An Efficient Implementation of NTRUSVES

Johannes Buchmann, Martin Döring*, and Richard Lindner

Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,doering,rlindner}@cdc.informatik.tu-darmstadt.de

Abstract. The NTRUSVES encryption scheme is an interesting alternative to well-established encryption schemes such as RSA, ElGamal, and ECIES. The security of NTRUSVES relies on the hardness of computing short lattice vectors and thus is a promising candidate for being quantum computer resistant. In this paper, we present a highly efficient implementation of NTRUSVES within the Java Cryptography Architecture. We present comprehensive experimental results that show that NTRUSVES is superior in practice to well-established encryption schemes.

Keywords: NTRU, SVES, post-quantum cryptography, efficient implementation.

1 Introduction

Encryption schemes commonly used today are RSA [14], ElGamal [3], and ECIES [10]. The security of those schemes relies on the difficulty of factoring large composite integers or computing discrete logarithms. However, it is unclear whether these computational problems remain intractable in the future. For example, Shor [18] showed that quantum computers can be used to factor integers and to compute discrete logarithms in the relevant groups in polynomial time. Also, in the past thirty years there has been significant progress in solving the integer factorization and discrete logarithm problem using classical computers [15,16,2,1]. It is therefore necessary to come up with alternative encryption schemes which do not rely on the difficulty of factoring and computing discrete logarithms and which are considered secure even against quantum computer attacks. Such encryption schemes are called post-quantum encryption schemes.

A promising candidate for such a post-quantum encryption scheme is the lattice-based public-key cryptosystem NTRUEncrypt [6] in its NAEP/SVES-3 variant [8,9]. The cryptosystem is patented by NTRU Cryptosystems, Inc., a company founded in 1996 by J. Hoffstein, J. Pipher, and J. H. Silverman. SVES-3 is currently undergoing a standardization process and will presumably included in the upcoming IEEE standard 1363.1 [5]. We refer to the SVES-3 variant proposed in the draft standard as *NTRUSVES*.

* Author supported by SicAri, a project funded by the German Ministry for Education and Research (BMBF). See http://www.sicari.de.

We provide a highly efficient Java implementation of NTRUSVES according to draft version 8 of IEEE P1363.1. The implementation is provided within the Java Cryptography Architecture (JCA, [19,20]) and will be part of the Java Cryptographic Service Provider FlexiProvider [4].

We give comprehensive measurement results that show that NTRUSVES provides superior time complexity and competitive key sizes compared to well-established and widely used encryption schemes such as RSA.

The paper specifies NTRUSVES keys using Abstract Syntax Notation One (ASN.1) [11] which guarantees interoperability and permits efficient generation of X.509 certificates [7] and PKCS #12 personal information exchange files [13].

Organization. The rest of the paper is organized as follows: In Section 2, we give a brief mathematical description of NTRUEncrypt, NAEP, and SVES-3. Section 3 provides details of our NTRUSVES implementation and specifies the ASN.1 structures of NTRUSVES keys. In Section 4, we present the experimental results of our measurements of NTRUSVES and RSA and compare the results. Section 5 concludes the paper.

2 Mathematical background

In this section we give a brief mathematical description of NTRUEncrypt in the binary and the product form variant. We also illustrate the *NTRU Asymmetric Encryption Padding (NAEP)* scheme in its most common instantiation: the *Shortest Vector Encryption Scheme, third revision (SVES-3)* [8,9].

2.1 NTRUEncrypt

First, we define the main parameters of NTRUEncrypt (Table 1). The stated security requirements are taken from IEEE P1361.1-D9 [5], which is the latest draft of this standard to date.

Range	Description	Security requirements
$N \in \mathbb{N}$	Dim parameter	N has to be prime
$p \in \mathbb{N}$	Small modulus	p has to be equal to 2
$q \in \mathbb{N}$	Big modulus	$q \neq N$ has to be prime and big enough to prevent decryption errors, see *correctness*.

