International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 309–327, July 2016

Portable Implementation of Lattice-based Cryptography using JavaScript

Ye Yuan
Graduate School of Mathematics, Kyushu University.

Chen-Mou Cheng
Institute of Mathematics for Industry, Kyushu University.
Department of Electrical Engineering, National Taiwan University.

Shinsaku Kiyomoto
KDDI Laboratories.

Yutaka Miyake
KDDI Laboratories.

Tsuyoshi Takagi
Institute of Mathematics for Industry, Kyushu University.
CREST, Japan Science and Technology Agency.

Received: February 11, 2016
Revised: May 2, 2016
Accepted: July 4
Communicated by Toru Nakanishi

Abstract

In recent years, lattice-based cryptography has attracted a high degree of attention in the cryptologic research community. It is expected to be in wide use in the foreseeable future once large quantum computers are in sight. On the other hand, JavaScript is a standard programming language for Web applications. It is now supported on a wide variety of computing platforms and devices with immense efficiency improvement in the past few years. In this paper, we present the results of our JavaScript implementation of several lattice-based encryption schemes and show the speed performance on four common Web browsers on PC. Furthermore, we show performance results on two smaller computing platforms, namely, tablets running the Android operating system, as well as Tessel, an embedded system equipped with an ARM Cortex M3 microcontroller. Our results demonstrate that some of today's lattice-based cryptosystems can already have efficient JavaScript implementations and hence are ready for use on a growing list of computing platforms with JavaScript support.

Keywords: Lattice-based cryptography, JavaScript, Android, Tessel

Table 1: Summary of the implemented lattice-based encryption schemes

Scheme	Key generation	Encryption	Decryption
NTRU [17]	1. Positive integers n, p, q, d_f, d_g, d_m, d_r; gcd(p, q) = 1; 2. Ring: $R_q = \mathbb{Z}_q[x]/(x^n - 1)$; 3. Two random polynomials $\boldsymbol{f} \in L_{d_f}$, $\boldsymbol{g} \in L_{(d_g, d_g - 1)}$; 4. $\boldsymbol{f}_p \cdot \boldsymbol{f} \equiv 1 (\bmod\ p)$, $\boldsymbol{f}_q \cdot \boldsymbol{f} \equiv 1 (\bmod\ q)$; 5. Secret key: $(\boldsymbol{f}, \boldsymbol{f}_p)$ 6. Public key: $\boldsymbol{h} \equiv p \boldsymbol{f}_q \cdot \boldsymbol{g} \pmod q$;	1. Plaintext $\boldsymbol{m} \in L_{d_m}$; 2. $\boldsymbol{r} \in L_{d_r}$; 3. $\boldsymbol{c} \equiv \boldsymbol{r} \cdot \boldsymbol{h} + \boldsymbol{m} \pmod q$;	1. $\boldsymbol{a} \equiv \boldsymbol{f} \cdot \boldsymbol{c} \pmod q$; 2. Choose the coefficients of $\boldsymbol{a}$ in the interval from $[-\frac{q}{2}, \frac{q}{2})$; 3. Output: $\boldsymbol{f}_p \cdot \boldsymbol{a} \pmod p$;
NTRU-IEEE07 [36]	1. Positive integers n, q, d_1, d_2, d_3, d_m; gcd$(3, q)$=1 (let p=3); 2. Ring: $R_q = \mathbb{Z}_q[x]/(x^n - 1)$; 3. Four random polynomials $\boldsymbol{f}_1 \in L_{d_1}$, $\boldsymbol{f}_2 \in L_{d_2}$, $\boldsymbol{f}_3 \in L_{d_3}$ and $\boldsymbol{g} \in L_{(n/3,(n/3)-1)}$; let $\boldsymbol{F} = \boldsymbol{f}_1 \cdot \boldsymbol{f}_2 + \boldsymbol{f}_3$, then $\boldsymbol{f} = 1 + 3\boldsymbol{F}$; 4. $\boldsymbol{f}_q \cdot \boldsymbol{f} \equiv 1 (\bmod\ q)$; 5. Secret key: $\boldsymbol{f}$; 6. Public key: $\boldsymbol{h} \equiv 3 \boldsymbol{f}_q \cdot \boldsymbol{g} \pmod q$;	1. Plaintext $\boldsymbol{m} \in L_{d_m}$; 2. $\boldsymbol{r} = \boldsymbol{r}_1 \cdot \boldsymbol{r}_2 + \boldsymbol{r}_3$ where $\boldsymbol{r}_1 \in L_{d_1}$, $\boldsymbol{r}_2 \in L_{d_2}$, $\boldsymbol{r}_3 \in L_{d_3}$; 3. $\boldsymbol{c} \equiv \boldsymbol{r} \cdot \boldsymbol{h} + \boldsymbol{m} \pmod q$;	1. $\boldsymbol{a} \equiv \boldsymbol{f} \cdot \boldsymbol{c} \pmod q$; 2. Choose the coefficients of $\boldsymbol{a}$ in the interval from $[-\frac{q}{2}, \frac{q}{2})$; 3. Output: $\boldsymbol{a} \pmod 3$;
Regev's LWE [26, 22]	1. Positive integers m, n, l, q, t, r ($t \ll q$ and $r \ll q$) and a real $\alpha > 0$; 2. Matrix $\mathbf{S} \in \mathbb{Z}_q^{n \times l}$; matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$; matrix $\mathbf{E} \in \mathbb{Z}_q^{m \times l}$ where each elements of $\mathbf{E}$ is chosen according to $\chi_{\alpha \cdot q}$; 3. Matrix $\mathbf{B} = \mathbf{AS} + \mathbf{E} \pmod q \in \mathbb{Z}_q^{m \times l}$; 4. Secret key: $\mathbf{S} \in \mathbb{Z}_q^{n \times l}$; 5. Public key: $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^{m \times l}$;	1. Plaintext $\mathbf{m} \in \mathbb{Z}_t^l$; 2. $\mathbf{r} \in \{-r, -r+1, ..., r-1, r\}^m$; 3. $(\mathbf{c}_1, \mathbf{c}_2) = (\mathbf{A}^T \mathbf{r}, \mathbf{B}^T \mathbf{r} + \text{encode}(\mathbf{m})) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$;	1. Output: decode$(\mathbf{c}_2 - \mathbf{S}^T \mathbf{c}_1) \in \mathbb{Z}_t^l$;
LPR10-LWE [21]	1. Integers n, $q > 0$ and a real $s > 0$; 2. Ring: $R_q = \mathbb{Z}_q[x]/(x^n + 1)$; 3. Three random polynomials $\boldsymbol{e} \leftarrow \chi_s$, $\boldsymbol{a} \in R_q$, and a small $\boldsymbol{s} \in R_q$; 4. $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e} \in R_q$; 5. Secret key: $\boldsymbol{s} \in R_q$; 6. Public key: $(\boldsymbol{a}, \boldsymbol{b}) \in R_q \times R_q$;	1. Plaintext $\boldsymbol{m} \in \{0,1\}^n$; 2. Three random polynomials $\boldsymbol{e}_1$, $\boldsymbol{e}_2 \leftarrow \chi_s$ and a small $\boldsymbol{t} \in R_q$; 3. $(\boldsymbol{c}_1, \boldsymbol{c}_2) = (\boldsymbol{a} \cdot \boldsymbol{t} + \boldsymbol{e}_1, \boldsymbol{b} \cdot \boldsymbol{t} + \boldsymbol{e}_2 + \text{encode}(\boldsymbol{m})) \in R_q \times R_q$;	1. Output: decode$(\boldsymbol{c}_2 - \boldsymbol{c}_1 \cdot \boldsymbol{s}) \in \{0,1\}^n$;
LP10 ring-LWE [19]	1. Integers n, $q > 0$ and a real $s > 0$; 2. Ring: $R_q = \mathbb{Z}_q[x]/(x^n + 1)$; 3. Three random polynomials $\boldsymbol{r}_1, \boldsymbol{r}_2 \leftarrow \chi_s$, and $\boldsymbol{a} \in R_q$; 4. $\boldsymbol{b} = \boldsymbol{r}_1 - \boldsymbol{a} \cdot \boldsymbol{r}_2 \in R_q$; 5. Secret key: $\boldsymbol{r}_2 \leftarrow \chi_s$; 6. Public key: $(\boldsymbol{a}, \boldsymbol{b}) \in R_q \times R_q$;	1. Plaintext $\boldsymbol{m} \in \{0,1\}^n$; 2. Three random polynomials $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3 \leftarrow \chi_s$; 3. $(\boldsymbol{c}_1, \boldsymbol{c}_2) = (\boldsymbol{a} \cdot \boldsymbol{e}_1 + \boldsymbol{e}_2, \boldsymbol{b} \cdot \boldsymbol{e}_1 + \boldsymbol{e}_3 + \text{encode}(\boldsymbol{m})) \in R_q \times R_q$;	1. Output: decode$(\boldsymbol{c}_1 \cdot \boldsymbol{r}_2 + \boldsymbol{c}_2) \in \{0,1\}^n$.
LP11 [20]	1. Integers n, $q > 0$ and a real $s > 0$; 2. Matrices $\mathbf{R}_1, \mathbf{R}_2 \leftarrow \chi_s^{n \times n}$, matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$; 3. Matrix $\mathbf{B} = \mathbf{R}_1 - \mathbf{A}\mathbf{R}_2 \in \mathbb{Z}_q^{n \times n}$; 4. Secret key: $\mathbf{R}_2 \leftarrow \chi_s^{n \times n}$; 5. Public key: $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^{n \times n}$;	1. Plaintext $\mathbf{m} \in \{0,1\}^n$; 2. $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \leftarrow \chi_s^{n \times n}$; 3. $\begin{bmatrix} \mathbf{c}_1^t & \mathbf{c}_2^t \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^t & \mathbf{e}_2^t & \mathbf{e}_3^t + \text{encode}(\mathbf{m})^t \end{bmatrix} \cdot \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{I} & \\ & \mathbf{I} \end{bmatrix} \in \mathbb{Z}_q^{1 \times 2n}$	1. Output: decode$(\mathbf{c}_1^t \cdot \mathbf{R}_2 + \mathbf{c}_2^t)^t \in \{0,1\}^n$;

