Hindawi
Security and Communication Networks
Volume 2018, Article ID 9846168, 14 pages
https://doi.org/10.1155/2018/9846168

Research Article

Portable Implementation of Postquantum Encryption Schemes and Key Exchange Protocols on JavaScript-Enabled Platforms

Ye Yuan ,[1] Junting Xiao,[1] Kazuhide Fukushima,[2] Shinsaku Kiyomoto,[2] and Tsuyoshi Takagi[3,4]

[1] *Graduate School of Mathematics, Kyushu University, Japan*
[2] *KDDI Research, Inc., Japan*
[3] *Department of Mathematical Informatics, The University of Tokyo, Japan*
[4] *CREST, Japan Science and Technology Agency, Japan*

Correspondence should be addressed to Ye Yuan; y-yuan@math.kyushu-u.ac.jp

Received 6 April 2018; Revised 27 June 2018; Accepted 18 July 2018; Published 13 September 2018

Academic Editor: Mun-Kyu Lee

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

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

Lizard	$\boldsymbol{m}$	$\boldsymbol{n}$	$\boldsymbol{l = p}$	$\boldsymbol{q}$	$\boldsymbol{t}$	$\boldsymbol{\alpha}^{-1}$
	960	608	256	1024	2	182
Ring-Lizard	$\boldsymbol{n = q}$	$\boldsymbol{p}$	$\boldsymbol{\alpha}^{-1}$	– –	– –	– –
	1024	256	154	– –	– –	– –
Kyber	$\boldsymbol{k}$	$\boldsymbol{n}$	$\boldsymbol{q}$	$\boldsymbol{\eta}$	$\boldsymbol{d_b = d_{c_1}}$	$\boldsymbol{d_{c_2}}$
	3	256	7681	4	11	3
Frodo	$\boldsymbol{b}$	$\boldsymbol{l}$	$\boldsymbol{m}$	$\boldsymbol{n}$	$\boldsymbol{q}$	$\boldsymbol{\sigma}$
	4	8	8	752	32768	1.3229
NewHope	$\boldsymbol{k}$	$\boldsymbol{n}$	$\boldsymbol{q}$	– –	– –	– –
	16	1024	12289	– –	– –	– –

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 implementationcan 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 (NIST Postquantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography).

The primary reason we chose those five cryptosystems is that they cover a majority of the variants of the LWE-based cryptography, which we will introduce in Section 2. In addition, the parameters and key sizes of those cryptosystems above are not too large and suitable for implementing on web browsers and small devices using JavaScript.

Contributions of this paper can be summarized as follows:

(i) We first implement the five lattice-based encryption schemes and key exchange protocols mentioned above using JavaScript. We observe running time of our implementation and find that the performance of the ring-LWE based cryptosystems is much higher than that of others. For example, on the web browsers, the entire calculation process of Kyber and NewHope can be accomplished within milliseconds; even the IoT device Tessel2 is fast enough to perform all of those operations in merely two seconds. Our implementation will be improved further for the NIST PQC standardization project in a future work.

(ii) We refactor our implementation to accelerate polynomial operations for ring-LWE based cryptographic algorithms. 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.

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 $\boldsymbol{a}$, while vectors are denoted by bold small letters such as $\mathbf{v}$ and matrices and bold large letters such as $\mathbf{A}$. For an integer $m \in \mathbb{N}$, we define the modulo operation $b \equiv a \bmod m$ in the range $[0, m) \cap \mathbb{Z}$.

2.2. LWE, Ring-LWE, and LWR Problems. Regev proposed the original LWE problem [3] using integer matrix in 2005. Let m, n, q be positive integers; the search LWE problem is required to find a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ by inputting a pair of matrices $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^m$. The decision LWE problem is to distinguish b between a uniformly distributed random vector from $\mathbb{Z}_q^m$ and a noisy inner product $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$. Usually, the elements of $\mathbf{A}$ are randomly selected from $\mathbb{Z}_q$, and the so-called error vector $\mathbf{e} \in \mathbb{Z}^m$ is sampled from a target probability distribution χ. The cryptography based on the LWE problem uses an unusual structure lattice which is called q-ary lattice:

$$L_q^{\perp}(\mathbf{A}) = \{\mathbf{v} \in \mathbb{Z}^n \mid \mathbf{A}\mathbf{v} \equiv 0 \bmod q\};$$
$$L_q(\mathbf{A}) = \left\{\mathbf{v} \in \mathbb{Z}^n, \mathbf{s} \in \mathbb{Z}^m \mid \mathbf{v} \equiv \mathbf{A}^T\mathbf{s} \bmod q\right\}; \tag{1}$$