Table 1. NTRUEncrypt main parameters

Let $D(d)$ denote the set of binary polynomials of degree less than N with d coefficients equal to 1:

$$D(d) := \left\{ f \in \mathbb{Z}_2[X] \;\middle|\; \begin{array}{l} \deg(f) < N \text{ and} \\ f \text{ has d coefficients equal to 1} \end{array} \right\}.$$

Two additional space parameters $d_F, d_g \in \mathbb{N}$ define the private key spaces $D(d_F), D(d_g)$.

Using the parameter N from above, define the ring of convolution modular polynomials

$$R := \mathbb{Z}[X] \ / \ (X^N - 1).$$

All computations in this section are performed in R.

Key pair generation. Choose uniformly at random the binary polynomials $F \in D(d_F)$ and $g \in D(d_g)$. Change F slightly to obtain $f := 1 + pF$. Check whether f is invertible in R modulo q and denote the inverse f_q^{-1}. If the inverse does not exist, start over. Otherwise, compute the polynomial

$$h := f_q^{-1} \cdot p \cdot g \pmod{q}.$$

The private key is f, the public key is h.

Encryption. Encode the message M into a binary polynomial m of degree less then N. Randomly choose a binary blinding polynomial $r \in D(d_r)$. The encrypted message is

$$e := m + h \cdot r \pmod{q}.$$

Decryption. Receive the ciphertext e. Compute

$$\begin{aligned} a &:= f \cdot e = f \cdot m + p \cdot g \cdot r \pmod{q} \\ &\overset{(\star)}{=} m + p(F \cdot m + g \cdot r) \pmod{q}. \end{aligned}$$

Reduce the coefficients of a into the interval $[0, q[$. Decode the message

$$m := a \pmod{p}.$$

Correctness. The correctness of the procedure described above rests upon the following easy-to-prove lemma:

Lemma 1. *Let $b \in D(d)$ and $r \in R$ be arbitrary. Then it holds that*

$$\|br\|_\infty \leq d\|r\|_\infty.$$

In the lemma, the max-norm on R is defined as

$$\left\| \sum_{i=0}^{N-1} r_i X^i \right\|_\infty := \max_{i=0,\ldots,N-1} \{|r_i|\}.$$

Proof. Let $B = \{i \mid b_i \neq 0\}$ be the set of indices of b's non-zero coefficients. Since $b \in D(d)$, it holds that $|B| = d$. Rewrite the product and use the triangle inequality to obtain

$$\|br\|_\infty = \left\| \sum_{i \in B} r(X) X^i \right\|_\infty \leq \sum_{i \in B} \left\| r(X) X^i \right\|_\infty$$

Note that the max-norm of $r(X)$ on R is not changed by a multiplication with powers of X since this multiplication corresponds to a rotation of the coefficients. Conclude that

$$|br\|_\infty \leq \sum_{i \in B} \left\|r(X)X^i\right\|_\infty = \sum_{i \in B} \|r(X)\|_\infty = d\,\|r\|_\infty\,.$$

□

Decryption works if equality ($\star$) holds over R without taking both sides modulo q. By the above lemma this is guaranteed by choosing d_F and d_g such that

$$1 + p(d_F + d_g) < q.$$

Usually, NTRUEncrypt is used with two additional space parameters d_m and d_r. The parameters define the spaces $D(d_m)$ and $D(d_r)$ from which the message polynomial m and blinding polynomials r are picked. Using these further parameters makes NTRUEncrypt more efficient and the constraint on the parameters is relaxed to

$$1 + p(\min\{d_F, d_m\} + \min\{d_g, d_r\}) < q.$$

Arithmetic. The key pair generation, encryption, and decryption algorithms described above require arithmetic with polynomials from the ring R. After all computations, the coefficients of the polynomials are reduced modulo q, except once during decryption, where they are reduced modulo p. As shown in the previous paragraph, this modulo p step may be preceded by a reduction modulo q because if the space parameters are chosen appropriately, the coefficients are already smaller than q and therefore will not be changed by this reduction. So, in order to ease implementation, all arithmetic operations can be performed modulo q, that is, in the finite ring $R_q := \mathbb{Z}[X] \;/\; (q, X^N - 1)$ instead of R.