Table 2: Summary of the selected parameters that provide about 128-bit security

Scheme	Parameters							Key size (kB)				Bit-security
								Public	Private	Plaintext	Cipher text	
NTRU	n	p	q	d_f	d_g	d_m	d_r	0.532	0.075	0.032	0.534	128
	439	3	2048	146	146	130	146					
NTRU-IEEE07	n	q	d_1	d_2	d_3	d_g	d_m	0.535	0.051	0.032	0.534	128
	439	2048	9	8	5	146	130					
Regev's LWE	m	n	l	q	α	t	r	10328.164	181.157	0.023	1.360	128
	7616	438	192	383693	3.63×10^{-4}	2	2					
LPR10-LWE & LP10 Ring-LWE	n	q	s	-	-	-	-	1.424	0.185	0.039	0.994	126
	320	590921	35.77	-	-	-	-					
LP11	n	q	s	-	-	-	-	176.019	12.596	0.031	0.687	≥ 128
	256	4093	8.35	-	-	-	-					
3072-bit RSA	$\log_2 p$	$\log_2 q$	e	-	-	-	-	0.377	0.750	0.375	0.375	128
	≈ 1536	≈ 1536	65537	-	-	-	-					
ElGamal ECC (NIST P-256)	a	$\log_2 b$	$\log_2 p$	$\log_2 r$	$\log_2 G_x$	$\log_2 G_y$	-	0.063	0.031	0.063	0.125	128
	-3	≈ 191	≈ 256	≈ 192	≈ 189	≈ 187	-					

1 Introduction

JavaScript is a cross-platform scripting language with safety features and reasonable performance. A JavaScript program typically runs inside a Web browser and hence can execute properly regardless of the operating environment outside the browser. Currently, almost all mainstream Web browsers support JavaScript. Furthermore, there are a growing number of devices with internet capabilities that support JavaScript. For example, Tessel is an embedded system designed for internet of things (IoT) applications. It has an ARM Cortex M3 microcontroller and supports JavaScript to ease the task of controlling a diverse set of IoT devices.

In the past few years, JavaScript's performance has increased dramatically. For example, its execution speed has increased more than a factor of 100 since 2001 [35]. Now it is the standard programming language of HTML5, the latest markup language for structuring and presenting Web contents. It allows the same applications to run smoothly across a wide variety of digital devices that people use on a daily basis, including but not limited to smartphones, tablets, in addition to portable and desktop computers. This greatly simplifies Web application developers' job, as they no longer need to manage the extra complexity of application development and maintenance due to a rapidly growing number of internet devices. Therefore, we expect to see an even wider adoption of JavaScript, thanks to the further-reaching spread of Web applications in the society, partly due to the growing popularity of cloud computing and continual improvement of network infrastructures.

JavaScript's popularity and pervasiveness also come with prices, especially when it comes to security. For example, we have seen a wide variety of JavaScript-related Web browser vulnerabilities in the recent years. It is also very difficult to perform secure computation in a browser because it is a malleable and potentially hostile runtime environment. Furthermore, there lacks support for secure key storage and cryptographic random-number generation in a browser. Last but not least, performance can also be a showstopper for in-browser cryptography with JavaScript, especially back in those days when JavaScript was slow.

Fortunately, the Web Cryptography API standardization effort is making progress toward solving some of the aforementioned issues related to browser insecurity [34]. In this paper, we focus on the last issue: performance. We would like to see whether modern computing platforms with JavaScript support are ready for the computationally intensive public-key cryptography.

For this, we investigate lattice-based cryptography, an approach for post-quantum cryptography. A lattice is the set of all integer linear combinations of its basis vectors in a Euclidean space. Since first introduced by Ajtai, many lattice-based cryptosystems provide strong provable-security guarantees based on the worst-case hardness of various lattice problems [1]. Thanks to the provably close relationship between worst-case and average-case hardnesses of these lattice problems, there are many efficient constructions that leverage average-case hardness of, e.g., the short integer solution problem [1], the learning with errors (LWE) problem [27], etc. Last but not least, there is also the NTRU encryption scheme [17], a lattice-based scheme well known for its efficiency, whose security depends on certain hard problems in ideal lattices.

To the best of our knowledge, this work is the first attempt in the academic literature that evaluates the performance of JavaScript implementations of lattice-based encryption schemes. Part of our implementation is derived and adapted from publicly available Java implementations, e.g., discrete Gaussian sampling, as well as matrix and polynomial multiplications [14, 16, 7, 33]. We will report the performance of our implementation on multiple computing platforms and devices, including PC Web browsers, Android devices, as well as IoT embedded systems.

Specifically, we have chosen two schemes from the NTRU family, the original NTRU scheme [17] and one of its derivatives [36, 6]; the latter we shall refer to as NTRU-IEEE07. We have also selected a multi-bit version of Regev's LWE-based scheme [27, 26, 22, 11], which we shall refer to as Regev's LWE. Finally, we have picked three more recent and efficient encryption schemes with provable security, namely, LPR10-LWE [21], LP11 [20], and LP10 Ring-LWE [19]. We will give a high-level overview of these schemes in the next section and summarize their operations in Table 1. The parameters have been selected to provide about 128-bit security based on the suggestions by Hirschhorn *et al.* [15], Tore Kasper Frederiksen [11], Ruckert and Schneider [28], Lindner and Peikert [20], Cabarcas, Weiden, and Buchmann [9], as well as NIST Special Publication

800-57 Part 1 [4]. For ease of reference, they are summarized in Table 2. We have also included implementations of 3072-bit RSA [3, 23] and ElGamal elliptic curve cryptosystem (ElGamal ECC) over a NIST P-256 curve [31] for comparison against the prevailing public-key cryptosystems. Table 2 also shows the key sizes, as well as that of plaintext and ciphertext for each of the encryption schemes considered in this paper.

The rest of this paper is organized as follows. We will explain our notation and give a brief mathematical background in Section 2. We will then describe our implementation techniques in Section 3 and introduce our computing platforms for performance experimentation in Section 4. We will give detailed performance reports on Web browsers and small devices in Section 5 and 6, respectively. Finally, we conclude this paper in Section 7.

2 Lattice-based cryptography

In this section, we present the relevant mathematical background for discrete Gaussian sampling, the NTRU encryption schemes, the LWE problem, and the LWE-based encryption schemes.

Throughout this paper, we denote as $\mathbb{Z}_q$ the set of integers $\{0, 1, \ldots, q-1\}$, and $\mathbb{Z}_q[x]$, polynomials whose coefficients are in $\mathbb{Z}_q$. Let $R_q = \mathbb{Z}_q[x]/f(x)$ be a quotient polynomial ring, where $f(x)$ is a degree n polynomial. Throughout this paper, polynomials are denoted by bold italic small letters such as $\boldsymbol{f}$, while vectors, bold small letters such as $\mathbf{v}$ and matrices, bold large letters such as $\mathbf{A}$. $L_{(i,j)}$ will denote the set of those polynomials in R_q that have i coefficients equal to 1, j coefficients equal to -1, and other coefficients equal to 0. L_i is defined as $L_{(i,i)}$, i.e., polynomials having i coefficients that are either 1 or -1.

2.1 Discrete Gaussian sampling

For a positive real $s \in \mathbb{R}^+$, the discrete Gaussian distribution χ_s on an interval of integers $[a, b]$ is a discrete probability distribution that assigns to each element x in the interval a probability proportional to $\exp(-(x-\mu)^2/2\sigma^2)$, where, similar to the continuous Gaussian distribution, μ denotes the mean, and $\sigma > 0$ is the standard deviation which is equal to $s/\sqrt{2\pi}$ [13, 9]. In other words, the discrete Gaussian distribution is obtained by limiting the domain of the probability density function of a continuous Gaussian distribution to the integers, followed by proper scaling so that the total probability equals 1. It is common that $\mu = 0$, in which case x is sampled with probability proportional to $\exp(-x^2/2\sigma^2)$.

