Quantum computers have the potential to solve some difficult mathematical problems efficiently and thus will inevitably exert a more significant impact on the traditional asymmetric cryptography. The National Institute of Standards and Technology (NIST) has opened a formal call for the submission of proposals of quantum-resistant public-key cryptographic algorithms to set the next-generation cryptography standards. Compared to powerful machines with ample amount of hardware resources such as racks of servers and IoT devices, including the massive number of microcontrollers, smart terminals, and sensor nodes with limited computing capacity, should also have some postquantum cryptography features for security and privacy. To ensure the correct execution of encryption algorithms on any platforms, the portability of implementation becomes more important. As distinguished from C/C++, JavaScript is a popular cross-platform language that can be used for the web applications and some hardware platforms directly, and it could be one of the solutions of portability. Therefore, we investigate and implement several recent lattice-based encryption schemes and public-key exchange protocols including Lizard, ring-Lizard, Kyber, Frodo, and NewHope in JavaScript, which are the active candidates of postquantum cryptography due to their applicabilities and efficiencies. We show and compare the performance of our JavaScript implementation on web browsers, embedded device Tessel2, Android phone, and several JavaScript-enabled platforms on PC and Mac. Our work shows that implementing lattice-based cryptography on JavaScript-enabled platforms is achievable and results in desirable portability.

1. Introduction

The rapid development of quantum computing coupled with Shor’s algorithm [1] brings a significant threat to widely used RSA and elliptic curve cryptography (ECC) based on the integer factorization and the discrete logarithm problems. Hence, postquantum cryptography (PQC) has generated a lot of attention among researchers. In the IoT era, tons of things or devices will get connected to the Internet, and they require efficient quantum-resistant approaches to protect the security and privacy. IoT software should work correctly on any architecture; therefore, the portability of software becomes more important. Besides, web browsers serve as an essential platform for web applications and should also have postquantum cryptographic features. As a favorite cross-platform/browser language, JavaScript is one of the solutions of the portability because its performance has improved considerably over the past few years.

Lattice-based cryptography, which is thought to be secure against attacks by quantum computers [2], has gained wide attention and deep researches from academia to industry due to its efficiency and applicability. In recent years, some derivatives of encryption schemes and key exchange protocols of lattice-based cryptography were presented, such as [3–7]. Implementations of those cryptosystems have been reported in some literature [8–12]. However, as of now, there is very little research on lattice-based cryptography in JavaScript [13, 14]. Therefore, we would like to investigate the performance
of several recent lattice-based cryptosystems on modern computing platforms with JavaScript implementation. We hope to contribute to the practical implementation of PQC.

We implemented and tested five recent lattice-based encryption schemes and public-key exchange protocols on four web browsers, a microcontroller Tessel2, an Android phone Xperia XZ, and other JavaScript-enabled platforms on PC and Mac. We chose an encryption scheme “Lizard” which is based on the learning with errors (LWE) and the learning with rounding (LWR) problems and its ring variant “ring-Lizard” [15], a modulo-LWE based encryption scheme “Kyber” [16], and two quantum secure key exchange protocols “Frodo” [17] and “NewHope” [18], which are based on the LWE problem and the ring-LWE problem, respectively. All the cryptosystems above were implemented in JavaScript. The source code of our implementation can be found at https://github.com/FuKyuToTo/lattice-based-cryptography.

To provide a fair comparison, we selected the parameters which have 128 bits of postquantum security from the estimation of Jung Hee Cheon et al. [15], Joppe Bos et al. [16, 17], and Erdem Alkim et al. [18], summarized in Table 1. However, there are many different models to estimate the secure parameters of lattice-based cryptography [19]. The analysis of the concrete quantum security levels of those parameters is beyond the scope of this paper, more detailed security estimation algorithms can be found in [20–22]. Our parameters should be rescaled after finalizing the secure parameters in NIST PQC standardization project in a future work. By implementing the improved number-theoretic transform (NTT) and inverse NTT (see [23, 24]) and reducing the memory overhead of creating temporary instances, we vastly improve the efficiency of polynomial operations compared with our previous work (see [13]).

(iii) Our implementation has good portability and scalability. Our JavaScript code can be directly executed on any JavaScript runtime environment without modification. More importantly, by comparing and analyzing these performance difference, we can further improve our implementation for particular platforms.

Table 1: Summary of the selected parameters that provide about 128-bit security.

<table>
<thead>
<tr>
<th>Cryptosystem</th>
<th>m</th>
<th>n</th>
<th>l = p</th>
<th>q</th>
<th>t</th>
<th>α⁻¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>960</td>
<td>608</td>
<td>256</td>
<td>1024</td>
<td>2</td>
<td>182</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>p</td>
<td>α⁻¹</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Kyber</td>
<td>k</td>
<td>n</td>
<td>q</td>
<td>η</td>
<td>d_k = d_{η}</td>
<td>d_{η}</td>
</tr>
<tr>
<td>Frodo</td>
<td>b</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>σ</td>
<td>3</td>
</tr>
<tr>
<td>NewHope</td>
<td>16</td>
<td>1024</td>
<td>12289</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

The rest of this paper is organized as follows. We will explain the notation, give a brief introduction of the mathematical background, and introduce the implemented cryptosystems in Section 2. We will introduce our experimental platforms in Section 3 and describe our implementation techniques in Section 4. We will then present the performance reports on web browsers in Section 5 and on IoT device Tessel2, Android phone, and other platforms in Section 6. Finally, we conclude this paper in Section 7. The appendix section contains an example of the usage of our source code.

