the runtimes of the encryption and decryption functions, which only last a few milliseconds, it is likely that parallelization brings in a certain overhead for thread creation and scheduling, which could cancel out the intended benefit. Also note that a time consuming part of U-LP is the generation of large portions of random data, which cannot be parallelized well in the current implementation, where random data is read from a virtual system device (on UNIX compatible OS).

However, in general, parallelization can improve the runtime behavior significantly, depending on the different usage of parallelization techniques. A handy feature for example, is the vectorization capability of modern CPUs (such as SSE and AVX), which can be used to compute multiple operations simultaneously. Unfortunately, as mentioned earlier, the lack of integer division and modulo functions is an issue. Another powerful method is the massive parallel execution of a program on the GPU (as possible with the OpenCL standard [Khr]). The recent OpenMP 4.0 standard brings also support for GPGPU[2] and the calculation on dedicated accelerator devices. Such levels of parallelization can lead to a huge performance gain, but are not trivial to implement. Additionally, GPUs and accelerator devices are not available on every system. Many embedded devices do not include such hardware but should nevertheless be able to perform cryptographic operations.

[2] General Purpose Computation on Graphics Processing Unit

5 Conclusion

During this thesis I implemented the U-LP and the LP cryptosystems and built a library around them, providing functions for key generation, encryption, and decryption, for the standard variant, as well as for the ring-LWE variant. This might be viable for comparison experiments, when evaluating new (lattice based) encryption schemes. Such schemes could become very important, when a breakthrough in quantum computing renders classical asymmetric encryption schemes useless. Additionally, certain lattice based encryption schemes bring in the advantage of strong security proofs. Two examples are LP and U-LP, where the latter is the first provably secure LWE encryption scheme, which gathers noise and secret from a uniform distribution.

In this thesis, I instantiated the cryptosystems with practice-oriented parameters and evaluated runtime behavior and structure sizes. Unfortunately, as shown in Section 4.2, U-LP has some performance drawbacks. Especially the times needed for key generation can be hindering for practical usage of U-LP. In Sections 4.3 and 4.4, I showed that this situation can be improved to some extent, by using the ring-LWE variant and parallelization. Nevertheless, further optimization has to be done, both in the theoretical structure of the encryption schemes and the parameter selection, and in the practical implementation. Massive parallelization on GPUs could lead to reduced execution times, while dedicated hardware support would diminish some overhead, for example when dealing with 128 bit operands. Another problematical point is the size of the key structures. Sizes of several megabytes are probably too large for real time applications such as browsing the Web. While the ciphertext is relatively small, compared to the keys, the blow up is immense. For example a 256 bit plain text message can result in a 6 kilobyte ciphertext.

However, while further research is necessary, lattice-based encryption schemes seem to be a promising alternative for classical cryptosystems. Peikert explained in [Pei14] how useful some schemes already can be as drop-in replacements in protocols for everyday use in the Internet.

References

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer, 2009.

[CGW14] Daniel Cabarcas, Florian Göpfert, and Patrick Weiden. Provably secure LWE encryption with smallish uniform noise and secret. In *Proceedings of the 2nd ACM workshop on ASIA public-key cryptography*, pages 33–42. ACM, 2014.

[cma] CMake. `http://www.cmake.org/`. Accessed: 2014-09-27.

[cpp] Cppcheck - a tool for static C/C++ code analysis. `http://cppcheck.sourceforge.net/`. Accessed: 2014-09-27.

[git] ulpcrypt on GitHub. `https://github.com/exploide/ulpcrypt`. Accessed: 2014-09-27.

[GNU] GNU Project. GCC, the GNU compiler collection. `https://gcc.gnu.org/`. Accessed: 2014-09-27.

[Goo] Google Inc. The go programming language. `https://www.golang.org/`. Accessed: 2014-09-27.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[Khr] Khronos Group. The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`. Accessed: 2014-09-27.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *Topics in Cryptology–CT-RSA 2011*, pages 319–339. Springer, 2011.