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_q is an ideal lattice if each polynomial over R_q has a bijective mapping to an ideal $\mathbb{Z}_q^n$. Given polynomials $\boldsymbol{a}, \boldsymbol{b} \in R_q$, the search version of the ring-LWE problem is to recover the secret $\boldsymbol{s} \in R_q$, where $\boldsymbol{a}$ is chosen uniformly and $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$ with an "error" $\boldsymbol{e} \in R$ sampled from a target probability distribution χ. The decision ring-LWE problem is similar to the decision LWE problem: given $\boldsymbol{a}, \boldsymbol{b} \in R_q$, we distinguish whether $\boldsymbol{b}$ is also chosen uniformly, or there exists a polynomial $\boldsymbol{s} \in R_q$ such that $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$. 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 (α-SVP) on ideal lattices.

Given a matrix $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\} \in \mathbb{Z}_q^{m\times n}$ and an inner product with rounding $\mathbf{b} = \lfloor \mathbf{A}\mathbf{s} \rfloor_p \in \mathbb{Z}_p^m$, the LWR problem (see [25]) is to find the vector $\mathbf{s} \in \mathbb{Z}_q^n$, where $p \ll q$. The information hiding technique or so-called derandomization technique of LWR is different from LWE: each value of the inner product $\mathbf{b}$ times a rounded value $\lfloor q/p \rfloor$ over $\mathbb{Z}_p$, 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 $\rho_\sigma = \exp(-\pi\|\mathbf{x}\|/s^2)$, where the Gaussian parameter $s = \sigma\sqrt{2\pi}$. A discrete version of Gaussian distribution over $\mathbb{Z}$ is defined by $D_\sigma(x) = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$. 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 $(a_0, a_1, \ldots, a_{k-1}, b_0, b_1, \ldots, b_{k-1}) \longleftarrow \{0, 1\}^{2k}$

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 $t \mid p \mid q$. For a real number $0 < \rho < 1$, we sample values $(v_1, v_2, \ldots, v_n) \longleftarrow \{-1, 0, 1\}^n$ from the distribution $\mathcal{ZO}_n(\rho)$ 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 $0 \le h \le m$, we sample the values $(v_1, v_2, \ldots, v_n) \longleftarrow \{-1, 0, 1\}^m$ from the distribution $\mathcal{HWT}_m(h)$ such that it has exactly h nonzero entries in those values.

In key generation, we choose a matrix $\mathbf{S} = \{\mathbf{s}_1, \ldots, \mathbf{s}_l\} \in \mathbb{Z}_q^{n\times l}$ by sampling column vectors $\mathbf{s}_i \in \mathbb{Z}_q^n$ $(i = 1, 2, \ldots, l)$ independently from the distribution $\mathcal{ZO}_n(1/2)$. Input a matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ whose elements are chosen uniformly from $\mathbb{Z}_q$; then we can compute the matrix $\mathbf{B} = \mathbf{A}\mathbf{S} + \mathbf{E} \in \mathbb{Z}_q^{m\times l}$, where the error matrix $\mathbf{E} \in \mathbb{Z}_q^{m\times l}$ is chosen according to $D_{\mathbb{Z},\alpha q}$. The secret key is $\mathbf{S}$ and the public key is the pair $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{m\times n} \times \in \mathbb{Z}_q^{m\times l}$. In encryption, by choosing a random vector $\mathbf{r} \in \mathbb{Z}^m$ from the distribution $\mathcal{HWT}_m(128)$, we compute a pair $(\mathbf{c}_1, \mathbf{c}_2) = (\mathbf{A}^T\mathbf{r}, \mathbf{B}^T\mathbf{r}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$. Given a message $\mathbf{m} \in \mathbb{Z}_t^l$, the cipher text is the pair $(\mathbf{c}_1', \mathbf{c}_2')$ where $\mathbf{c}_1' = \lfloor (p/q) * \mathbf{c}_1 \rceil \in \mathbb{Z}_p^n$ and $\mathbf{c}_2' = \lfloor (p/t) * \mathbf{m} + (p/q) * \mathbf{c}_2 \rceil \in \mathbb{Z}_p^l$. Lastly, we output the vector $\mathbf{m}' = \lfloor (t/p) * (\mathbf{c}_2' - \mathbf{S}^T\mathbf{c}_1') \rceil \in \mathbb{Z}_t^l$ 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 $\boldsymbol{e} \longleftarrow D_{\mathbb{Z},\alpha q}$; choose a "small" random polynomial s from $\mathcal{HWT}_n(128)$ and a uniformly random polynomial $\boldsymbol{a} \in R_q$; then output the public key $(\boldsymbol{a}, \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}) \in R_q \times R_q$ and the secret key $\boldsymbol{s} \in R$.