2. Lattice-Based Cryptography

In this section, we introduce the relevant mathematical background for the LWE, ring-LWE, and LWR problems and summarize the postquantum cryptographic schemes based on those problems.

2.1. Notation. Let \( n, q \) be positive integers; we denote \( \mathbb{Z}_q \) as the set of integers \( \{0, 1, \ldots, q - 1\} \) and \( R = \mathbb{Z}[x]/(x^n + 1) \), \( R_q = \mathbb{Z}_q[x]/(x^n + 1) \) as the polynomial rings. Polynomials are denoted by bold italic letters such as \( \mathbf{a} \), while vectors are...
denoted by bold small letters such as \( v \) and matrices and bold
large letters such as \( A \). For an integer \( m \in \mathbb{N} \), we define
the modulo operation \( b \equiv a \mod m \) in the range \([0, m) \cap \mathbb{Z} \).

2.2. LWE, Ring-LWE, and LWR Problems. Regev proposed
Let \( m, n, q \) be positive integers; the search LWE problem is
required to find a secret vector \( s \in \mathbb{Z}_q^n \) by inputing a pair of
matrices \((A, b) = As + e \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m \). The decision LWE
problem is to distinguish \( b \) between a uniformly distributed
random vector from \( \mathbb{Z}_q^m \) and a noisy inner product \( b = As + e \).
Usually, the elements of \( A \) are randomly selected from \( \mathbb{Z}_q^n \),
and the so-called error vector \( e \in \mathbb{Z}_q^m \) is sampled from a
target probability distribution \( \chi \). The cryptography based on
the LWE problem uses an unusual structure lattice which is
called \( q \)-ary lattice:

\[
L_q^\perp(A) = \{ v \in \mathbb{Z}_q^n \mid Av \equiv 0 \mod q \};
\]

\[
L_q(A) = \{ v \in \mathbb{Z}_q^n, s \in \mathbb{Z}_q^m \mid v = ATs \mod q \};
\]

all the elements of the \( q \)-ary lattice are obtained using an
integer modulo of \( q \).

The ring-LWE problem (see [5]) is a variant of Regev’s
original LWE problem. \( R \) is an ideal lattice if every polynomial
over \( R \) has a bijective mapping to an ideal \( \mathbb{Z}_q^n \). Given
polynomials \( a, b \in R_q \), the search version of the ring-LWE
problem is to recover the secret \( s \in \mathbb{Z}_q^n \), where \( a \) is chosen
uniformly and \( b = a \cdot s + e \) with an “error” \( e \in R \) sampled
from a target probability distribution \( \chi \). The decision ring-LWE
problem is similar to the decision LWE problem: given
\( a, b \in R_q \), we distinguish whether \( b \) is also chosen uniformly,
or there exists a polynomial \( s \in R_q \) such that \( b = a \cdot s + e \).
If there were not any error adding, the LWE and ring-LWE
problems would be the simple linear algebra computation and
easy to solve. In the worst-case, such LWE and ring-LWE
problems can be reduced to the approximate versions of NP-
hard shortest vector problem (\( \alpha \)-SVP) on ideal lattices.

Given a matrix \( A = [a_1, \ldots, a_n] \in \mathbb{Z}_q^{m \times n} \) and an inner
product with rounding \( b = [As]_p \in \mathbb{Z}_p^n \), the LWR problem
(see [25]) is to find the vector \( s \in \mathbb{Z}_p^n \) where \( p \ll q \). The
information hiding technique or so-called derandomization
algorithm of LWR is different from LWE: each value of
the inner product \( b \) times a rounded value \( \lfloor q/p \rfloor \) over \( \mathbb{Z}_q^p \),
instead of adding a random error value; therefore, the error in LWR
is deterministic.

2.3. Discrete Gaussian Sampling. For a real \( \sigma > 0 \), the
Gaussian distribution evaluated at \( x \in \mathbb{R} \) is defined by
\( \rho_\sigma(x) = \exp(-x^2/\sigma^2) \), where the Gaussian parameter \( s = \sigma \sqrt{2\pi} \). A discrete version of Gaussian distribution over \( \mathbb{Z} \) is defined by
\( D_\sigma(x) = \rho_\sigma(x)/\rho_\sigma(0) \). In order to find out where to drop
the negligible probability of far samples, a tail-cut factor \( t > 0 \)
is set to determine the range of sampled values. Choosing
a suitable length of the tail-cut factor for a target discrete
Gaussian distribution is necessary; otherwise, no sampling
algorithm could cover it. The tail-bound is closely related
to the maximum statistical distance allowed by the security
discrete Gaussian parameters [26, 27]. Note that sampling
values from the discrete Gaussian distribution are different
to sampling from a normal distribution [28]. We implement
modified Knuth-Yao algorithm [27, 29] and modified discrete
Ziggurat algorithm [30] to perform such a sampling. The
sampling methods will be discussed in Section 4.1.

2.4. Binomial Distribution. The binomial distribution is a
discrete probability distribution of the successful number in
\( n \) Bernoulli trials. In this paper, we follow the definition in
[16, 18] and denote \( B_k \) as the centered binomial distribution
for a positive integer \( k \):

Input: a binary string \( \{a_0, a_1, \ldots, a_k-1, b_0, b_1, \ldots, b_{k-1}\} \leftarrow \{0, 1\}^k \)

Output: an integer \( \sum_{i=0}^{k-1} (a_i - b_i) \)

For the convenience of calculations, we only sample and
compute integers over \( \mathbb{Z}_q \).