[MP13] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In *Advances in Cryptology–CRYPTO 2013*, pages 21–39. Springer, 2013.

[ope] The OpenMP API specification for parallel programming. `http://openmp.org/`. Accessed: 2014-09-27.

[Pei14] Chris Peikert. Lattice cryptography for the internet. *IACR Cryptology ePrint Archive*, 2014:70, 2014.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[Reg10] Oded Regev. The learning with errors problem. *Invited survey in CCC*, 2010.

[rus] The rust programming language. `http://www.rust-lang.org/`. Accessed: 2014-09-27.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.

[val] Valgrind. `http://www.valgrind.org/`. Accessed: 2014-09-27.

A ulpcrypt API Documentation v1.0

General Usage

ulpcrypt compiles to a shared and a static library. To make the function definitions available, just include the header `ulpcrypt.h`. Most functions return the value 0 on success and a negative value, otherwise. Exceptions are the functions for allocating structures, which return a pointer, and the functions for deallocating structures, which return nothing.

Structures

ulp_public_key

Public key for U-LP.
Structure members:
`size_t n` - security parameter
`size_t l` - message length
`uint64_t q` - modulus
`uint64_t se` - error bound for encryption
`uint64_t* A` - part of the public key
`uint64_t* P` - part of the public key

ulp_private_key

Private key for U-LP.
Structure members:
`size_t n` - security parameter
`size_t l` - message length
`uint64_t q` - modulus
`uint64_t*` S - secret

ulp_ciphertext

Ciphertext for U-LP.
Structure members:
`size_t n` - security parameter
`size_t l` - message length
`uint64_t* c1` - first part of the ciphertext
`uint64_t* c2` - second part of the ciphertext

Functions

ulp_alloc_public_key

Allocate heap memory for storing a U-LP public key.
Parameters:
`size_t n` - security parameter
`size_t l` - message length
Return value:
`ulp_public_key*` - pointer to the allocated heap memory

ulp_alloc_private_key

Allocate heap memory for storing a U-LP private key.
Parameters:
`size_t n` - security parameter
`size_t l` - message length
Return value:
`ulp_private_key*` - pointer to the allocated heap memory

ulp_alloc_ciphertext

Allocate heap memory for storing a U-LP ciphertext.
Parameters:
`size_t n` - security parameter
`size_t l` - message length
Return value:
`ulp_ciphertext*` - pointer to the allocated heap memory

ulp_free_public_key

Deallocate heap memory for a U-LP public key.
Parameters:
`ulp_public_key* pub_key` - pointer to the memory to free
Return value:
`void`

ulp_free_private_key

Deallocate heap memory for a U-LP private key.
Parameters:
`ulp_private_key* priv_key` - pointer to the memory to free
Return value:
`void`

ulp_free_ciphertext

Deallocate heap memory for a U-LP ciphertext.
Parameters:
`ulp_ciphertext* ciphertext` - pointer to the memory to free
Return value:
`void`

ulp_generate_parameters

Generate the parameters for the U-LP cryptosystem dependent on n and l.
Parameters:
`size_t n` - security parameter
`size_t l` - message length
`uint64_t* sk` - pointer to error bound for key generation (will be generated)
`uint64_t* se` - pointer to error bound for encryption (will be generated)
`uint64_t* q` - pointer to modulus (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

ulp_generate_key_pair

Generate a keypair for the U-LP cryptosystem.
Parameters:
`size_t n` - security parameter
`size_t l` - message length
`uint64_t sk` - error bound for key generation
`uint64_t se` - error bound for encryption
`uint64_t q` - modulus, must be less than 2^{63} due to possible overflow problems
`ulp_public_key** pub_key_p` - pointer to a public key pointer (will be generated)
`ulp_private_key** priv_key_p` - pointer to a private key pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

ulp_encrypt

Encrypt a message with the U-LP cryptosystem.
Parameters:
`uint8_t msg[]` - the bytes to encrypt (number of bits has to match the l parameter in the key)
`ulp_public_key* pub_key` - the public key used for encryption
`ulp_ciphertext** ciphertext_p` - pointer to the ciphertext pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

ulp_decrypt

Decrypt a ciphertext with the U-LP cryptosystem.
Parameters:
`ulp_ciphertext* ciphertext` - pointer to the ciphertext to decrypt
`ulp_private_key* priv_key` - the private key used for decryption
`uint8_t** msg_p` - pointer to the message buffer pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