Encryption. Choose a random polynomial $\boldsymbol{r}$ from $\mathscr{HWT}_n(128)$; given a plaintext $\boldsymbol{m} \in \{0,1\}^n$, then compute $\boldsymbol{c}_1 = \lfloor (p/q) * (\boldsymbol{a} \cdot \boldsymbol{r}) \rceil \in R_p$ and $\boldsymbol{c}_2 = \lfloor (p/2) * \boldsymbol{m} + (p/q) * (\boldsymbol{b} \cdot \boldsymbol{r}) \rceil \in R_p$. The ciphertext is the pair $(\boldsymbol{c}_1, \boldsymbol{c}_2)$.

Decryption. Output $\lfloor (2/p) * (\boldsymbol{c}_2 - \boldsymbol{c}_1 \cdot \boldsymbol{s}) \rceil \in \{0,1\}^n$.

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 $d_b, d_{c_1}, d_{c_2}, k, n, \eta$ 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:

$$\begin{aligned} \mathit{Compress}_q(x,d) &= \left\lfloor \left(\frac{2^d}{q}\right) * x \right\rceil \bmod 2^d; \\ \mathit{Decompress}_q(x,d) &= \left\lfloor \left(\frac{q}{2^d}\right) * x \right\rceil. \end{aligned} \tag{2}$$

In key generation, a binary string α is chosen uniformly at random from $\{0,1\}^n$. The matrix $\mathbf{A} \in (R_q)^{k\times k}$ can be pregenerated by method *SHAKE-128*(α), and two vectors $\mathbf{s}, \mathbf{e}$ are sampled from $(B_\eta)^k$. We compute $= \mathit{Compress}_q(\mathbf{A}\mathbf{s} + \mathbf{e}, d_b)$. The secret key is $\mathbf{s}$ and the public key is the pair $(\mathbf{A}, \mathbf{b})$. In encryption, we generate vectors $\mathbf{r}, \mathbf{e}_1 \longleftarrow (B_\eta)^k$ and $\mathbf{e}_2 \longleftarrow B_\eta$. Then we obtain the vector $\mathbf{b}_1$ from $\mathbf{b}$ by method $\mathit{Decompress}_q(\mathbf{b}, d_b)$. Given a message $\mathbf{m} \in \mathbb{Z}_2^n$, the cipher text is the pair $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_1 = \mathit{Compress}_q(\mathbf{A}^T\mathbf{r} + \mathbf{e}_1, d_{c_1})$ and $\mathbf{c}_2 = \mathit{Compress}_q(\mathbf{b}_1^T\mathbf{r} + \mathbf{e}_2 + \lfloor q/2 \rceil * \mathbf{m}, d_{c_2})$. In decryption, we compute $\mathbf{u} = \mathit{Decompress}_q(\mathbf{c}_1, d_{c_1})$ and $\mathbf{v} = \mathit{Decompress}_q(\mathbf{c}_2, d_{c_2})$ and then output the result $\mathit{Compress}_q(\mathbf{v} - \mathbf{s}^T\mathbf{u}, 1)$.

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 $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ is generated from $\mathbf{seed_A}$ 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 $\mathbf{A}$. Let $b' = (\log_2 q) - b$, for a matrix $\mathbf{M} \in \mathbb{Z}_q^{x\times y}$, the *rounding* function $\lfloor \mathbf{M} \rceil_{2^b}$ and the *cross* $-$ *rounding* function $\langle \mathbf{M} \rangle_{2^b}$ are defined as follows, respectively:

$$\begin{aligned} \mathit{rounding}: \lfloor \mathbf{M} \rceil_{2^b} &= \left\lfloor 2^{-b'} * \mathbf{M} \right\rceil \bmod 2^b; \\ \mathit{cross}-\mathit{rounding}: \langle \mathbf{M} \rangle_{2^b} &= \left\lfloor 2^{1-b'} * \mathbf{M} \right\rfloor \bmod 2. \end{aligned} \tag{3}$$

The reconciliation function $\mathit{rec}_{2^b}()$ is defined in [33]. The output v is the closest element to $w \in \mathbb{Z}_q$ such that $\langle v \rangle_{2^b} = 0$ or 1. Alice and Bob can obtain the same shared key $\mathbf{K}$ via this reconciliation mechanism.