2.5. Lizard and Ring-Lizard. Lizard encryption scheme [15] is
parameterized by positive integers \( h, m, n, l, t, p, q \in \mathbb{Z} \) and
an error rate \( \alpha \in \mathbb{R} \), where the moduli \( t, p, q \) satisfy \( t | p \mid q \).
For a real number \( 0 < \rho < 1 \), we sample values \( \{v_1, v_2, \ldots, v_n\} \leftarrow \{-1, 0, 1\}^n \) from the distribution \( \mathcal{D}_{\Omega_q}(\rho) \) such that each value \( v_i \) \((i = 1, 2, \ldots, n) \) is chosen satisfying \( \Pr[v_i = 0] = 1 - \rho \) and
\( \Pr[v_i = 1] = \Pr[v_i = -1] = \rho/2 \). For an integer \( 0 < h \leq m \), we sample the values \( \{v_1, v_2, \ldots, v_n\} \leftarrow \{-1, 0, 1\}^n \)
from the distribution \( \mathcal{D}_{\Omega_q}(h) \) such that it has exactly \( h \) nonzero
entries in those values.

In key generation, we choose a matrix \( S = [s_1, \ldots, s_t] \in \mathbb{Z}_q^{m \times l} \)
by sampling column vectors \( s_i \in \mathbb{Z}_q^n \) \((i = 1, 2, \ldots, l) \)
independently from the distribution \( \mathcal{D}_{\Omega_q}(1/2) \). Input a
matrix \( A \in \mathbb{Z}_q^{m \times n} \) whose elements are chosen uniformly from
\( \mathbb{Z}_q^n \); then we can compute the matrix \( B = AS + E \in \mathbb{Z}_q^{m \times l} \),
where the error matrix \( E \in \mathbb{Z}_q^{m \times l} \) is chosen according to \( D_{\Xi_{\mathbb{Z}_q}} \).
The secret key is \( S \) and the public key is the pair \((A, B) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^{m \times l} \). In encryption,
by choosing a random vector \( \mathbf{r} \in \mathbb{Z}_q^m \) from the distribution \( \mathcal{D}_{\Xi_{\mathbb{Z}_q}}(128) \), we compute a
couple \((\mathbf{c}_1, \mathbf{c}_2) = (A^T \mathbf{r}, B^T \mathbf{r}) \in \mathbb{Z}_q^m \times \mathbb{Z}_q^l \).
Given a message \( \mathbf{m} \in \mathbb{Z}_q^l \), the ciphertext is the pair \((\mathbf{c}_1, \mathbf{c}_2) \),
where \( \mathbf{c}_1 = [(p/q) \cdot \mathbf{c}_1] \in \mathbb{Z}_p^m \)
and \( \mathbf{c}_2 = [(p/t) \cdot \mathbf{m} + (p/q) \cdot \mathbf{c}_2] \in \mathbb{Z}_l^l \). Lastly, we output the
vector \( \mathbf{m}' = [(t/p) \cdot (\mathbf{c}_1 - \mathbf{c}_2')] \in \mathbb{Z}_p^m \) in decryption.

Ring-Lizard encryption scheme [15] is a variant of Lizard
and based on the hardness of the ring-LWE and the ring-
LWR problems. It exploits better key sizes and delivers faster
speed of encryption and decryption compared with Lizard.
The following procedures define the ring-lizard scheme.

Key Generation. Sample \( e \leftarrow D_{\Xi_{\mathbb{Z}_q}} \); choose a “small”
random polynomial \( s \) from \( \mathcal{D}_{\Xi_{\mathbb{Z}_q}}(128) \) and a uniformly
random polynomial \( a \in R_q \); then output the public key
\((a, b = a \cdot s + e) \in R_q \times R_q \) and the secret key \( s \in R \).
Encryption. Choose a random polynomial $r$ from $\mathbb{Z}/W \mathbb{T}_a(128)$; given a plaintext $m \in \{0,1\}^n$, then compute $c_1 = [(p/q) \ast (a \cdot r)] \in R_q$ and $c_2 = [(p/2) \ast m + (p/q) \ast (b \cdot r)] \in R_p$. The ciphertext is the pair $(c_1, c_2)$.

Decryption. Output $[(2/p) \ast (c_1 - c_1 \cdot s)] \in \{0,1\}^n$.

2.6. Kyber. Kyber [16] is a recent module-LWE [31, 32] based CPA- (Chosen Plaintext Attack-) secure encryption scheme and can be applied to build CCA- (Chosen Ciphertext Attack-) secure key encapsulation mechanism (KEM). In this paper, we focus on the former, implementing the Kyber’s public-key encryption scheme. For positive integers $d_b, e_1, e_2, e_3, k, n, \eta$ and modulus $q \in \mathbb{Z}$, Kyber needs to generate matrices with small dimension, and each matrix contains several polynomials with coefficients in $R_q$ as its elements. The compression and decompression functions of Kyber are defined as follows:

$$Compress_q(x, d) = \left(\frac{2^d}{q}\right) \ast x \mod 2^d;$$

$$Decompress_q(x, d) = \left(\frac{q}{2^d}\right) \ast x.$$

In key generation, a binary string $\alpha$ is chosen uniformly at random from $\{0,1\}^n$. The matrix $A \in (R_q)^{k \times k}$ can be pregenerated by method SHAKE-128($\alpha$), and two vectors $s, e$ are sampled from $(B_q)^k$. We compute $= Compress_q(As + e, d)$, where the secret key is $s$ and the public key is the pair $(A, b)$. In encryption, we generate vectors $r, e_1 \leftarrow (B_q)^k$ and $e_2 \leftarrow B_q$. Then we obtain the vector $b_1$ from $b$ by method $Decompress_q(b, d)$. Given a message $m \in \mathbb{Z}_q^n$, the ciphertext is the pair $(c_1, c_2)$, where $c_1 = Compress_q(A^T r + e_1, d)$ and $c_2 = Compress_q(b_1 r + e_2 + [q/2] \ast m, d)$. In decryption, we compute $u = Decompress_q(c_1, d)$ and $v = Decompress_q(c_2, d)$ and then output the result $Compress_q(v - s^T u, 1)$.