There are efficient algorithms for sampling from a discrete Gaussian distribution, e.g., by Gentry, Peikert, and Vaikuntanathan [13]. In our work, we use the algorithm described in Section 3.4 to perform such a sampling.

2.2 Learning with errors

Let $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ be a set of linearly independent vectors. The lattice L generated by $\mathbf{A}$ is the set of all integer linear combinations of vectors in $\mathbf{A}$, i.e., $L(\mathbf{A}) = \{\sum_{i=1}^{n} x_i \mathbf{a}_i \mid x_i \in \mathbb{Z}\}$. In cryptography, it is common to consider $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{Z}^m$, which are the lattice basis vectors put as columns in the matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$. The shortest vector problem (SVP) is the problem of finding a shortest nonzero vector $\mathbf{v} \in L(\mathbf{A})$ given a basis $\mathbf{A}$ of the lattice. A related problem, the closest vector problem (CVP) is to find a closest vector $\mathbf{v} \in L(\mathbf{A})$ to a target vector $\mathbf{t}$ not in the lattice. It is generally believed that no polynomial time algorithm can approximate these problems within any polynomial factor.

The learning with errors (LWE) problem is to recover a secret $\mathbf{s}$, chosen uniformly at random from $\mathbb{Z}_q^n$, given several noisy observations $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$, where $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ is chosen uniformly at random, with $\mathbf{e} \in \mathbb{Z}_q^m$ sampled from a discrete Gaussian distribution. An efficient solution to the LWE problem implies an efficient algorithm for solving SVP, and therefore the LWE problem has been used to construct secure public-key cryptosystems assuming the hardness of SVP.

There is a natural bijection between $\mathbb{Z}_q^n$ and the ring $\mathbb{Z}_q[x]/(x^n+1)$. This way we can identify vectors in a lattice as ring elements, and if our lattice happens to correspond to an ideal in the ring, then we call such lattice an ideal lattice. SVP on ideal lattices is connected with the hardness of the Ring-LWE problem, which, similar to the LWE problem, asks to find a secret $\boldsymbol{s}$, chosen uniformly at random from $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, given several noisy observations $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$, where $\boldsymbol{a} \in R_q$ is chosen uniformly at random, with $\boldsymbol{e} \in R_q$ sampled from a discrete Gaussian distribution. It has been shown that such a problem is also hard on average [21] and hence has been used in many efficient constructions of lattice-based schemes.

2.3 NTRU encryption schemes

First proposed in 1998 [17], the NTRU family includes several variants such as the original NTRU and NTRU-IEEE07. There have been several implementations on different platforms, but to the best of our knowledge, ours is the first in JavaScript.

Polynomial operations in the original NTRU scheme [17] are carried out in the ring $R_q = \mathbb{Z}_q[x]/(x^n - 1)$, where the dimension n is a prime, and q is a power of 2. Naturally, the polynomials will have degree $n - 1$. Let p be an element of R_q with small coefficients; p is coprime with q and will determine the plaintext space. A common choice is to take $p = 3$ for q a power of 2 [36, 15, 9]. The secret key is then a pair of polynomials $(\boldsymbol{f}, \boldsymbol{g})$ whose coefficients are chosen uniformly at random from $\{-1, 0, 1\}$. The polynomial $\boldsymbol{f}$ must have inverses modulo p ($\boldsymbol{f}_p$) and modulo q ($\boldsymbol{f}_q$), so the public key is $\boldsymbol{h} \equiv p\boldsymbol{f}_q \cdot \boldsymbol{g} \bmod q$.

The IEEE version of NTRU generates polynomial $\boldsymbol{f}$ from three polynomials $\boldsymbol{f}_1$, $\boldsymbol{f}_2$, and $\boldsymbol{f}_3$. To encrypt a message polynomial $\boldsymbol{m} \in R_p$, one chooses a random polynomial $\boldsymbol{r} \in R_q$ of small Euclidean norm and then compute the ciphertext polynomial $\boldsymbol{c} \equiv \boldsymbol{r} \cdot \boldsymbol{h} + \boldsymbol{m} \bmod q$. To decrypt, we first compute polynomial $\boldsymbol{a} = \boldsymbol{f} \cdot \boldsymbol{c}$ and reduce the result modulo q, which gives $p\boldsymbol{g} \cdot \boldsymbol{r} + \boldsymbol{f} \cdot \boldsymbol{m} \bmod q$. We then reduce modulo p and arrive at $\boldsymbol{f} \cdot \boldsymbol{m} \bmod p$. Finally, we multiply $\boldsymbol{a}$ by $\boldsymbol{f}_p$ to obtain the message. We note that the IEEE version of NTRU does not need to compute $\boldsymbol{f}_p \cdot \boldsymbol{a}$.

Overall, the security of the NTRU encryption schemes is based on the hardness of the SVP and CVP problems [24].

Differences between NTRU and NTRU-IEEE07. In key generation, NTRU requires the random polynomial $\boldsymbol{f}$ to satisfy certain conditions. That is, we need $\boldsymbol{f}$ to have inverses modulo p (denoted $\boldsymbol{f}_p$) as well as modulo q (denoted $\boldsymbol{f}_q$), and if it does not, we will need to resample $\boldsymbol{f}$. In practice, we have found that it is easy for randomly generated $\boldsymbol{f}$ to satisfy this condition. However, it does take some time to compute the inverses, which occupies a significant portion of the key-generation time. Another difference is that $\boldsymbol{f}$, $\boldsymbol{g}$, and the plaintext polynomial $\boldsymbol{m}$ used in NTRU are ternary polynomials with coefficients from $\{-1, 0, 1\}$.

NTRU-IEEE07 use slightly different polynomials. In key generation, it needs to generate polynomials $\boldsymbol{f}_1$, $\boldsymbol{f}_2$, and $\boldsymbol{f}_3$ whose coefficients are mostly 0 with a few exceptions (± 1). In encryption, it also generates three similar polynomials $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, and $\boldsymbol{r}_3$.

In addition, NTRU-IEEE07 has the parameter $p = 3$, so $\boldsymbol{f}$ always has inverse $\boldsymbol{f}_p$ modulo p. We only need to check whether it has inverse $\boldsymbol{f}_q$ modulo q, which has greatly reduced the running time of key generation. Similarly in decryption, we no longer need to compute to $\boldsymbol{f}_p$, which partly explains why NTRU-IEEE07 runs almost twice as fast as NTRU-IEEE07 under similar parameters. Therefore NTRU-IEEE07 is a faster variant of NTRU.

2.4 LWE-based encryption schemes

In this paper, we will investigate four LWE-based schemes. The first one was the seminal scheme proposed by Regev [27]. Secondly, Lyubashevsky, Peikert, and Regev constructed an LWE-based cryptosystem over $R_q = \mathbb{Z}_q/(x^n + 1)$ [21]. Thirdly, Lindner and Peikert proposed another Ring-LWE based cryptosystem [19]. Lastly, Lindner and Peikert presented an LWE-based scheme using integer matrix [20]. We have summarized in Table 1 the detailed operations of these four LWE-based schemes.

The construction of these four LWE-based schemes is quite similar, so here we only explain Regev's scheme [27]. Regev's original LWE scheme was subsequently extended to a multi-bit version [26, 22], which we refer to as Regev's LWE in this paper.

The Regev's LWE is parameterized by positive integers m, n, l, q, t, r, and a real number $\alpha > 0$. It also requires a pair of simple error-tolerant encoder and decoder, given by encode : $\mathbb{Z}_t^l \to \mathbb{Z}_q^l$ and decode : $\mathbb{Z}_q^l \to \mathbb{Z}_t^l$, where $\mathbb{Z}_t^l$ is the message alphabet, and the message length > 1. For example, if $\mathbb{Z}_t^l = \{0,1\}^l$, then we can define encode$(\mathbf{m}) = \bar{\mathbf{m}}$, which means the element $\mathbf{m}$ of $\mathbb{Z}_t^l$ multiplied by $\lfloor q/2 \rfloor$, and decode$(\bar{\mathbf{m}}) = 1$ if $\bar{\mathbf{m}}$ is closer to $\lfloor q/2 \rfloor$ mod q and 0 otherwise. In key generation, we choose a matrix $\mathbf{S}$ uniformly at random from $\mathbb{Z}_q^{n\times l}$ and a matrix $\mathbf{A}$ uniformly at random from $\mathbb{Z}_q^{m\times n}$. Let $\mathbf{B} \equiv \mathbf{AS} + \mathbf{E} \bmod q$, where $\mathbf{E}$ is a matrix consisting of small errors with the error parameter α. The secret key is then $\mathbf{S}$, and the public key, the pair $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^{m\times l}$.