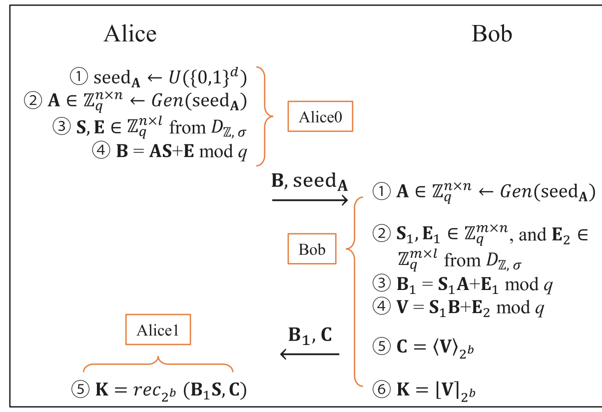

FIGURE 1: Quantum-secure key exchange protocol Frodo.

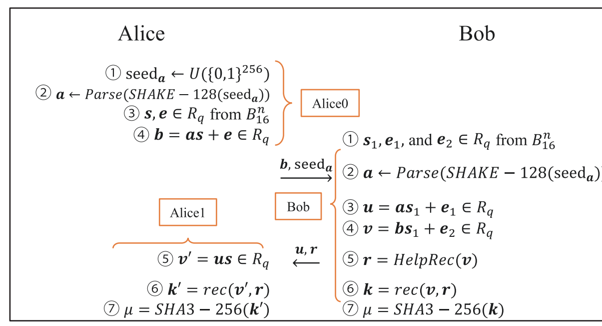

FIGURE 2: Ring-LWE based public-key exchange protocol NewHope.

2.8. NewHope. Compared with another ring-LWE based key exchange protocol BCNS [7], NewHope [18] achieves some improvement in the parameters selecting, errors sampling, and reconciliation mechanism. NewHope is a famous ring-LWE based key exchange protocol due to its experiment which is taking place in Google Canary channel; the result shows that NewHope operates well for Google's postquantum TLS experiment while still being computationally inexpensive (https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html). We would like to know its performance on web browsers or other IoT devices in JavaScript.

Let k, n, and the modular q be positive integers; as with the definition of polynomial ring in Kyber, the keys and errors are all over R_q. A $\mathbf{seed_A}$ is a component of the exchange and generated from a binary string. It is designed to output a polynomial $\mathbf{a} \in R_q$ by *SHAKE-128* method. NewHope also needs to sample random values from a binomial distribution B_k. For reconciliation mechanism, we follow the method in [18] and may use non-floating-point arithmetic [34] in our future work. The overview of NewHope is described in Figure 2.

3. Experimental Runtime Environments

3.1. Web Browsers. Implementing postquantum cryptographic primitives on web browsers is necessary and urgent 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 580 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 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 an Android phone Xperia XZ SOV34 (Android version 8.0.0 (Oreo)).

Some operating systems also provide tools to execute plain text files within a shell/script. The Microsoft Windows Script Host (WSH) is described as an administration tool to provide a scripting environment for batch files. The Active Scripting language engines of WSH can interpret and run script files such as JScript or VBScript. Similarly, Mac users can run JavaScript files by using osascript command, which works with AppleScripts or other Open Scripting Architecture (OSA) language scripts on macOS.

In addition, some nonbrowser software such as Node.js or Pacifista also provide JavaScript runtime environment. Like Google Chrome and Android WebView, Node.js is also built on the Google V8 JavaScript engine and offers a rich variety of JavaScript modules which will be of benefit to development. Pacifista is a simple Java-based open-source project that builds a Linux environment and can upgrade OpenSSL using JavaScript; it can be downloaded at https://github.com/ukiuni/pacifista.