2.7. Frodo. Frodo [17], the key-exchange protocol based on the LWE problem, has parameters $b, d, l, m, n, q \in \mathbb{Z}$ and a real number $\sigma > 0$. The matrix $A \in \mathbb{Z}_q^{2m \times n}$ is generated from $seed_A$ via a pseudorandom generating function $Gen()$.

In this paper, we focus on the main computation process in Figure 1: we skip the generating function $Gen()$ and precompute the matrix $A$. Let $b' = (\log_2(q) - b$, for a matrix $M \in \mathbb{Z}_q^{2x \times y}$, the rounding function $[M]_{2^b}$ and the cross - rounding function $\langle M \rangle_{2^b}$ are defined as follows, respectively:

$$rounding : [M]_{2^b} = \left[2^b \ast M \right] \mod 2^b;$$

$$cross - rounding : \langle M \rangle_{2^b} = \left[2^b - 2^{b-1} \ast M\right] \mod 2.$$
because web browsers are one of the essential platforms for NIST PQC standardization project. In this paper, we choose Mozilla Firefox 57.0.2 as our benchmark platform and propose an open-source project Alea (available URL: https://github.com/nquinlan/better-random-numbers-for-javascript-mirror) to be our secure pseudorandom number generator (PRNG). For comparison, we execute same programs on Google Chrome 63.0.3239.108, Opera 53.0.2907.68, and Microsoft Edge 42.17134.1.0. What we want to see is the performance difference between those web browsers. We will show the running time of several lattice-based cryptosystems on web browsers in Section 5.

3.2. Tessel2. Similar to its old model, Tessel2 is a JavaScript-enabled embedded system with on-board WiFi capabilities designed for IoT developers. Tessel2 features a 380 MHz Mediatek MT7620n router-on-a-chip + 48MHz Atmel SAMD21 coprocessor, running Linux built on OpenWRT with 64MB of DDR2 RAM, and 32MB of Flash memory. Tessel2 is compatible with Node.js and runs JavaScript programs directly for controlling a wide variety of IoT modules; it allows developers to easily control modules via a pair of multipurpose ports. Tessel2 is also programmable in other programming languages; however, a part of browser-side JavaScript libraries or objects is not supported.

3.3. Android WebView, PC, and Mac. Android has a built-in browser-like activity which is called WebView. It can be used to display web pages or HTML files as a part of UI. Developers can build a WebView activity to show online content or user data within applications. Android 4.4 has replaced the rendering engine of WebView with Chromium’s V8 engine to deliver improved JavaScript performance. We chose WebView in Android 4.4 (KitKat) to benchmark our JavaScript implementation and ran our implementation on a test PC and a MacBook Pro, respectively. We tested our implementation on the four JavaScript run-time environments above. We ran the code on WSH and Node.js for PC and ran on osascript and Pacifista for Mac (see Appendix for the commands).

4. Efficient Algorithms for JavaScript Implementation

4.1. Discrete Gaussian Sampling. Let $l \in \mathbb{Z}$ be the precision of binary expansion of the probabilities and $n \in \mathbb{Z}$; there are $n$ binary probabilities $p_0, p_1, \ldots, p_{n-1} \in \mathbb{Z}_2$. A probability matrix $P_{\text{mat}} = [p_0, p_1, \ldots, p_{n-1}] \in \mathbb{Z}_2^{n \times n}$ is composed of all the computed probabilities, and each column stores one probability. Let $k_0, k_1, \ldots, k_{n-1} \in \mathbb{Z}_2^n$ be all the rows of $P_{\text{mat}}$; hence, $P_{\text{mat}}$ can be stored as a one-dimensional array $k = (k_0, k_1, \ldots, k_{n-1}) \in \mathbb{Z}_2^n$ for Algorithm 1.

With limited computing capacity, the computation of probabilities would become a time-consuming operation for some programming languages or platforms. In general, discrete Gaussian sampling requires a high-precision floating-point operation or large storage requirement [35] to ensure the security level. Inspired by the idea of implementing Knuth-Yao algorithm in FPGAs [27], we modify and implement the algorithm in JavaScript. Moreover, discrete Ziggyurt algorithm [30] which allows for a time-memory trade-off has been changed to be portable in chosen platforms. In this case, Knuth-Yao algorithm shows better performance than modified discrete Ziggyurt algorithm. In fact, with different features, the performance of those two sampling algorithms varies on different platforms. Therefore, we choose Knuth-Yao algorithm to speed up discrete Gaussian sampling.

4.2. Number Theoretic Transform. NTT is an efficient approach of generalization of fast Fourier transforms (FFT) doing a transform over the finite field $\mathbb{Z}_q$ ($q > 0$) instead of the complex number field $\mathbb{C}$. It has lower asymptotic complexity $O(n \log n)$ for multiplying polynomials with higher degrees.