2.2 Product form variant

There is a more efficient variant of NTRUEncrypt, called the *product form variant*. In this section, we describe the differences to the regular, sometimes called *binary* variant. In the product form variant, the binary polynomials F and r are replaced by so-called *product form polynomials*. Product form polynomials are of the form $f_1 \cdot f_2 + f_3$, where f_1, f_2, and f_3 are very sparse binary polynomials. The product form variant is faster than the binary variant because of an optimized algorithm for multiplication of elements of R_q with product form polynomials (see Section 3.4).

Parameters. Choose N, p, q as before. The space parameters are d_{f_1}, d_{f_2}, d_{f_3}, $d_g \in \mathbb{N}$.

Key pair generation. Randomly choose $f_i \in D(d_{f_i})$ for $1 \leq i \leq 3$ and compute $F := f_1 \cdot f_2 + f_3$. The remaining steps are as before.

Correctness. In the product form variant, it is still needed that

$$a = f \cdot m + p \cdot g \cdot r = m + p(f_1 \cdot f_2 \cdot m + f_3 \cdot m + g \cdot r) \text{ in } R,$$

to avoid decryption failures. This is guaranteed if

$$1 + p(\min\{d_{f_1}, d_{f_2}\} + d_{f_3} + d_g) < q.$$

2.3 NAEP/SVES-3

NAEP/SVES-3 is a scheme based on NTRUEncrypt that is provably secure against adaptive chosen ciphertext attacks in the random oracle model, similar to OAEP+ for RSA. The scheme uses two hash functions G and H. Fix the maximal message bit length $maxLen$ and the bit length $bLen$ of some random strings. Precompute the internal message bit length

$$nLen := bLen + (\log_2(maxLen) + 1) + maxLen.$$

Encryption (see Figure 1). In order to encrypt a message M, compute its bit length $MLen$ and choose a random string b of length $bLen$. Compute a blinding polynomial $r = G(b||M||ID)$, where ID is a number that uniquely identifies the used parameter set.

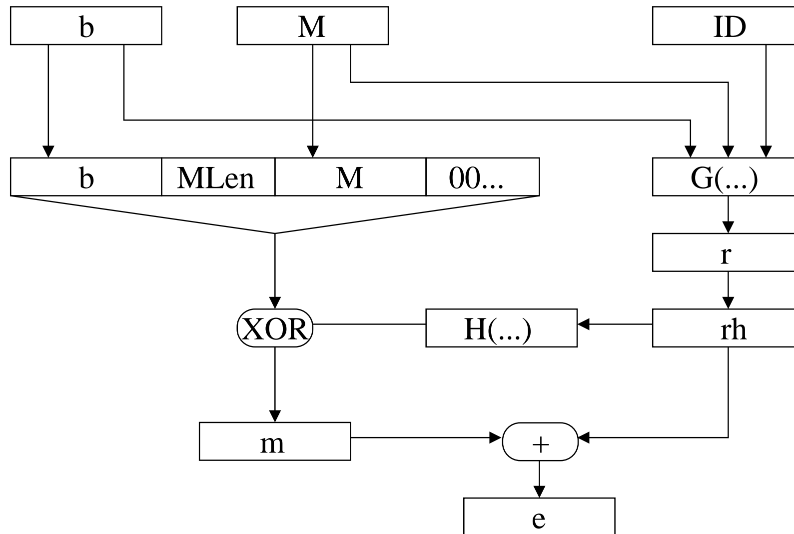

Fig. 1. SVES-3 encryption

Pad the message as $(b||MLen||M||00\ldots)$ to obtain a string M of the predefined bit length $nLen$. Compute the exclusive-or of M with $H(r \cdot h)$, the image of the product of the blinding polynomial and the public key under the second hash function H to obtain m. Encrypt m using the NTRUEncrypt encryption primitive as described in section 2.1.

Decryption (see Figure 2). Decrypt a ciphertext e with the NTRUEncrypt decryption primitive as described in Section 2.1 into a polynomial m. Compute the difference $rh := e - m$ and the exclusive-or of e with rh to obtain a bit string of length $nLen$. Interpret this bit string as $(b'||MLen'||M'||trunc)$. Check that $trunc$ consists only of zeroes and that $MLen'$ is the bit length of M'. Compute $r' = G(b'||M'||ID)$ and check whether $r'h = rh$ which was computed earlier. If all checks are positive, return M as the decrypted message.