Our implementation is measured on a test PC and a MacBook Pro. The test PC has the following specification: CPU: Intel(R) Core(TM) i5-8250U @1.6GHz; 8GB DDR3 RAM; 256GB SSD; Windows 10 build 17134 Home x64. The MacBook Pro (15-inch, 2017 model) features quad-core Intel(R) Core(TM) i7 @2.8GHz, 16GB 2133MHz LPDDR3 memory, and 512GB SSD with macOS High Sierra 10.13.5. Node.js 8.11.2 and Pacifista 0.0.30 are installed on the test PC and 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 $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-1} \in \mathbb{Z}_2^l$. A probability matrix $\mathbf{P}_{mat} = [\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-1}] \in \mathbb{Z}_2^{l\times n}$ is composed of all the computed probabilities, and each column stores one probability. Let $\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_{l-1} \in \mathbb{Z}_2^n$ be all the rows of $\mathbf{P}_{mat}$; hence, $\mathbf{P}_{mat}$ can be stored as a one-dimensional array $\mathbf{k} = (\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_{l-1}) \in \mathbb{Z}_2^{ln}$ 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 Ziggurat 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 Ziggurat 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 $q \equiv 1 \bmod 2n$, NTT accepts a polynomial $\boldsymbol{a} \in R_q$, whose coefficients are in the standard order as input, and outputs another polynomial $\boldsymbol{a}' = NTT(\boldsymbol{a})$. $\boldsymbol{a}'$ can be defined as $a_i' = \sum_{j=0}^{n-1} a_j\omega^{ij} \bmod q$ $(i = 0, 1, \ldots, n-1)$, where ω is a n-th primitive root of unity in $\mathbb{Z}_q$. Similarly, we denote the inverse NTT as NTT^{-1} that $\boldsymbol{a} = NTT^{-1}(\boldsymbol{a}')$, where $a_i = n^{-1}\sum_{j=0}^{n-1} a_j'\omega^{-ij} \bmod q$ $(i = 0, 1, \ldots, n-1)$, such that the output of NTT^{-1} satisfies $NTT^{-1}(NTT(\boldsymbol{a})) = \boldsymbol{a}$.

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
$\mathbf{k} = (\mathbf{k}_0, \mathbf{k}_1, \ldots, \mathbf{k}_{l-1}) \in \mathbb{Z}_2^{ln}$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$
Let $d = 0$, $x = 0$, $sign = 0$;
while true **do**
  $r \longleftarrow \{0, 1\}$ uniformly at random;
  $d = 2d + r$;
  for $i = n$ down to 0 by 1 **do**
   $d = d - \mathbf{k}_i$;
   if $d = -1$ **then**
    if $i = 0$ **then** $sign \longleftarrow \{0, 1\}$ uniformly at random;
    else
     $sign \longleftarrow \{-1, 1\}$ uniformly at random;
      return $s = sign * row$;
    endif
    if $sign = 1$ **then return** $s = i$;
    else
     $d = 0$;
     $r \longleftarrow \{0, 1\}$ uniformly at random;
     $d = 2d + r$;
     $x = 0$;
     continue
    endif
   endif
  endfor
  $x+ = 1$;
endwhile

Algorithm 1: Knuth-Yao algorithm.

Input: Polynomial $\boldsymbol{a} \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$, and a LUT $\Psi_{rev} \in \mathbb{Z}_q^n$ in bit-reversed order
Output: Polynomial $\boldsymbol{a}' = NTT(\boldsymbol{a}) \in R_q$
$t = n$;
for $m = 1$ **to** $n - 1$ by $m = 2m$ **do**
  $t = t/2$;
  for $i = 0$ **to** $m - 1$ **do**
   $j_1 = 2 * i * t$;
   $j_2 = j_1 + t - 1$;
   $S = \Psi_{rev}[m + i]$;
   for $j = j_1$ **to** j_2 **do**
    $U = a_j$;
    $V = a_{j+t} * S$;
    $a_j = U + V \bmod q$;
    $a_{j+t} = U - V \bmod q$;
   endfor
  endfor
endfor
return $\boldsymbol{a}$.