For $n$ being a power of 2 and $q$ a prime number with $q \equiv 1 \pmod{2n}$, NTT accepts a polynomial $a \in R_q$ whose coefficients are in the standard order as input, and outputs another polynomial $a' = \text{NTT}(a)$, $a'$ can be defined as $a'_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \pmod{q}$ ($i = 0, 1, \ldots, n - 1$), where $\omega$ is a $n$-th primitive root of unity in $\mathbb{Z}_q$. Similarly, we denote the inverse NTT as $\text{NTT}^{-1}$ that $a = \text{NTT}^{-1}(a')$, where $a_i = n^{-1} \sum_{j=0}^{n-1} a'_j \omega^{-ij} \pmod{q}$ ($i = 0, 1, \ldots, n - 1$), such that the output of $\text{NTT}^{-1}$ satisfies $\text{NTT}^{-1}(\text{NTT}(a)) = a$.

We have implemented iterative forward NTT [11, 36] algorithm in our previous works [12, 13]. Both Kyber and NewHope are required to perform polynomial multiplication, and some literature such as [23, 24] provided efficient polynomial multiplication methods to combine bit reversal
Input: $l, n \in \mathbb{Z}$, a probability array $k = (k_0, k_1, \ldots, k_{l-1}) \in \mathbb{Z}_2^{ln}$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$
1 Let $d = 0$, $x = 0$, $sign = 0$
2 while true do
3 $r \leftarrow \{0, 1\}$ uniformly at random;
4 $d = 2d + r$;
5 for $i = n$ down to 0 by 1 do
6 $d = d - k_i$;
7 if $d = -1$ then
8 if $i = 0$ then $sign \leftarrow \{0, 1\}$ uniformly at random;
9 else $sign \leftarrow \{-1, 1\}$ uniformly at random;
10 return $s = sign \cdot row$;
11 endif
12 if $sign = 1$ then return $s = i$;
13 else $d = 0$;
14 $r \leftarrow \{0, 1\}$ uniformly at random;
15 $d = 2d + r$;
16 $x = 0$;
17 continue
18 endif
19 endfor
20 $x += 1$;
21 endwhile
Algorithm 1: Knuth-Yao algorithm.

Input: Polynomial $a \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$, and a LUT $\Psi_{rev} \in \mathbb{Z}_q^n$ in bit-reversed order
Output: Polynomial $a' = \text{NTT}(a) \in R_q$
1 $t = n$;
2 for $m = 1$ to $n - 1$ by $m = 2m$ do
3 $t = t/2$;
4 for $i = 0$ to $m - 1$ do
5 $j_1 = 2 \cdot i \cdot t$;
6 $j_2 = j_1 + t - 1$;
7 $S = \Psi_{rev}[m + i]$;
8 for $j = j_1$ to $j_2$ do
9 $U = a_j$;
10 $V = a_{j+1} \cdot S$;
11 $a_j = U + V \mod q$;
12 $a_{j+1} = U - V \mod q$;
13 endfor
14 endfor
15 endfor
16 return $a$.
Algorithm 2: Cooley-Tukey (CT) forward number theoretic transform (NTT).

with NTT computation; hence, in this paper, we follow the state-of-the-art and implement optimized \text{NTT}/\text{NTT}^{-1} as shown in Algorithms 2 and 3.

Let $\psi \in \mathbb{Z}_q$ be a primitive $2n$-th root of unity such that $\omega = \psi^2$. We write two polynomials $f = (f_0, f_1, \ldots, f_{n-1})$ and $\overline{f} = (f_0, \psi f_1, \ldots, \psi^{n-1} f_{n-1}) \in R_q$. To compute the polynomial multiplication $c = a \cdot b \in R_q$, first we precompute all $2n$ powers of $\psi$ and $\psi^{-1}$ and then store $n$ powers of $\psi$ and $\psi^{-1}$ with bit-reversed order in look-up tables $\Psi_{rev}, \Psi_{rev}^{-1} \in \mathbb{Z}_q^n$, respectively. So the bit-reverse operation for input polynomial can be merged into precomputation. Then we obtain the negative wrapped convolution $c =$...
Input: Polynomial $a' \in R_q = Z_q[x]/(x^n + 1)$, and a LUT $\Psi^{-1}_{rev} \in Z^n_q$ in bit-reversed order

Output: Polynomial $a = NTT^{-1}(a') \in R_q$

1. $t = 1$
2. for $m = n$ to 2 by $m = m/2$ do
3. $h = m/2$, $j_1 = 0$
4. for $i = 0$ to $h - 1$ do
5. $j_2 = j_1 + t - 1$
6. $S = \Psi^{-1}_{rev}[h + i]$
7. for $j = j_1$ to $j_2$ do
8. $U = a_j$
9. $V = a_j + t$
10. $a_{j+t} = (U - V) \cdot S \mod q$
11. $a_j = U + V \mod q$
12. endfor
13. $j_1 = j_1 + 2t$
14. endfor
15. $t = 2t$
16. endfor
17. for $i = 0$ to $n - 1$ do
18. $a_i = a_i \cdot n^{-1} \mod q$
19. endfor
20. return $a$

Algorithm 3: Gentleman-Sande (GS) inverse number theoretic transform ($NTT^{-1}$).