For each bit of the message vector $\mathbf{m}$, we choose a vector $\mathbf{r}$ uniformly at random which contains small integers and then let the encryption be $(\mathbf{A}^T\mathbf{r}, \mathbf{B}^T\mathbf{r} + \text{encode}(\mathbf{m})) = (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$. The decryption of the pair $(\mathbf{a}, \mathbf{b})$ is then carried out by computing decode$(\mathbf{b} - \mathbf{S}^T\mathbf{a}) \in \mathbb{Z}_t^l$.

The IND-CPA security of Regev's LWE is based on the hardness of solving LWE problems for distinguishing the distribution of $\mathbf{B} \equiv \mathbf{AS} + \mathbf{E} \bmod q$ from the uniform distribution.

Differences of the implemented LWE-based encryption schemes. Regev's LWE [27, 26, 22] requires relatively large parameters to achieve a certain bit security, which, not surprisingly, has an advert impact on speed performance. In key generation, the secret key is $\mathbf{S} \in \mathbb{Z}_q^{n\times l}$, while the public key is $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$, so the computation of $\mathbf{AS}$ takes up most of the time when m, n, and l are large. In fact, matrix multiplication is almost always the performance-critical step in determining the running time of various steps in Regev's LWE except in decryption, where more time is spent in transposing the $\mathbf{S}$ than multiplying a smaller matrix with a vector. Therefore, it is possible to further improve the performance by using more efficient matrix multiplication algorithms.

LP11 scheme [20] also using matrices, but LP11 has much smaller keys than Regev's LWE. In contrast to Regev's LWE, LP11 has much smaller parameters to match roughly a same level of bit security, so the key size becomes very small. The computation time is also significantly reduced compared with Regev's LWE. The space efficiency is also better, i.e., LP11 tends to have more bits in the plaintext than Regev's LWE for the same level of bit security. However, it is important to note that the parameters given by Lindner and Peikert [20] are actually too small, resulting in a very high error rate (about 1%) [9]. In order to have a negligible error rate, we would need to adjust the parameters properly.

Both of LPR10-LWE [21] and LP10 Ring-LWE [19] are Ring-LWE based schemes. LP10 Ring-LWE is a generalization of LP11, in which matrix computation is replaced by polynomial computation. The two schemes, LPR10-LWE and LP10 Ring-LWE, are quite similar and hence have similar speed performance.

3 Implementation using JavaScript

In this section, we describe our implementation techniques for matrix multiplication, polynomial multiplication, and discrete Gaussian sampling. For NTRU schemes, we have chosen to implement the Karatsuba algorithm for polynomial multiplication. For that in the Ring-LWE schemes, we have implemented the iterative forward number theoretic transform, which we will describe in more detail later in this section.

Although there are many open-source JavaScript mathematics libraries, most of them are not so optimized for carrying out our target cryptographic computation. Instead, we adapt some open-source Java and JavaScript code to the mathematical objects of our own design. JavaScript is a scripting language with weak typing, so we need ways to achieve certain desirable characteristics of object-oriented programming such as encapsulation in our adaption. For example, to create a polynomial object, our design of the polynomial structure is as below.

```
var Polynomial = function() {
    //member variables
    var coeffs = [];  //coefficients
    //...

    //private methods
    function shuffle(args) {
      //method body
    }
    //...

    //public methods
    var init_polynomial = function() {
       //method body
    };
    //...

    return {
      coeffs : function() {return coeffs;},
      //...
      init_polynomial : init_polynomial,
      //...
    };
  };
```

Consequently, we adapt existing open-source Java or JavaScript implementation of polynomial multiplication accordingly.

Another problem is related to our implementation platforms. Some browser-specific objects such as the `window.crypto` object cannot be used on Tessel. Hence, we have to take into consideration platform differences to make our implementation truly portable. For example, Tessel does not support the long array, so we need to modify some of our data structures in a way that does not sacrifice speed performance too much.

3.1 Random-number generation

In JavaScript, the `Math.random()` function returns a floating-point pseudo-random number in the range $[0, 1)$. However, it is not a cryptographically secure pseudo-random number generator (CSPRNG). Therefore, we will not use this function except on Opera, where `Math.random()` is actually based on a CSPRNG [1].

For other web browsers, we use `window.crypto.getRandomValues()` from the latest W3C Web Cryptography API draft [34]. It generates cryptographically secure random numbers, which is also used by the ElGamal ECC implementation that we compare against [31]. However, `window.crypto` object is not available on some platforms such as Tessel. Instead, on Tessel we adapt the open-source `Alea` PRNG for its speed [2].

Opera also has an implementation of the `window.crypto.getRandomValues()` function. However, when generating a floating-point random number uniformly in the range $[0, 1)$, `window.crypto.getRandomValues()` is more than 40 times slower than `Math.random()` in Opera. Therefore, in our performance comparison we use `Math.random()` on Opera, `window.crypto.getRandomValues()` on all Web browsers, and `Alea` on other Android devices and Tessel.

3.2 Matrix multiplication

JavaScript provides an array object designed to store data values indexed by an integer-valued key. It can be used to create one-dimensional array objects. However, there are no two-dimensional array

[1]See also the email: `https://lists.w3.org/Archives/Public/public-webcrypto/2013Jan/0063.html`

objects in JavaScript. Therefore, we use nested arrays of array objects to represent matrices instead. That is, if we want to build a matrix, we can build a one dimensional array consisting of some one dimensional arrays, all having the same number of elements. For example, to construct a matrix $\mathbf{A} \in \mathbb{Z}^{x \times y}$, we build an array object of length x containing one-dimensional arrays of length y, and each element of the array is a row of $\mathbf{A}$. Then we can use $\mathbf{A}[i][j]$ to obtain the element in row i, column j.

For matrix multiplication, we have implemented the Strassen algorithm [32]. However, for the selected parameters, the Strassen algorithm is not necessarily faster than the standard textbook matrix multiplication algorithm. Therefore, we choose to optimize the standard matrix multiplication algorithm as follows.

First, if we compute the matrix product $\mathbf{C} = \mathbf{AB}$, we need to multiply a row vector in matrix $\mathbf{A}$ by a column vector in matrix $\mathbf{B}$ as shown in the pseudocode below.

```
for(var i=0; i<A.rows_length; i++) {
  for(var j=0; j<B.columns_length; j++) {
    var s = 0;
    for(var k=0; k<A.columns_length; k++) {
      s += A[i][k] * B[k][j];
    }
    C[i][j] = s;
  }
}
```

Loading the elements of matrix $\mathbf{B}$ in this way can be slow in JavaScript, as it is stored in row-major order. We can cache the columns of matrix $\mathbf{B}$ before the inner loop to speed up element loads as follows.

```
var Bcolj = [];
for(var j=0; j<B.columns_length; j++) {
  for(var k=0; k<A.columns_length; k++) {
    Bcolj[k] = B[k][j];      // step 1
  }
  for(var i=0; i<A.rows_length; i++) {
    var Arowi = A[i];      // step 2
    var s = 0;
    for(var k=0; k<A.columns_length; k++) {
      s += Arowi[k] * Bcolj[k];
    }
    C[i][j] = s;
  }
}
```

Similarly, when computing vector-matrix multiplication such as $\mathbf{C} = \mathbf{rA}$, we can also change the order of element multiplications as follows.

```
for(var i=0; i<A.columns_length; i++) {
  C[i] = 0;
}
for(var i=0; i<A.rows_length; i++) {
  var Arowi = A[i];
  for(var j=0; j<A.columns_length; j++) {
    C[j] += Arowi[j] * r[i];
  }
}
```

Although the number of arithmetic operations is the same, the computation can be significantly sped up due to faster memory loads.

3.3 Karatsuba and Fast Fourier Transform (FFT)

We use an array $[f_0, \ldots, f_{n-1}]$ to represent a polynomial $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i x^i \in R_q$. In other words, we simply create an array object to store the coefficients to represent a polynomial. In our case, the coefficients are stored in an ascending order. The array object provides all coefficients from the lowest to the highest degree such that the position of each coefficient corresponds to the degree of the term to which it belongs.

To multiply two polynomials, we can use the standard schoolbook polynomial multiplication. Other approaches include the Karatsuba algorithm and Fast Fourier Transforms (FFT). Considering the parameters of NTRU encryption schemes from Table 2, we have experimented with the Karatsuba algorithm in our implementations, which we have adapted from an open-source implementation [25]. Algorithm 1 shows the Karatsuba algorithm for polynomial multiplication that we have implemented for NTRU encryption schemes.

```
  Input: Polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$
  Output: $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$
1 Let $n = \max(\text{degree}(\boldsymbol{a}), \text{degree}(\boldsymbol{b})) + 1$;
2 If $n == 1$, then return $\boldsymbol{a} \cdot \boldsymbol{b}$;
3 Let $\boldsymbol{a} = a_1(x) + a_2(x)x^{\lceil n/2 \rfloor}$;
4 Let $\boldsymbol{b} = b_1(x) + b_2(x)x^{\lceil n/2 \rfloor}$;
5 $c_1(x) = \text{KaratsubaAlgorithm}(a_1(x), b_1(x))$;
6 $c_2(x) = \text{KaratsubaAlgorithm}(a_2(x), b_2(x))$;
7 $c_3(x) = \text{KaratsubaAlgorithm}(a_1(x) + a_2(x), b_1(x) + b_2(x))$;
8 return $c_1(x) + c_2(x)x^n + (c_3(x) - c_2(x) - c_1(x))x^{\lceil n/2 \rfloor}$
```

Algorithm 1: Karatsuba algorithm

FFT has lower asymptotic complexity $O(n \log n)$ for multiplying polynomials with larger degrees. In this paper, we follow the state of the art and use the number theoretic transform (NTT) [29, 8]. NTT is a generalization of FFT by replacing complex numbers with an n-th primitive root of unity in a finite ring $\mathbb{Z}_q$, the integers modulo a prime q. Thus, there is no need for any floating-point arithmetic, and all operations are additions and multiplications modulo q.