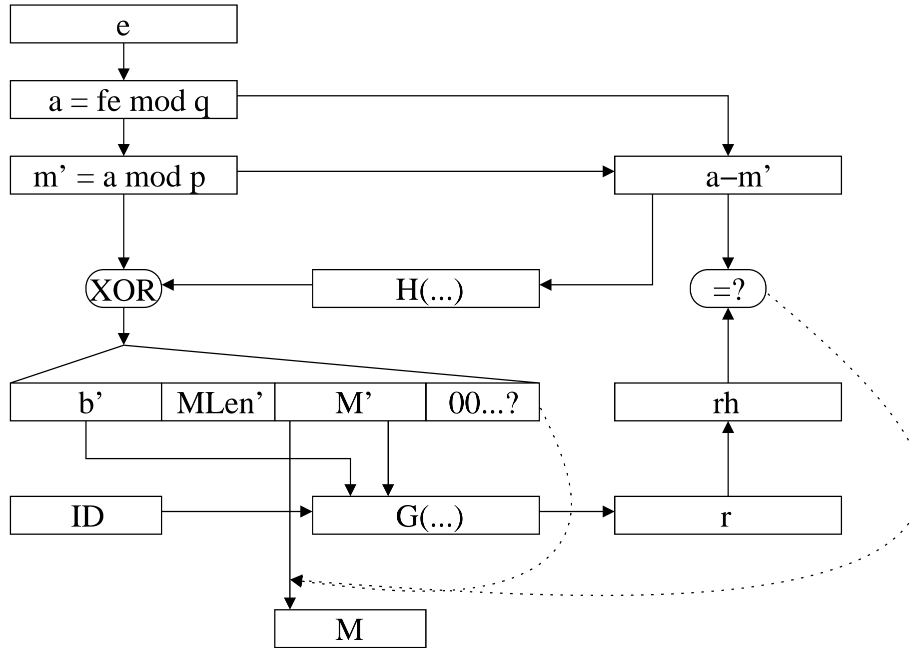

Fig. 2. SVES-3 decryption

3 Implementation notes

In this section, we provide details of our NTRUSVES implementation. We begin with describing the instantiation of SVES-3 given in IEEE P1363.1. Afterwards, we describe the supported parameters, the format of the keys, the used arithmetic, and the encoding format of polynomials and keys.

3.1 Instantiation

IEEE P1363.1 proposes concrete instantiations of the hash functions G and H used in the NAEP/SVES-3 scheme. The hash function G is called *Blinding Value*

Generation Method (BVGM) (in draft 8) or *Blinding Polynomial Generation Method (BPGM)* (in draft 9). We decide to use the latter notation for the rest of the paper. The BPGM itself uses a so-called *Seed Expansion Function (SEF)* (draft 8) or *Index Generation Function (IGF)* (draft 9), which in turn uses a hash function. Again, we use the latter name for the rest of the paper.

The draft standard proposes two different BPGM instantiations. The first one (LBP-BPGM1) is used to generate a binary blinding polynomial, the second one (LBP-BPGM2) produces a product form blinding polynomial. Both use the same IGF (IGF-MGF1). The underlying hash function is either SHA-1 or SHA-256 for the proposed parameter sets (see Section 3.2).

The input $(ID||m||b)$ to the BPGM can be extended to $(ID||m||b||hTrunc)$, where $hTrunc$ are some bits of the encoded public key h. Although this option is not used with the proposed parameter sets (i.e., the length of $hTrunc$ is 0), it is supported by our implementation (see also Section 3.3).

The function H is called *Mask Generation Function (MGF)* and uses a hash function. The draft standard proposes one instantiation (MGF1) which uses either SHA-1 or SHA-256 as hash function.

We do not describe the BPGM, IGF, and MGF algorithms in this paper, but rather refer the reader to [5]. Our implementation follows the description of the algorithms of draft 8 precisely.