(1, $\psi^{-1}, \ldots, \psi^{-(n-1)}$) $\circ NTT^{-1}(NTT(a) \circ NTT(b))$, where $\circ$ denotes the point-wise multiplication.

5. Performance on Web Browsers

We implemented three encryption schemes: Lizard, ring-Lizard [15], Kyber [16], and two key exchange protocols: Frodo [17] and NewHope [18] in JavaScript. Again, it should be noted that we mainly focus on the computation process and discrete Gaussian sampling in this paper. Hence, we omitted some steps about the generation, encoding/decoding functions for uniformly chosen public key component or binary seeds. We will go into detail of our implementation performance in this section. The simple usage of our implementation is described in the Appendix.

For comparison, we implemented those five lattice-based cryptosystems corresponding to about 128-bit postquantum security level (see Table 1). Figure 3 shows the performance results of our implementation executed on the Firefox browser. As we expected, the ring-LWE based cryptosystems including Kyber and NewHope are apparently very efficient. The key size of Kyber is smaller than that of Lizard, although Kyber has large moduli. Key generation of Kyber runs over 400 times faster than that of Lizard, but decryption of Lizard is the fastest. Key generation and encryption of ring-Lizard are over 60 and 4 times faster than that of Lizard; however, Kyber is still much more efficient than ring-Lizard. Compared with Frodo, both Alice's and Bob's sides of NewHope run over 8 times and 13 times faster, respectively.
For Lizard, we stored the matrices in two-dimensional arrays, to reduce the running time of matrix multiplication due to the row-major order matrix convention in JavaScript. Specifically, we computed the product of a vector with a matrix transpose instead of calculating the matrix-vector product. In addition, the elements of $s$ in key generation and $r$ in encryption only contain the values from the set $\{0, \pm 1\}$; hence, we could replace integer multiplication with addition and subtraction if multiplicand equals $\pm 1$. For ring-Lizard, we computed polynomial multiplication by using Karatsuba algorithm because the moduli of ring-Lizard are powers of 2.

For Kyber, we skipped the generation of binary seeds and polynomials. In key generation and encryption, we precomputed $A$ and $\text{NTT}(\langle q/2 \rangle \ast m)$ and sampled the error vectors from a binomial distribution $B_q$. Each element of matrices and vectors in Kyber is a polynomial over $R_q$ with degree equal to $n - 1$ ($q \equiv 1 \mod 2n$); hence, NTT can be applied to Kyber to effectively compute polynomial multiplication. Let $i, j, k$ be positive integers; we assume a matrix $A = (a_{ij}) \in (R_q)^{k \times k}$ is in NTT domain, and the coefficients of each element $a_{ij}$ are in bit-reversed order. In key generation, we performed NTT on error vectors such that the component of public key $b = \text{NTT}^{-1}(\langle NTT(s) + NTT(e) \rangle)$, only 6 calls of NTT and 3 calls of $\text{NTT}^{-1}$ are necessary if $k = 3$. Similarly, we computed $\text{NTT}^{-1}(\langle NTT(A^2) \rangle NTT(r) + NTT(e_1))$ and $\text{NTT}^{-1}(\langle NTT(b^2) NTT(r) + NTT(e_2) \rangle + NTT(\langle q/2 \rangle \ast m))$ by invoking NTT 10 times and $\text{NTT}^{-1}$ 4 times in encryption and outputted $\text{NTT}^{-1}(\langle NTT(s)^2 \rangle u)$ by invoking NTT 4 times and $\text{NTT}^{-1}$ 1 time in decryption.

For Frodo, we skipped the generation of the seed $a$ from a binary string and precomputed the matrix $A$ on both Alice's and Bob's sides. There is no problem to perform floating-point arithmetic on the JavaScript-enabled platforms, but we replace floating point arithmetic to integer arithmetic in the rounding/cross-rounding and reconciliation functions considering our follow-up development on memory-constrained devices. To sample the error matrices, we performed our modified Knuth-Yao algorithm as shown in Algorithm 1.

For NewHope, we also performed $\text{NTT}/\text{NTT}^{-1}$ to speed up the polynomial multiplication which is a bottleneck for ring-LWE based cryptography in JavaScript (e.g., see [13, 14]). In this case, we implemented NewHope following [18] (Section 7.1, Protocol 3) but skipped SHAKE-128 method, hash function SHA3-256, and key encoding/decoding functions. We precomputed the polynomial $a$ on both Alice's and Bob's sides and sent polynomials $b, u, r$ directly. Comparing our implementation with the approach in [24], we only computed $\text{NTT}^{-1}(b \ast \text{NTT}(s)) + e_2$ on Bob's side so that the computation of $\text{NTT}(e_3)$ has been omitted.

Figure 4 shows the decomposition of computation time of our implementation. Although each implementation technique and performance is different, polynomial and matrix multiplication are still the most time-consuming computation. In Lizard and Frodo, matrix multiplication accounts for at least 70%. In Kyber and NewHope, more than 50% of the running time is spent in NTT/$\text{NTT}^{-1}$. Except for ring-Lizard, the error elements generation including discrete Gaussian sampling and binomial sampling costs little running time in the calculations, accounting about 20% for Frodo and about 10% in Lizard, Kyber, and NewHope; discrete Gaussian sampling accounts for about 50% in key-generation of ring-Lizard.

We executed the same JavaScript programs on other desktop PC browsers including Google Chrome, Opera, and
Edge. Taking Kyber and NewHope as examples, Figure 5 shows the running time on those web browsers. It appears that the performance of our implementation executed on both Chrome and Opera is quite similar, and Firefox delivers the better performance than Edge.