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 NTT/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 $\boldsymbol{f} = (f_0, f_1, \ldots, f_{n-1})$ and $\overline{\boldsymbol{f}} = (f_0, \psi f_1, \ldots, \psi^{n-1} f_{n-1}) \in R_q$. To compute the polynomial multiplication $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in R_q$, first we precompute all $2n$ powers of ψ and ψ^{-1} and then store n powers of ψ and ψ^{-1} with bit-reversed order in look-up tables Ψ_{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 $\boldsymbol{c} =$

Input: Polynomial $\boldsymbol{a}' \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$, and a LUT $\Psi_{rev}^{-1} \in \mathbb{Z}_q^n$ in bit-reversed order
Output: Polynomial $\boldsymbol{a} = NTT^{-1}(\boldsymbol{a}') \in R_q$

$t = 1$;
for $m = n$ **to** 2 by $m = m/2$ **do**
  $h = m/2$, $j_1 = 0$;
  for $i = 0$ **to** $h - 1$ **do**
    $j_2 = j_1 + t - 1$;
    $S = \Psi_{rev}^{-1}[h + i]$;
    for $j = j_1$ **to** j_2 **do**
      $U = a_j$;
      $V = a_{j+t}$;
      $a_j = U + V \bmod q$;
      $a_{j+t} = (U - V) * S \bmod q$;
    endfor
    $j_1 = j_1 + 2t$;
  endfor
  $t = 2t$;
endfor
for $i = 0$ **to** $n - 1$ **do**
  $a_i = a_i * n^{-1} \bmod q$;
endfor
return $\boldsymbol{a}$.

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

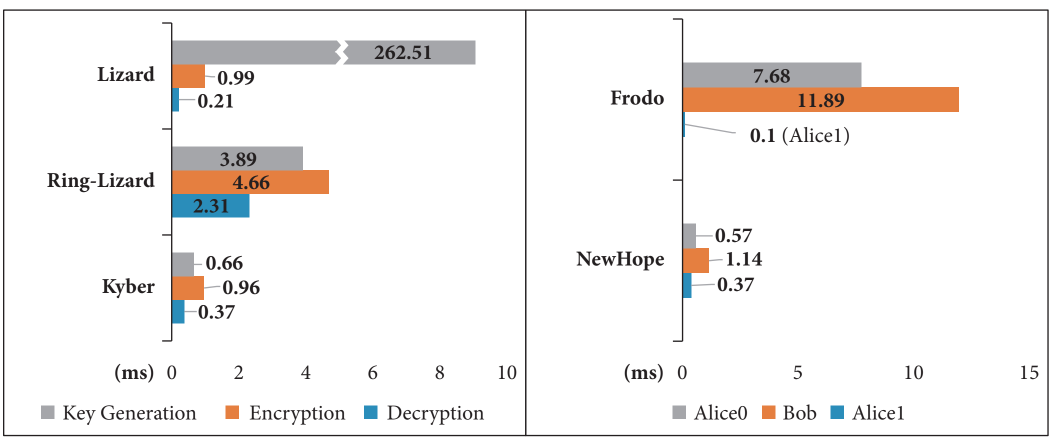

Figure 3: Running time (ms) of lattice-based cryptosystems Lizard, ring-Lizard, Kyber, Frodo, and NewHope on Firefox.

$(1, \psi^{-1}, \ldots, \psi^{-(n-1)}) \circ NTT^{-1}(NTT(\overline{\boldsymbol{a}}) \circ NTT(\overline{\boldsymbol{b}}))$, 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.