Algorithm 2 shows the iterative forward number theoretic transform algorithm for polynomial multiplication that we have implemented for Ring-LWE encryption schemes. *BitReverse* at line 1 is a permutation of a sequence of n items so that the new index of each element is the reverse of the bit string of its old index, where n is a power of 2.

Here the trick is to perform polynomial operations in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where n is a power of 2. Considering the parameters of the target Ring-LWE encryption schemes, we need to pad zeros to obtain an equivalent polynomial with $2n$ coefficients, as well as to reduce the final product modulo $(x^n + 1)$. In order to multiply two polynomials $\boldsymbol{a}$ and $\boldsymbol{b} \in R_q$, it is required to compute the product $\boldsymbol{c} = NTT^{-1}(NTT(\boldsymbol{a}) \otimes NTT(\boldsymbol{b}))$, where $\otimes$ is entry-wise multiplication [6]. It is generally believed that FFT is faster than the standard schoolbook polynomial multiplication when the polynomials have high-enough degrees [14].

3.4 Discrete Gaussian sampling

There are several methods to sample values from a discrete Gaussian distribution. Rejection sampling [13, 10] is one of these available approaches. In this approach, a sample value is chosen uniformly from $[-\tau\sigma, ..., \tau\sigma]$, where τ is a the tail-cut factor that determines where to drop the negligible probability of far samples. The selected integer is then accepted with probability proportional to $\exp(-x^2/2\sigma^2)$, or otherwise rejected. In the latter case, one simply resamples a new integer and then repeats this process until success. By choosing a suitable tail-cut factor τ, the resulting distribution can have negligible difference from the ideal distribution [5].

Algorithm 3 shows the basic rejection sampling method for discrete Gaussian distribution [10]. *RandomInt* in line 2 is a random, uniformly distributed integer in the range [a, b], where a and b

```
    Input: Polynomial $\boldsymbol{a} \in \mathbb{Z}_q[x]$ of degree $n-1$ and $n$-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
    Output: Polynomial $\boldsymbol{A} = NTT(\boldsymbol{a})$
 1  Let $\boldsymbol{A} = BitReverse(\boldsymbol{a})$;
 2  for $m = 2$ to $n$ by $m = 2m$ do
 3      $\omega_m = \omega_n^{n/m}$ , $\omega = 1$;
 4      for $j = 0$ to $m/2 - 1$ do
 5          for $k = 0$ to $n - 1$ by $m$ do
 6              $t = \omega \cdot \boldsymbol{A}[k + j + m/2] \bmod q$;
 7              $u = \boldsymbol{A}[k + j]$;
 8              $\boldsymbol{A}[k + j] = u + t \bmod q$;
 9              $\boldsymbol{A}[k + j + m/2] = u - t \bmod q$;
10          $\omega = \omega \cdot \omega_m$;
11  return $\boldsymbol{A}$
```

Algorithm 2: Iterative forward number theoretic transform (NTT)

$\in \mathbb{Z}$. *RandomFloat* in line 4 is a random, uniformly distributed floating-point number in the range (0, 1].

```
    Input: Floating-point numbers $c$, $r$ and $\tau$
    Output: A sample value $x \in \mathbb{Z}$
 1  Let $h = -\pi/r^2$; $x_{min} = \lfloor c - \tau r \rfloor \in \mathbb{Z}$, $x_{max} = \lceil c + \tau r \rceil \in \mathbb{Z}$;
 2  Let $x = RandomInt(x_{min}, x_{max})$;
 3  Let $p = \exp(h \cdot (x - c)^2)$;
 4  Let $r = RandomFloat()$;
 5  If $r < p$ then return $x$; else goto step 2
```

Algorithm 3: Rejection sampling on $\mathbb{Z}$

When it comes to actual implementation, rejection sampling only needs to store a few parameters and hence enjoys a small memory footprint. However, it requires a lot of computation and easily becomes a bottleneck in runtime. In this paper, we have adopted a time-memory trade-off strategy by implementing the inverse-transform sampling to reduce runtime computation. That is, we precompute all possible values of $\exp(-x^2/2\sigma^2)$ via rejection sampling as shown in Algorithm 3. We then add up the individual probabilities for these values and store them in a table. This way we only need to generate a uniform random number and then perform a table lookup at runtime to get a sample from the target discrete Gaussian distribution.

4 Implementation platforms

In this section, we introduce our implementation platforms including PC[2] Web browsers (Google Chrome, Firefox, Opera, and Internet Explorer), Android devices (ASUS MeMO Pad7 and Nexus7), and Tessel.

[2]The test PC has the following specifications:
CPU: Intel(R) Core(TM) i7-4710MQ @ 2.5GHz;
Memory: 8GB DDR3 RAM;
Hard disk: 1TB 5400rpm;
OS: Windows 8.1 build 6.3.9600 Pro x64;
Java(JDK) version: jdk1.7.0_67;
Tomcat version: apache-tomcat-7.0.55;
JavaScript version: 1.3;
Web browser: Google Chrome 41.0.2272.101 m; Firefox 36.0.4; Opera 28.0.1750.48; Internet Explorer 11.0.9600.17690.

4.1 PC Web browsers

Most mainstream Web browsers nowadays have very good support for JavaScript. JavaScript allows multiple tasks to execute only in the browser client without networking and servers support. It achieves so without needing to load any virtual machines and thus is ideal for distributed computing and processing. In this implementation, we have chosen four desktop PC browsers as our benchmark platforms, namely, Google Chrome, Firefox, Opera and Internet Explorer. Interestingly, we have found that the efficiency of running our same JavaScript program is quite different across these four browsers, which will be described in more detail in the next section. In our implementations we choose Opera to be the main test platform because its CSPRNG has the best performance.

4.2 Android

On the Android operating system, there are also a variety of Web browsers. In addition, the Android SDK has a built-in, high-performance browser component called WebView. WebView can be used to display Web pages in a non-browser application, making it easy to open hyperlinks without bringing up a full-blown Web browser. It also supports JavaScript, so we have chosen WebView for Android 4.4.2 to benchmark our implementation. We have run our experiments on two Android tablets: ASUS MeMO Pad7 ME572C[3] and Nexus7[4].

4.3 Tessel

Tessel is an embedded system designed for IoT applications. It runs JavaScript for controlling a wide variety IoT devices on top of a 180 MHz ARM Cortex M3 (LPC1830) microcontroller with 32 MB of SDRAM and 32 MB of Flash memory. Tessel supports "Node-compatible" JavaScript, which usually runs on the server side and is slightly different from that runs on the client side, as it does not need to deal with user interfaces. Tessel allows to execute JavaScript programs directly, but due to the hardware limitation, the efficiency is not so good.

5 Performance on Web browsers

In this section, we report the performance results of running our target lattice-based encryption schemes on several PC Web browsers. We will also compare the results with that of 3072-bit RSA and ElGamal ECC on Opera. We do not claim any optimality of our implementation when it comes to, e.g., cache/memory usage. In contrast, we mostly focus on algorithmic improvement and portability across different platforms.

5.1 Performance results on Opera

Table 3 shows the performance results of various schemes executed on the Opera browser. At a first glance, encryption is fast and typically takes less than 10 ms except Regev's LWE and 3072-bit RSA. As we have expected, Regev's LWE is the slowest in key generation and encryption because of the large matrix size. Decryption is also fast, and even the slowest NTRU takes less than 6 ms. Overall,

[3]
ASUS MeMO Pad7 ME572C:
Android version: 4.4.2;
CPU: Intel(R) Atom(TM) Z3560, 1.83 GHz(Quad-Core), 64bit;
Internal Flash memory: 16GB;
RAM: 2GB.

[4]
Nexus7:
Android version: 4.4.2;
CPU: Nvidia Tegra 3 / 1.3 GHz (Quad-core);
Internal Flash memory: 8GB;
RAM: 1GB.