6. Performance on Other JavaScript-Enabled Platforms

In this section, we present the implementation performance comparison on IoT device Tessel2, Android phone, Windows, and macOS. Our implementation is designed to be portable and can be executed on those experimental platforms directly without modification. In this case, we precomputed random values generation and discrete Gaussian sampling because of the difficulty of implementing cryptographic secure PRNG in JavaScript on microcontrollers such as Tessel2 (see [13]).

6.1. Tessel2. Figure 6 shows the performance of our implementation executed on Tessel2 (for Lizard, the sizes of keys are too large to be generated on Tessel2). Note that the running time is measured in seconds. We have implemented the ring-LWE based encryption scheme [5] on the old model of Tessel (see [13]). As in our previous work, the performance results achieved on Tessel2 are several orders of magnitude slower than that on web browsers. However, Tessel2 has upgraded hardware specification with better computing capacity. For example, encryption and decryption of Kyber are over 1000 times slower than that of running on Firefox. But the performance of Kyber and NewHope is still unexpectedly high, and the calculation process can be completed within 1 or 2 seconds. Even though the computation of Kyber/NewHope is more complicated than [5], noticeable effects can be achieved in hardware performance and memory costs with our improved implementation.

6.2. Android Phone. WebView is an extension of Android’s View class to display web pages and applications. It provides different performance from other web browsers on Android framework. We ran our implementation on Android phone Xperia XZ au SOV34, which is equipped with Qualcomm Snapdragon 820 MSM8996/2.2GHz DualCore + 1.6GHz DualCore.
Table 2: Performance results on Android phone.

<table>
<thead>
<tr>
<th>Key Generation</th>
<th>Average running time (ms)</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>1575.91</td>
<td>38.62</td>
<td>9.63</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>13.24</td>
<td>15.05</td>
<td>5.61</td>
</tr>
<tr>
<td>Kyber</td>
<td>3.57</td>
<td>5.78</td>
<td>2.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alice0</th>
<th>Bob</th>
<th>Alice1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frodo</td>
<td>38.10</td>
<td>88.03</td>
</tr>
<tr>
<td>NewHope</td>
<td>5.14</td>
<td>10.08</td>
</tr>
</tbody>
</table>

Table 3: Performance results on WSH.

<table>
<thead>
<tr>
<th>Key Generation</th>
<th>Average running time (ms)</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>27021.1</td>
<td>372.2</td>
<td>72.3</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>145.44</td>
<td>283.19</td>
<td>140.37</td>
</tr>
<tr>
<td>Kyber</td>
<td>9.89</td>
<td>15.76</td>
<td>6.15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alice0</th>
<th>Bob</th>
<th>Alice1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frodo</td>
<td>827.03</td>
<td>1102.92</td>
</tr>
<tr>
<td>NewHope</td>
<td>8.85</td>
<td>17.51</td>
</tr>
</tbody>
</table>

Table 4: Performance results on Node.js.

<table>
<thead>
<tr>
<th>Key Generation</th>
<th>Average running time (ms)</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>271.08</td>
<td>6.54</td>
<td>1.47</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>1.86</td>
<td>2.48</td>
<td>1.22</td>
</tr>
<tr>
<td>Kyber</td>
<td>0.44</td>
<td>0.67</td>
<td>0.27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alice0</th>
<th>Bob</th>
<th>Alice1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frodo</td>
<td>7.04</td>
<td>14.73</td>
</tr>
<tr>
<td>NewHope</td>
<td>0.20</td>
<td>0.64</td>
</tr>
</tbody>
</table>

DualCore and 3GB RAM. We created an HTML file that includes our JavaScript code and loaded it as a local file into WebView.

Table 2 shows the running time of our implementation on Android phone. From the performance results, it is clear that the performance of ring-LWE based cryptosystems is also acceptable. For encryption schemes, Kyber runs about 3 times faster than ring-Lizard, as well as over 4 times faster than Lizard. For key-exchange protocols, Frodo runs about 10 times slower than NewHope; matrix multiplication accounts for about 80% in Alice’s side and 90% in Bob’s side; the ratio is higher than that of on Firefox. Overall, the running speed achieved on Xperia XZ au SOV34 is at least 5 times slower than that on Firefox.

6.3. Other JavaScript Run-Time Environments on Windows and macOS. For comparison, we investigated the performance of our JavaScript implementation on PC and Mac. It is not difficult to execute our code on other JavaScript run-time environments directly since our implementation has excellent portability. Those environments rely on specific platforms or OS for scripting. For example, JavaScript files (.js type) can be run in GUI mode via WScript.exe and Windows Command Prompt by calling CScript.exe; running Pacifista requires the installation of Java Runtime Environment (JRE). As of now, the performance of postquantum cryptography in JavaScript on those platforms has rarely been studied. To the best of our knowledge, this work is the first. In this case, we used WSH and Node.js on Windows 10 Home and used osascript and Pacifista on macOS High Sierra.

From Tables 3, 4, 5, and 6, we can see that there is a huge performance gap in running the JavaScript code on WSH with other platforms. The running speed of WSH is the slowest; e.g., key generation of Lizard on WSH is about 100 times slower than that on osascript, encryption is over 250 times slower, and decryption is over 150 times slower. Running NewHope on WSH is about 15 times slower than that on Firefox (without consideration of the cost of random values generation).

Node.js delivers almost the best performance for ring-LWE based cryptosystems. For example, Kyber runs about
Table 5: Performance results on osascript.