Ring Structures

ulp_ring_public_key

Public key for U-LP ring variant.
Structure members:
`size_t n` - security parameter
`uint64_t q` - modulus
`uint64_t se` - error bound for encryption
`uint64_t* a` - part of the public key
`uint64_t* p` - part of the public key

ulp_ring_private_key

Private key for U-LP ring variant.
Structure members:
`size_t n` - security parameter
`uint64_t q` - modulus
`uint64_t* s` - secret vector

ulp_ring_ciphertext

Ciphertext for U-LP ring variant.
Structure members:
`size_t n` - security parameter
`uint64_t* c1` - first part of the ciphertext
`uint64_t* c2` - second part of the ciphertext

Ring Functions

ulp_ring_alloc_public_key

Allocate heap memory for storing a U-LP public key (ring variant).
Parameters:
`size_t n` - security parameter and message length
Return value:
`ulp_ring_public_key*` - pointer to the allocated heap memory

ulp_ring_alloc_private_key

Allocate heap memory for storing a U-LP private key (ring variant).
Parameters:
`size_t n` - security parameter and message length
Return value:
`ulp_ring_private_key*` - pointer to the allocated heap memory

ulp_ring_alloc_ciphertext

Allocate heap memory for storing a U-LP ciphertext (ring variant).
Parameters:
`size_t n` - security parameter and message length
Return value:
`ulp_ring_ciphertext*` - pointer to the allocated heap memory

ulp_ring_free_public_key

Deallocate heap memory for a U-LP public key (ring variant).
Parameters:
`ulp_ring_public_key* pub_key` - pointer to the memory to free
Return value:
`void`

ulp_ring_free_private_key

Deallocate heap memory for a U-LP private key (ring variant).
Parameters:
`ulp_ring_private_key* priv_key` - pointer to the memory to free
Return value:
`void`

ulp_ring_free_ciphertext

Deallocate heap memory for a U-LP ciphertext (ring variant).
Parameters:
`ulp_ring_ciphertext* ciphertext` - pointer to the memory to free
Return value:
`void`

ulp_ring_generate_key_pair

Generate a keypair for the U-LP cryptosystem (ring variant).
Parameters:
`size_t n` - security parameter and message length
`uint64_t sk` - error bound for key generation
`uint64_t se` - error bound for encryption
`uint64_t q` - modulus, must be less than 2^{63} due to possible overflow problems
`ulp_ring_public_key** pub_key_p` - pointer to a public key pointer (will be generated)
`ulp_ring_private_key** priv_key_p` - pointer to a private key pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

ulp_ring_encrypt

Encrypt a message with the U-LP cryptosystem (ring variant).
Parameters:
`uint8_t msg[]` - the bytes to encrypt (number of bits has to match the n parameter in the key)
`ulp_ring_public_key* pub_key` - the public key used for encryption
`ulp_ring_ciphertext** ciphertext_p` - pointer to the ciphertext pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise

ulp_ring_decrypt

Decrypt a ciphertext with the U-LP cryptosystem (ring variant).
Parameters:
`ulp_ring_ciphertext* ciphertext` - pointer to the ciphertext to decrypt
`ulp_ring_private_key* priv_key` - the private key used for decryption
`uint8_t** msg_p` - pointer to the message buffer pointer (will be generated)
Return value:
`int` - 0 on success, a negative value otherwise