Table 3: Performance results on Opera

Scheme	Average running time (ms)			Bit-security
	Key generation	Encryption	Decryption	
NTRU	165.382	8.180	5.848	128
NTRU-IEEE07	119.774	7.241	3.114	128
Regev's LWE	2162.051	79.752	3.475	128
LPR10-LWE	1.642	3.247	1.612	126
LP10 Ring-LWE	1.636	3.244	1.623	126
LP11	49.192	2.256	1.034	≥ 128
3072-bit RSA (Baird) [3]	15950.10	7.75	229.65	128
3072-bit RSA (MSR) [23]	N/A[1]	10.73	378.88	128
ElGamal ECC (NIST P-256) [31]	17.82	38.24	20.92	128

[1] We have to import RSA key pairs from outside because otherwise it would take too much time.

lattice-based schemes seem reasonably fast. Similar results have also been reported in the literature that Ring-LWE based schemes are more efficient than the original LWE-based schemes [12].

Furthermore, the decomposition of computation time is shown in Table 4. In key generation, computation of polynomial inverses modulo p and q is the most time consuming operation, which accounts for close to 80% of the total time. Excluding those in the inverse calculation, the other polynomial multiplications account for less than 10% of the total time. In LPR10-LWE and LP10 Ring-LWE, polynomial multiplication using Algorithm 2 (NTT) is the single bottleneck computation, accounting for at least 90%. We have also tried Algorithm 1 (Karatsuba), which is about 3% slower. Discrete Gaussian sampling takes less than 10%, while the other time is spent on addition and modulus calculation.

In contrast to some other schemes in the literature, discrete Gaussian sampling costs very little. In Regev's LWE, most time is spent in matrix multiplication (accounting for close to 80%), which is also the case for LP11 (70%). For encryption and decryption, the time is almost spent in multiplication and accounts for more than 70% of the total time.

Compared with matrix-based schemes such as LP11, ring-based schemes such as LP10 Ring-LWE is more space efficient. LP11 also takes more time than LP10 Ring-LWE in key generation. However, in encryption and decryption, LP11 is faster.

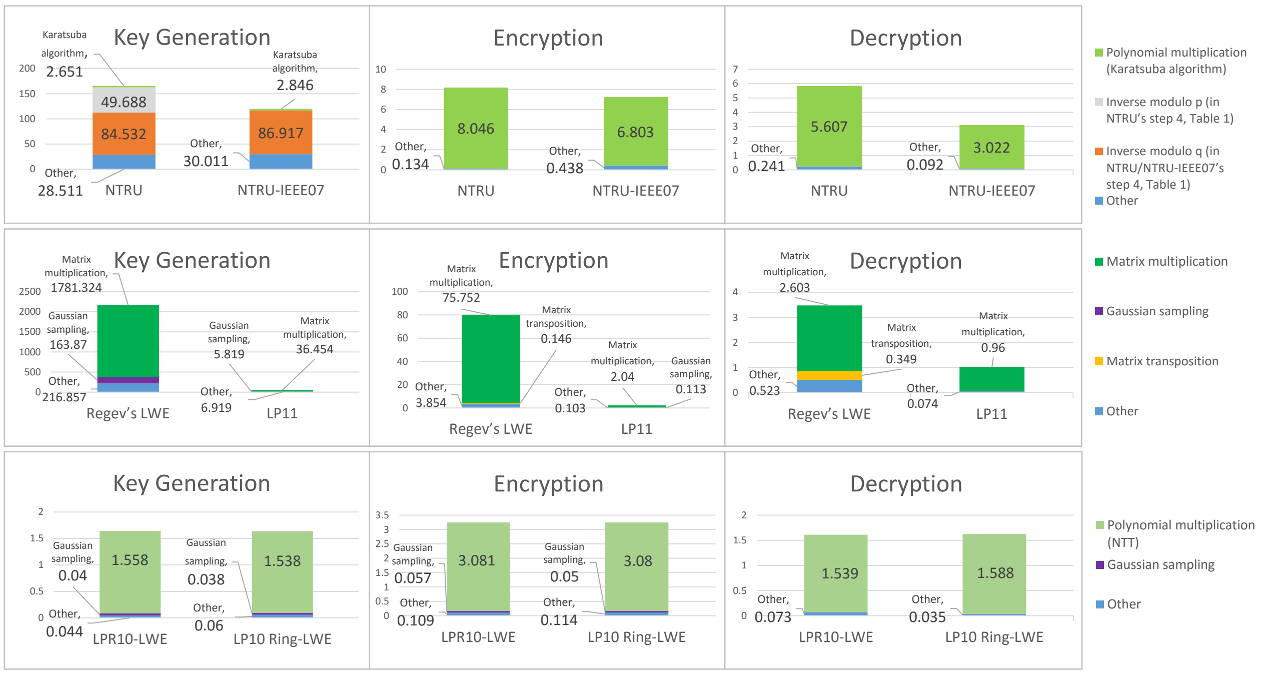
Compared with NTRU and NTRU-IEEE07, although LP10 Ring-LWE has large moduli and polynomial coefficients, the key size is actually smaller, c.f. Table 2. Furthermore, for LP10 Ring-LWE we no longer need to compute polynomial inverses. It can be seen that the running time of key generation of LP10 Ring-LWE is less than NTRU because computing polynomial inverses can take up a considerable amount of time. Nevertheless, the cost for encryption and decryption is similar, which is evident from the fact that the speeds are all very close.

Finally, we note that the state-of-the-art implementations of lattice-based schemes often use native libraries for efficiency consideration, e.g., Cabarcas, Weiden, and Buchmann [9] using the Number Theory Library (NTL) [30] to implement LPR10-LWE. In contrast, we use JavaScript for portability consideration. Compared with the native implementation by Cabarcas, Weiden, and Buchmann [9], our JavaScript implementation is no more than twice slower under the same parameter setting, indicating that today's JavaScript engines are already quite efficient for carrying out intensive PKC computation.

5.2 Comparison against prevailing public-key cryptosystems

We have chosen 3072-bit RSA and ElGamal ECC for comparison with the lattice-based schemes at the 128-bit security level. In our RSA implementation, we use Baird's library [3] and the Microsoft

Table 4: Decomposition of computation time (ms) on Opera



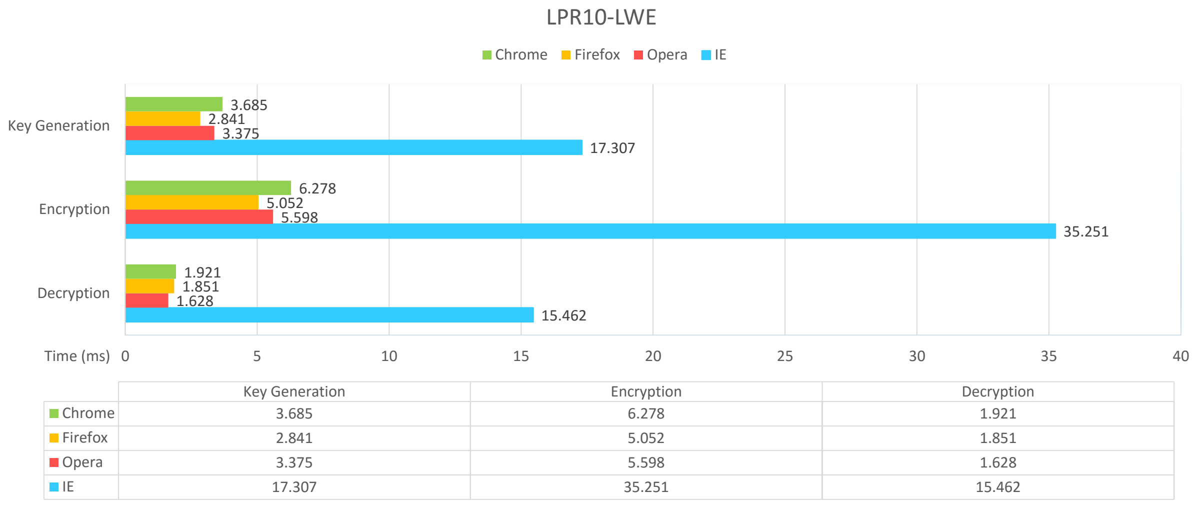
Research JavaScript Cryptography Library [23] for multi-precision integer arithmetic; in the rest of this paper, we shall refer to the former as RSA (Baird) and the latter as RSA (MSR). We also use the well-known method based on Chinese Remainder Theorem (CRT) to accelerate RSA decryption. For ElGamal ECC, we use the Stanford Javascript Crypto Library (SJCL) [31].

The maximum numeric value representable in JavaScript is approximately 1.79×10^{308}, or 2^{1024}. This is not enough for carrying out the computation in 3072-bit RSA encryption and decryption. In RSA (Baird), most time is spent on generating p and q in key generation, as they exceed the limit of what JavaScript number object can represent. Take the performance results on Opera as an example: it takes 15950.10 ms to generate a 3072-bit RSA key, while the generation of p and q alone takes 15917.78 ms. Overall, key generation of 3072-bit RSA is more than 9000 times slower than that of Ring-LWE schemes; encryption is more than 2 times slower, and decryption, more than 140 times slower. Even without considering key-generation time, Ring-LWE schemes are still much more efficient than RSA (Baird).