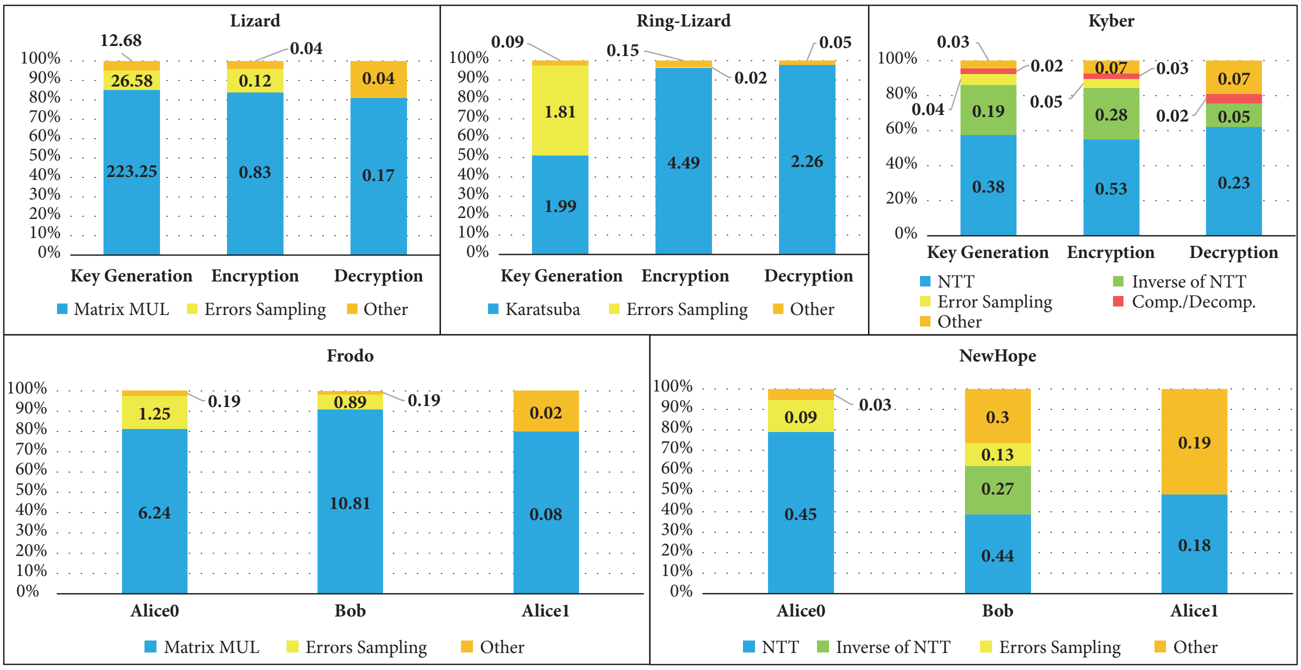

Figure 4: Decomposition of computation time (ms) of Lizard, ring-Lizard, Kyber, Frodo, and NewHope on Firefox.

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 $\mathbf{s}$ in key generation and $\mathbf{r}$ 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 ± 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 $\mathbf{A}$ and $NTT(\lfloor q/2 \rceil * \mathbf{m})$ and sampled the error vectors from a binomial distribution B_4. Each element of matrices and vectors in Kyber is a polynomial over R_q with degree equal to $n - 1$ ($q \equiv 1 \bmod 2n$); hence, NTT can be applied to Kyber to effectively compute polynomial multiplication. Let i, j, k be positive integers; we assume a matrix $\mathbf{A} = (\mathbf{a}_{ij}) \in (R_q)^{k \times k}$ is in NTT domain, and the coefficients of each element $\mathbf{a}_{ij}$ are in bit-reversed order. In key generation, we performed NTT on error vectors such that the component of public key $\mathbf{b} = NTT^{-1}(\mathbf{A}NTT(\mathbf{s}) + NTT(\mathbf{e}))$, only 6 calls of NTT and 3 calls of NTT^{-1} are necessary if $k = 3$. Similarly, we computed $NTT^{-1}(NTT(\mathbf{A}^T)NTT(\mathbf{r}) + NTT(\mathbf{e}_1))$ and $NTT^{-1}(NTT(\mathbf{b}^T)NTT(\mathbf{r}) + NTT(\mathbf{e}_2) + NTT(\lfloor q/2 \rceil * \mathbf{m}))$ by invoking NTT 10 times and NTT^{-1} 4 times in encryption and outputted $NTT^{-1}(NTT(\mathbf{s})^T\mathbf{u})$ by invoking NTT 4 times and NTT^{-1} 1 time in decryption.

For Frodo, we skipped the generation of the $\mathbf{seed}_{\mathbf{A}}$ from a binary string and precomputed the matrix $\mathbf{A}$ 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 NTT/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 $\boldsymbol{a}$ on both Alice's and Bob's sides and sent polynomials $\boldsymbol{b}, \boldsymbol{u}, \boldsymbol{r}$ directly. Comparing our implementation with the approach in [24], we only computed $NTT^{-1}(\boldsymbol{b} \circ NTT(\boldsymbol{s}_1)) + \boldsymbol{e}_2$ on Bob's side so that the computation of $NTT(\boldsymbol{e}_2)$ 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/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