<table>
<thead>
<tr>
<th></th>
<th>Key Generation</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>209.99</td>
<td>1.43</td>
<td>0.41</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>3.10</td>
<td>4.66</td>
<td>2.44</td>
</tr>
<tr>
<td>Kyber</td>
<td>0.80</td>
<td>1.20</td>
<td>0.49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Average running time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice0</td>
<td>Bob</td>
</tr>
<tr>
<td>Frodo</td>
<td>6.88</td>
</tr>
<tr>
<td>NewHope</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 6: Performance results on Pacifista.

<table>
<thead>
<tr>
<th></th>
<th>Key Generation</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizard</td>
<td>1301.54</td>
<td>24.67</td>
<td>5.76</td>
</tr>
<tr>
<td>Ring-Lizard</td>
<td>43.77</td>
<td>34.24</td>
<td>17.97</td>
</tr>
<tr>
<td>Kyber</td>
<td>7.39</td>
<td>4.49</td>
<td>1.78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Average running time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice0</td>
<td>Bob</td>
</tr>
<tr>
<td>Frodo</td>
<td>35.80</td>
</tr>
<tr>
<td>NewHope</td>
<td>2.57</td>
</tr>
</tbody>
</table>

2 times faster than that on osascript, and ring-Lizard runs over 10 times faster than that on Pacifista. The performance of Node.js is almost the same as on Google Chrome, which also uses Google's V8 JavaScript engine.

Osascript is also an effective platform for macOS; e.g., running Frodo on osascript is slightly faster than that on Firefox; encryption of ring-Lizard is about 3 times and 5000 times faster than that on Xperia XZ au SOV34 and Tessel2, respectively.

The running speed of Pacifista is less than Node.js and osascript, but still higher than WSH and can be comparable to Android WebView; hence, its performance is acceptable to the developers. The exception for all three encryption schemes is that the running time of key generation is longer than that of encryption.

7. Conclusions

We first implemented five new lattice-based encryption schemes (Lizard, ring-Lizard, Kyber) and key exchange protocols (Frodo, NewHope) in JavaScript and tested their performances on web browsers, Tessel2, Android phone, and other platforms on PC and Mac. Our code can be executed on any JavaScript-enabled platforms since it has good portability. We used NTT to improve the speed of polynomial multiplication and modified Knuth-Yao algorithm for discrete Gaussian sampling. We reported the performance results of our implementation on multiple JavaScript-enabled platforms; by contrast, the ring-LWE based cryptosystems show better performance than others. Our proof-of-concept implementation demonstrates that some of the lattice-based cryptosystems can be implemented efficiently in JavaScript. Hence, our work could be a good reference for lattice-based cryptography in the standardization process of NIST. In our future work, we expect to improve the implementation for particular platforms and investigate more lattice-based public-key encryption schemes and KEM on more platforms for the NIST PQC standardization project.

Appendix

Simple Usage of Our Implementation

We take Lizard as an example for explaining how to use our source code.

Execution

Web Browsers. To run Lizard on web browsers, we create an HTML file which containing necessary contents as in Pseudocode 1.

prng.js is our main number generator which includes a fast PRNG algorithm. If Lizard is executed on Opera, we can also use the standard function of ECMAScript Math.random() which is implemented securely (See https://lists.w3.org/Archives/Public/public-webcrypto/2013Jan/0063.html). lizard_random_values.js contains the pregenerated random numbers for testing. The main function of Lizard is testlizard() in lizard.js (see Pseudocode 2).

Android Phone. We create the Android application package (APK) file using Eclipse Kepler Service Release 2 and Android Development Toolkit (ADT, Version: 23.0.7.2120684). We copy the necessary code from those .js files and paste it into an HTML file for use in our project. This HTML file is placed within the assets folder as a local file. Then we
modify the onCreate() function in MainActivity.java (see Pseudocode 3). We can export the created .apk file from the bin folder and install it on the Android phone.

Other Platforms. We copy the necessary code and paste it into a .js file. The program can be executed in a command shell; for example, as follows.

Tessel2. It needs to import the interface to Tessel hardware at the top of the .js file:

```javascript
var tessel = require('tessel');
```

In the command line, enter

C:\tesel2-code\t2 run new_lizard.js
to run Lizard in Tessel2's RAM.

WSH

C:\new_folder\Lizard>cscript new_lizard.js
C:\new_folder\Lizard>WScript new_lizard.js

Node.js

C:\new_folder\Lizard>node new_lizard.js

Data Availability

The relevant test data used to support the findings of this study are included in the article.
import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;
public class MainActivity extends Activity {
    private WebView webview;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        webview = new WebView(this);
        webview.setSettings()
            .setJavaScriptEnabled(true);
        webview.loadUrl
            ("file:///android_asset/lizard.html");
        setContentView(webview);
    }
    // ...
}

Pseudocode 3

Test Lizard:
Input:
m = 960
n = 608
l = 256
t = 2
p = 256
q = 1024
// ...
Output:
plaintext = 0,0,1,0,1,0,0,1,1,1,0,0,1,0,0,1,0,1,1,1,1, //...
result = 0,0,1,0,1,0,1,1,1,0,0,1,0,0,1,0,1,1,1,1, //...
Success!

Pseudocode 4

Disclosure

A preliminary version of this paper was presented at the 2018 Symposium on Cryptography and Information Security (SCIS2018) held in Niigata, Japan, on January 25, 2018 [14].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References