Recently Microsoft Research released the latest version of the MSR JavaScript cryptography library [23], which supports RSA encryption and decryption. This library provides a cryptography object called `window.msrCrypto`. It is, however, different from `window.msCrypto` used in Internet Explorer 11. In addition, newer non-Microsoft browsers seem to be using `window.crypto`. RSA (MSR) supports the JSON Web Key (JWK) format [18] for storing key pairs and parameters. However, `window.msrCrypto` could not be used to generate the RSA key pairs because it would spend so much time. Instead, we need to import the key pairs from outside. According to our experimental results, the speed of 3072-bit RSA (MSR) is slower than that of RSA (Baird) in that encryption and decryption are about 1.5 times slower. However, as Web Worker is a feature of HTML5, some platforms such as Tessel and several web browsers do not yet have the support for it, so this library is not portable at this point.

We have also tried the ElGamal public key encryption scheme over the NIST P-256 curve, which offers 128-bit security. SJCL supports elliptic curve cryptography, based on which we have imple-

Table 5: Performance comparison across Web browsers on PC



	Key Generation	Encryption	Decryption
🟩 Chrome	3.685	6.278	1.921
🟨 Firefox	2.841	5.052	1.851
🟥 Opera	3.375	5.598	1.628
🟦 IE	17.307	35.251	15.462

mented the ElGamal public key encryption scheme. In key generation, encryption, and decryption, ElGamal ECC is more than 10 times slower than LPR10-LWE.

These prevailing public-key cryptosystems such as RSA and ElGamal ECC cannot resist attacks from large quantum computers. Moreover, some of them are actually slower than the fast lattice-based encryption schemes. Therefore, we are confident that lattice-based cryptography will soon find applications in practice, perhaps starting from replacing some of the existing cryptosystems.

5.3 Performance on other web browsers

We have also tested our JavaScript implementation on several other Web browsers on PC. Take LPR10-LWE as an example: Table 5 shows the running time on these browsers. Notice that `window.crypto.getRandomValues()` function is invoked to generate random numbers. For Internet Explorer, it is replaced by `window.msCrypto`, as `window.crypto` is undefined there.

It appears that Google Chrome has similar performance with Opera, both of which are based on Blink. The performance of Internet Explorer is more than 5 times slower than Opera. Overall, Firefox delivers the best performance.

6 Performance on small devices

In this section, we report the performance results and compare them with 3072-bit RSA and ElGamal ECC. We use Baagøe's implementation of PRNG [2] on Android tablets and Tessel. 3072-bit RSA (MSR) and ElGamal ECC are only tested on Android tablets because `window` object cannot run on Tessel. As we have seen in Section 5, we are mainly interested in algorithmic aspects and the portability of the implementation.

6.1 Performance on Android devices

WebView and Android Web browsers are different from those on PC. As Android mostly run on devices with a touchscreen, the event trigger mechanisms are different. Also, typical Android devices have a configuration of less computational power than PC, so the speed performance on Android is not as good as that on PC browsers. However, the decomposition of running time for each of

Table 6: Performance results on Android

Scheme	Average running time (ms)					
	Key generation		Encryption		Decryption	
	MeMO Pad7	Nexus7	MeMO Pad7	Nexus7	MeMO Pad7	Nexus7
NTRU	458.91	851.39	17.24	37.59	15.35	35.05
NTRU-IEEE07	355.80	669.60	18.09	40.57	8.03	18.79
Regev's LWE	9653.30	21437.05	382.45	871.15	12.60	26.85
LPR10-LWE	5.03	9.59	9.11	19.39	4.31	8.79
LP10 Ring-LWE	5.09	10.19	9.13	18.77	4.26	8.92
LP11	212.88	492.46	15.92	31.30	7.69	18.18
3072-bit RSA (Baird) [3]	40959.3	101316.6	30.5	72.2	794.0	2082.7
3072-bit RSA (MSR) [23]	N/A	N/A	97.1	169.6	2192.2	4400.3
ElGamal ECC (NIST P-256) [31]	74.37	137.60	154.41	283.34	77.37	142.54

Table 7: Performance comparison of LP11 across Android browsers

Android browser	Average running time (ms)					
	Key generation		Encryption		Decryption	
	MeMO Pad7	Nexus7	MeMO Pad7	Nexus7	MeMO Pad7	Nexus7
Google Chrome	197.83	476.33	8.38	17.52	3.04	6.77
Firefox	187.86	489.06	1.63	5.26	0.81	2.12
Opera	195.64	499.17	8.34	17.47	2.99	6.86
WebView	212.88	492.46	15.92	31.30	7.69	18.18

the schemes is almost the same as on PC browsers. From Table 6, we see that the ASUS MeMO Pad7 delivers better performance than the Nexus7, while the performance on either device is quite acceptable.

As can be seen from the results in Table 6, Ring-LWE schemes provide the best performance results on WebView, around twice as fast as LP11 in both encryption and decryption. We have also tested our implementations on other Android web browsers like Google Chrome 50.0.2661.89, Firefox 46.0, and Opera 36.2.2126.102826. For Ring-LWE schemes, the performance of these browsers is very similar to that of their PC counterparts. However, we have found that some test data of WebView activity are significantly different from that of Android web browsers. For example, Table 7 shows the running time of LP11 on ASUS MeMO Pad7 and Nexus7. The difference depends both on the JavaScript runtime architecture and platform used in experiments, and WebView can deliver different performance for lattice-based encryption schemes from other Android web browsers.

6.2 Performance on Tessel

Tessel can run JavaScript programs directly. However, on Tessel a JavaScript program does not run inside any browser but directly supported by the system. We have successfully tested all our implementation on Tessel except Regev's LWE, which fails to run due to insufficient memory.

Table 8 shows the running time of our implementations on Tessel. Note that the execution time is in *seconds*. In general, the performance achieved on Tessel is several orders of magnitude slower. For example, key generation of NTRU on Tessel is more than 5000 times slower than that on Opera, as well as more than 1000 times slower than that on ASUS MeMO Pad7. Key generation, encryption, and decryption of LPR10-LWE are more than 12000 times slower than that of on Opera.

We note that on Tessel, it would take more than a few days to generate large integers p and q for 3072-bit RSA (Baird). Therefore, it is excluded from the results shown in Table 8. We conclude that

Table 8: Performance results on Tessel

Scheme	Average running time (s)			Bit-security
	Key generation	Encryption	Decryption	
NTRU	886.635	28.767	53.540	128
NTRU-IEEE07	752.350	29.283	26.769	128
Regev's LWE	Insufficient memory			128
LPR10-LWE	21.324	41.816	19.880	126
LP10 Ring-LWE	21.369	41.690	19.702	126
LP11	1259.900	12.229	4.457	≥ 128
3072-bit RSA (Baird) [3]	2952.874[2]	456.647	18459.746	128

[2] This is the running time of key generation on Tessel without generating big integers p and q, as it would take a very long time to generate p and q on Tessel.

it is not practical to run 3072-bit RSA (Baird) on such a small device like Tessel using JavaScript.