3.2 Parameters

Our implementation supports all recommended parameter sets of draft version 9 of IEEE P1363.1 (see Annex A.5 of the draft standard). For each choice of the main parameter $N \in \{251, 347, 397, 491, 587, 787\}$, there is a binary and a product form parameter set. The parameter choices correspond to bit security levels of 80, 112, 128, 160, 192, and 256 bits, respectively. Each parameter set is chosen to maximize efficiency for the selected security level.

3.3 Key pairs

The name of the parameter set used to generate the keys is stored both in the public and in the private key.

Public key. The public key is the polynomial $h = f_q^{-1} \cdot p \cdot g \pmod q$.

Private key. Differing from the draft standard, we do not store the polynomial f as the private key. Instead, the pair of polynomials (F, g) is stored, where F either is a binary or a product form polynomial, and g is a binary polynomial. On the one hand, this speeds up decryption (see Section 3.5) and reduces the size of the encoded private key (see Section 3.6). On the other hand, the public polynomial h is needed to generate the input to the Blinding Polynomial Generation Method (see Section 3.1), so it must be possible to reconstruct h from the private key.

3.4 Arithmetic

IEEE P1363.1 proposes two algorithms optimized for the multiplication of elements of the ring $R_q = \mathbb{Z}[X]/(q, X^N - 1)$ with binary polynomials and with product form polynomials, respectively. These algorithms are considerably faster than the generic algorithm for the multiplication of two arbitrary elements of R_q.

The first algorithm is described in Section 6.2.5 of draft 8. If a binary polynomial has d coefficients equal to 1, then the algorithm requires dN additions over $\mathbb{Z}$ and N reductions modulo q.

The second algorithm is described in Section 6.2.6. of the draft. For product form polynomials consisting of three binary polynomials each with d coefficients equal to 1, the algorithm requires $3dN$ multiplications over $\mathbb{Z}$ and N reductions modulo q.

3.5 Decryption

The central decryption operation is the computation of the polynomial

$$a = f \cdot e \pmod{q},$$

where $f = 1 + pF$ is the private polynomial. Since in our implementation, the (binary or product form) polynomial F is stored in the private key (see Section 3.3), this computation is performed as

$$a = e + p \cdot e \cdot F \pmod{q},$$

using the efficient multiplication algorithms described in Section 3.4 for the computation of $e \cdot F$.

3.6 Encoding of polynomials and keys

Several steps of the encryption and decryption processes require the encoding of polynomials as (and the decoding from) octet strings. Additionally, in order to make the keys usable by public key infrastructures, they have to be encoded as well. In the following sections, we describe the encoding format of polynomials and keys.

Sparse binary polynomials. Sparse binary polynomials are stored as a sorted array of the degrees of the monomials having a non-zero coefficient. The degrees are encoded in descending order. Each degree is an integer in the interval $[0, N-1]$. It is encoded as an octet string (byte array) of length $\lceil \log_{256}(N-1) \rceil$ in big endian byte order.

Non-sparse binary polynomials are encoded using the BRE2OSP primitive described in Section 7.7.1 of IEEE P1363.1-D8.

Product form polynomials. A product form polynomial $f = f_1 \cdot f_2 + f_3$ consists of three sparse binary polynomials with the same number of non-zero coefficients. Product form polynomials are encoded as the concatenation of the encodings of f_1, f_2, and f_3 (see Section 3.6).

Other ring elements. Since all ring computations are performed modulo q, ring elements are stored as their coefficient vector with coefficients reduced modulo q. The ring elements are encoded using the RE2OSP primitive described in Section 7.5.1 of IEEE P1363.1-D8.

NTRUSVES keys. NTRUSVES keys are encoded into ASN.1 structures in order to be used with public key infrastructures. The polynomials are encoded as octet strings as described in the preceding sections.

Public key. The NTRUSVES public key ASN.1 structure is

```
NTRUSVESPublicKey ::= SEQUENCE {
  paramName     IA5STRING      -- name of the parameter set
  encH          OCTET STRING   -- encoded polynomial h
}
```

This structure is embedded into a SubjectPublicKeyInfo structure as defined in RFC 3280 [7].