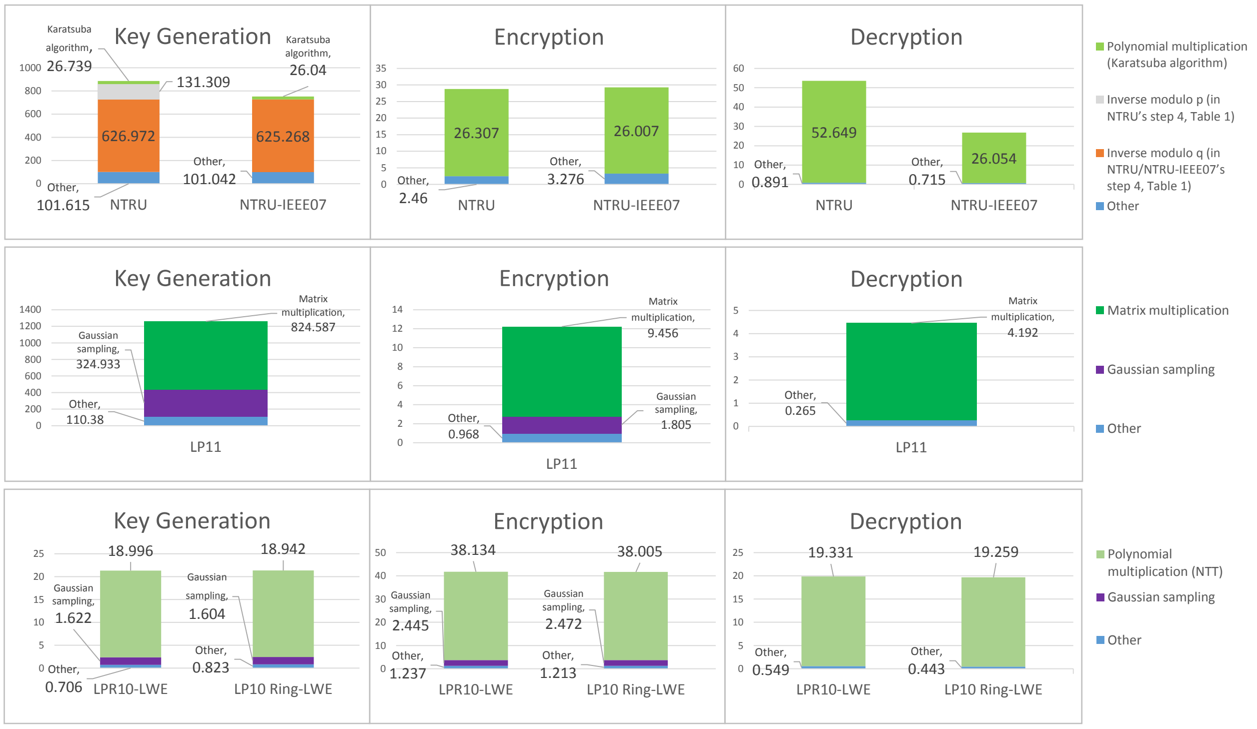
Such a huge performance gap between running JavaScript on PC vs. on Tessel cannot simply be explained by the difference of CPU. A typical PC's CPU runs at a few GHz, which can perhaps explain a factor of 10 to 20 in performance difference. Besides, PC's CPU usually has several arithmetic units and complicated out-of-order execution engines, as well as larger on-die cache memory, all of which can help harvest instruction-level parallelism and data locality. Together, they can perhaps explain another factor of 10 to 20. However, this still leaves us a factor of 100 in performance difference, which we need other explanation such as the one below. In JavaScript, there is only one numerical type, the IEEE double-precision floating-point numbers. Unfortunately, the ARM Cortex M3 microcontroller generally does not have hardware support for such floating-point arithmetic. Therefore, all arithmetic is emulated using the (already slow) 32-bit integer arithmetic on Cortex M3. Our preliminary experiment shows that the Cortex M3 microcontroller on Tessel can execute about 36000 multiplications per second with a deeply unrolled JavaScript program. The same JavaScript program can run more than 15000 times faster on a PC, which more or less agrees with the performance gap we have observed.

6.3 Detailed running time on Tessel

The speed performance of our implementation is relatively fast on PC browsers and the Android device but slow on Tessel. In Table 9, we see that in key generation of the NTRU family of algorithms, computation of polynomial inverse accounts for about 80% of the time, while polynomial multiplication accounts less than 10% of the time. In LPR10-LWE and LP10 Ring-LWE, polynomial multiplication almost occupies more than 90% of the key generation time, while Gaussian sampling takes very little time. It appears that polynomial multiplication is indeed the bottleneck operation for encryption and decryption as well.

For LP11, matrix multiplication takes up around 70% of the running time. The reason why LP11 can run on Tessel is mainly because the key size is much smaller than Regev's LWE. In addition, discrete Gaussian sampling takes a certain proportion of the running time, especially in key generation. For example, nearly 20% of the key generation time is spent on discrete Gaussian sampling for the LWE-based schemes. Overall, ring-based LWE schemes, LPR10-LWE and LP10 Ring-LWE are the most efficient in terms of total amount of time spent in key generation, encryption, and decryption on Tessel, as they use polynomial operations and have smaller key sizes. In addition, when the key size is small enough, matrix-based schemes such as LP11 can run successfully on Tessel.

Table 9: Decomposition of computation time (s) on Tessel



7 Conclusions

We have implemented several lattice-based encryption schemes using JavaScript and tested their performance on multiple computing platforms. Overall, we have found that Ring-LWE-based schemes are the most efficient, as polynomial multiplication is generally much faster than matrix operations. In addition to Web browsers on PC and Android, small embedded systems like Tessel can also execute our JavaScript implementations directly, and we expect more and more such small devices to have JavaScript support in the future.

References

[1] Miklos Ajtai. "Generating hard instances of lattice problems." In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pp. 99–108, 1996.

[2] Johannes Baagøe. "Better random numbers for Javascript." `https://github.com/nquinlan/better-random-numbers-for-javascript-mirror`, 2013.

[3] Leemon Baird. `http://www.leemon.com/crypto/BigInt.html`, 2013.

[4] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. "Recommendations for key management — Part 1: General (Revision 3)." NIST Special Publication 800-57 Part 1 Revision 3, 2012.

[5] Ahmad Boorghany and Rasool Jalili. "Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers." Cryptology ePrint Archive, Report 2014/078, 2014.

[6] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. "On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards." Cryptology ePrint Archive, Report 2014/514, 2014.

[7] Tim Buktu. "NTRU: Quantum-resistant cryptography." `https://github.com/tbuktu/ntru`, 2013.

[8] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "Efficient software implementation of Ring-LWE encryption." Cryptology ePrint Archive, Report 2014/725, 2014.

[9] Daniel Cabarcas, Patrick Weiden, and Johannes Buchmann. "On the efficiency of provably secure NTRU." In *Proceedings of PQCrypto 2014*, pp. 22–39, 2014.

[10] Leo Ducas and Phong Q. Nguyen. "Faster Gaussian lattice sampling using lazy floating-point arithmetic." In *Proceedings of ASIACRYPT 2012*, pp. 415–432, 2012.

[11] Tore Kasper Frederiksen. "A practical implementation of Regev's LWE-based cryptosystem." Technical report, 2010. `http://daimi.au.dk/~jot2re/lwe/`

[12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. "On the design of hardware building blocks for modern lattice-based encryption schemes." In *Proceedings of CHES 2012*, pp. 512–529, 2012.

[13] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In *Proceedings of STOC 2008*, pp. 197–206, 2008.

[14] Michael T. Goodrich and Roberto Tamassia. "Algorithm design: Foundations, analysis, and internet examples." Wiley 2001.

[15] Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. "Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches." In *Proceedings of ACNS 2009*, pp. 437–455, 2009.

[16] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. "JAMA: A Java matrix package." `http://math.nist.gov/javanumerics/jama/`, 2012.

[17] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. "NTRU: A ring-based public key cryptosystem." In *Proceedings of ANTS 1998*, pp. 267–288, 1998.

[18] `https://github.com/mitreid-connect/mkjwk.org`, 2014.

[19] Richard Lindner and Chris Peikert. "Better key sizes (and attacks) for LWE-based encryption." Cryptology ePrint Archive, Report 2010/613, 2010.

[20] Richard Lindner and Chris Peikert. "Better key sizes (and attacks) for LWE-based encryption." In *Proceedings of CT-RSA 2011*, pp. 319–339, 2011.

[21] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." In *Proceedings of EUROCRYPT 2010*, pp. 1–23, 2010.

[22] Daniele Micciancio and Oded Regev. "Lattice-based cryptography." In *Post-Quantum Cryptography*, pp. 147–191. Springer, 2008.

[23] MSR JavaScript Cryptography Library, version 1.4. `http://research.microsoft.com/en-us/downloads/29f9385d-da4c-479a-b2ea-2a7bb335d727/`, 2015.

[24] Phong Q. Nguyen and Jacques Stern. "The two faces of lattices in cryptology." In *Proceedings of Cryptography and Lattices Conference (CaLC 2001)*, pp. 146–180, 2001.

[25] https://code.google.com/p/pts-mini-gpl/source/browse/trunk/javascript-multiplication/karaperf.js, 2009.

[26] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. "A framework for efficient and composable oblivious transfer." In *Proceedings of CRYPTO 2008*, pp. 554–571, 2008.

[27] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography." *Journal of the ACM*, 56(6):34, pp. 1–40, 2009.

[28] Markus Ruckert and Michael Schneider. "Estimating the security of lattice-based cryptosystems." Cryptology ePrint Archive, Report 2010/137, 2010.

[29] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. "Compact Ring-LWE cryptoprocessor." In *Proceedings of CHES 2014*, pp. 371–391, 2014.

[30] Victor Shoup. "NTL: A Library for doing Number Theory." http://www.shoup.net/ntl/, 2016.

[31] Stanford Javascript Crypto Library (SJCL). https://github.com/bitwiseshiftleft/sjcl, 2015.

[32] Volker Strassen. "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 13, issue 4, pp. 354–356, 1969.

[33] Robert Sedgewick and Kevin Wayne. http://introcs.cs.princeton.edu/java/stdlib/StdRandom.java.html, 2014.

[34] "Web Cryptography API." http://www.w3.org/TR/WebCryptoAPI/, 2014.

[35] "The JavaScript world domination." https://medium.com/@slsoftworks/javascript-world-domination-af9ca2ee5070, 2015.

[36] William Whyte (editor), Nick Howgrave-Graham, Jeff Hoffstein, Jill Pipher, Joseph H. Silverman, and Phil Hirschhorn. "IEEE P1363.1 Draft 10: Draft standard for public-key cryptographic techniques based on hard problems over lattices." Cryptology ePrint Archive, Report 2008/361, 2008.