Private key. The NTRUSVES private key ASN.1 structure is

```
NTRUSVESPrivateKey ::= SEQUENCE {
  paramName     IA5STRING      -- name of the parameter set
  encF          OCTET STRING   -- encoded polynomial F
  encG          OCTET STRING   -- encoded polynomial g
}
```

This structure is embedded into a PrivateKeyInfo structure as defined in PKCS #8 [12].

4 Measurement results and comparison

In this section, we state the experimental results of the measurements of our NTRUSVES implementation. We provide time measurements as well as key sizes for all parameter sets proposed by IEEE P1363.1-D9. We also provide similar results for the RSA PKCS #1 v2.1 encryption scheme. Based on these experiments, we compare the complexity of the two encryption schemes. The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 512 MB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

4.1 NTRUSVES

In this section, we state the results of the measurements of our NTRUSVES implementation. The results are summarized in table 2.

Column "Parameter set" denotes the used parameter set. The first six parameter sets are binary parameter sets, the other six sets are product form parameter sets.

Column "k" denotes the bit security level of NTRUSVES with the given parameter set. The estimates are taken from IEEE P1363.1-D9.

Columns "$s_{privKey}$" and s_{pubKey}" denote the size of the DER-encoded private key and public key ASN.1 structures, respectively (see Section 3.6).

Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the time measurement results for key pair generation, encryption, and decryption, respectively. For each parameter set, 500 key pairs were generated. For each key pair, 2000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

Parameter set	k	$s_{privKey}$	s_{pubKey}	t_{kpg}	t_{enc}	t_{dec}
ees251ep6	80	218 bytes	296 bytes	17.9 ms	0.2 ms	0.3 ms
ees347ep2	112	529 bytes	740 bytes	31.7 ms	0.3 ms	0.5 ms
ees397ep1	128	595 bytes	840 bytes	40.9 ms	0.4 ms	0.6 ms
ees491ep1	160	723 bytes	1028 bytes	62.0 ms	0.6 ms	1.0 ms
ees587ep1	192	853 bytes	1220 bytes	87.2 ms	0.9 ms	1.4 ms
ees787ep1	256	1118 bytes	1620 bytes	155.0 ms	1.4 ms	2.3 ms
ees251ep7	80	194 bytes	548 bytes	17.5 ms	0.1 ms	0.2 ms
ees347ep3	112	462 bytes	740 bytes	32.5 ms	0.2 ms	0.3 ms
ees397ep2	128	518 bytes	840 bytes	42.2 ms	0.2 ms	0.3 ms
ees491ep2	160	630 bytes	1028 bytes	63.7 ms	0.3 ms	0.5 ms
ees587ep2	192	738 bytes	1220 bytes	89.9 ms	0.5 ms	0.7 ms
ees787ep2	256	969 bytes	1620 bytes	159.8 ms	0.7 ms	1.1 ms

Table 2. NTRUSVES key sizes and time measurement results

4.2 RSA PKCS #1 v2.1

In this section, we state the results of the measurements of our RSA PKCS #1 v2.1 implementation. The implementation is part of the Java Cryptographic Service Provider FlexiProvider [4]. The implementation uses the built-in modular arithmetic of Java (class `BigInteger`). The results are summarized in table 3.

Column "Key size" denotes the bit size of the modulus. Column "k" denotes the bit security level of RSA for the given key size. The estimates are taken from the NIST Key Management Guideline [17].

Columns "$s_{privKey}$" and s_{pubKey}" denote the size of the DER-encoded private key and public key ASN.1 structures, respectively (see Section 3.6).

Columns "t_{kpg}", "t_{enc}", and "t_{dec}" denote the time measurement results for key pair generation, encryption, and decryption, respectively. For each key size, 20 key pairs were generated. The public exponent was chosen as $e = 2^{16} + 1$ for all key sizes and key pairs. For each key pair, 1000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

Key size	k	$s_{privKey}$	s_{pubKey}	t_{kpg}	t_{enc}	t_{dec}
1024	80	634 bytes	162 bytes	0.9 s	0.7 ms	13.2 ms
2048	112	1218 bytes	194 bytes	6.8 s	2.7 ms	91.7 ms
3072	128	1794 bytes	422 bytes	27.3 s	5.9 ms	294.4 ms
4096	144	2374 bytes	550 bytes	104.1 s	10.3 ms	682.5 ms

Table 3. RSA PKCS #1 v2.1 key sizes and time measurement results

4.3 Comparison

The measurement results stated above show that the NTRUSVES key pair generation, encryption and decryption operations are substantially faster than their RSA counterparts for the same security level. This is true also for larger security parameters because the asymptotic complexity of NTRUSVES grows slower in terms of the security parameter than the complexity of RSA.

For the same security level, the size of NTRUSVES private keys is about 1/3 of the size of RSA private keys. NTRUSVES public keys are about twice as large as RSA public keys.

5 Conclusion

In this paper, we present an efficient implementation of NTRUSVES. The implementation provides superior time complexity and competitive key sizes compared to the widely used RSA encryption scheme. This demonstrates that it is already possible today to use quantum computer resistant encryption schemes without any loss of efficiency. Because NTRUSVES is implemented as part of a Java Cryptographic Service Provider, it can be used with any application that uses the cryptographic framework provided by Java.

References

1. K. Aoki, J. Franke, T. Kleinjung, A. K. Lenstra, and D. A. Osvik. A kilobit special number field sieve factorization. Cryptology ePrint Archive, Report 2007/205, 2007. Available at `http://eprint.iacr.org/2007/205`.

2. S. Cavallar, B. Dodson, A. K. Lenstra, W. M. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. C. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. Factorization of a 512-Bit RSA Modulus. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer Verlag, 2000.
3. T. Elgamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology – CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Verlag, 1985.
4. The FlexiProvider group at Technische Universität Darmstadt. *FlexiProvider, an open source Java Cryptographic Service Provider*, 2001–2007. Available at `http://www.flexiprovider.de/`.
5. The IEEE P1363 Study Group for Future Public-Key Cryptography Standards. Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices. Available at `http://grouper.ieee.org/groups/1363/lattPK/draft.html`, January 2007.
6. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *Proceedings of the Third International Symposium on Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Verlag, 1998.
7. R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280 (Proposed Standard): Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Available at `http://www.ietf.org/rfc/rfc3280.txt`, April 2002. Updated by RFCs 4325, 4630.
8. N. Howgrave-Graham, J. H. Silverman, A. Singer, and W. Whyte. NAEP: Provable Security in the Presence of Decryption Failures. Cryptology ePrint Archive, Report 2003/172, 2003. Available at `http://eprint.iacr.org/2003/172`.
9. N. Howgrave-Graham, J. H. Silverman, and W. Whyte. Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3. In *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 118–135. Springer Verlag, 2005.
10. IEEE. IEEE Standard Specifications for Public-Key Cryptography, January 2000. See also "IEEE 1363 Amendment 1: Additional Techniques".
11. International Telecommunication Union. *X.680: Information technology Ũ- Abstract Syntax Notation One (ASN.1): Specification of basic notation*, 2002. Available at `http://www.itu.int/rec/T-REC-X.680/`.
12. RSA Laboratories. PKCS #8: Private-Key Information Syntax Standard (version 1.2). Available at `http://www.rsa.com/rsalabs/node.asp?id=2130`, November 1993.
13. RSA Laboratories. PKCS #12: Personal Information Exchange Syntax (version 1.0). Available at `http://www.rsa.com/rsalabs/node.asp?id=2138`, June 1999.
14. RSA Laboratories. PKCS #1: RSA Cryptography Standard (version 2.1). Available at `http://www.rsa.com/rsalabs/node.asp?id=2125`, June 2002.
15. A. K. Lenstra and H. W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer Verlag, 1993.
16. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
17. National Institute of Standards and Technology (NIST) Computer Security Resource Center (CSRC). SP 800-57 Part 1, Recommendation for Key Management – Part 1: General (Revised). Available at `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`, March 2007.

18. P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1994)*, pages 124–134. IEEE Computer Society Press, 1994.
19. Sun Microsystems. *The Java Cryptography Architecture API Specification & Reference*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html`.
20. Sun Microsystems. *The Java Cryptography Extension (JCE) Reference Guide*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html`.