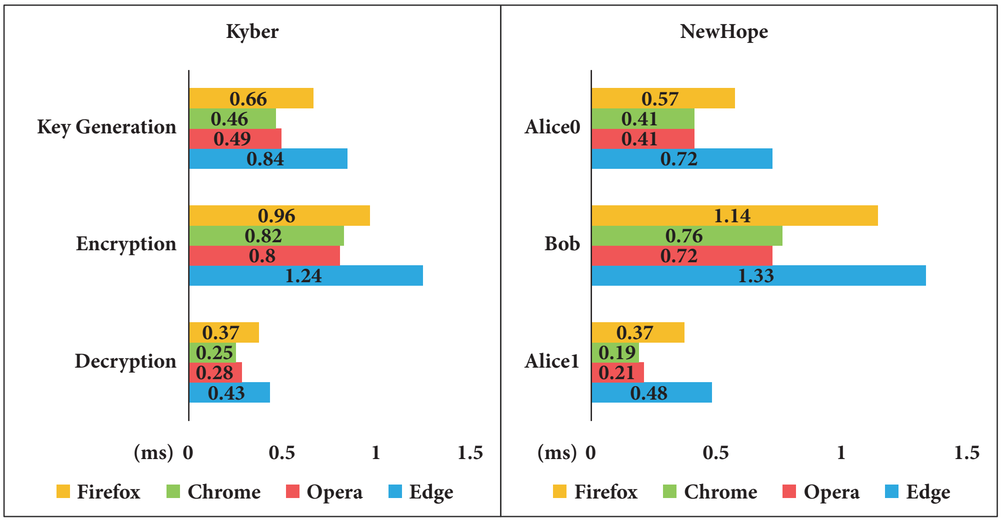

Figure 5: Running time (ms) of Kyber and NewHope on Firefox, Google Chrome, Opera, and Microsoft Edge.

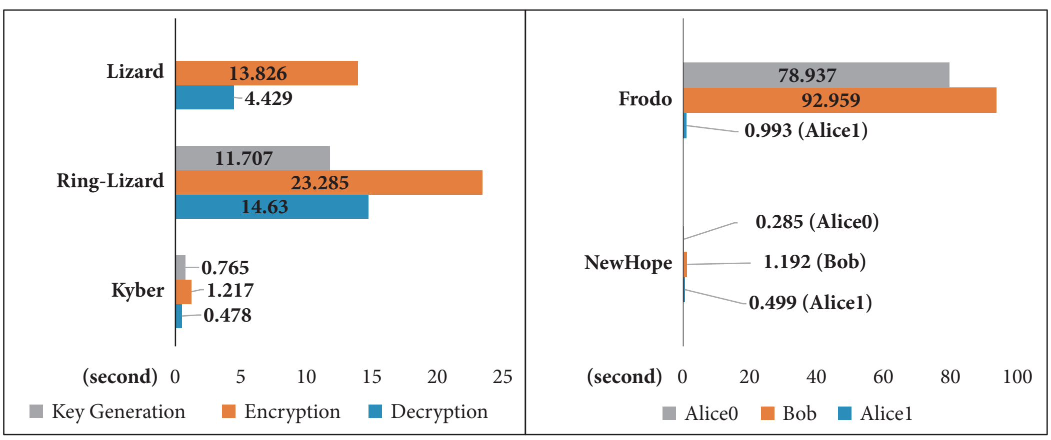

Figure 6: Running time (second) of lattice-based cryptosystems Lizard, ring-Lizard, Kyber, Frodo, and NewHope on Tessel2.

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

Table 2: Performance results on Android phone.

	Average running time (ms)		
	Key Generation	Encryption	Decryption
Lizard	1575.91	38.62	9.63
Ring-Lizard	13.24	15.05	5.61
Kyber	3.57	5.78	2.14
	Average running time (ms)		
	Alice0	Bob	Alice1
Frodo	38.10	88.03	0.79
NewHope	5.14	10.08	2.68

Table 3: Performance results on WSH.

	Average running time (ms)		
	Key Generation	Encryption	Decryption
Lizard	27021.1	372.2	72.3
Ring-Lizard	145.44	283.19	140.37
Kyber	9.89	15.76	6.15
	Average running time (ms)		
	Alice0	Bob	Alice1
Frodo	827.03	1102.92	12.70
NewHope	8.85	17.51	6.67

Table 4: Performance results on Node.js.

	Average running time (ms)		
	Key Generation	Encryption	Decryption
Lizard	271.08	6.54	1.47
Ring-Lizard	1.86	2.48	1.22
Kyber	0.44	0.67	0.27
	Average running time (ms)		
	Alice0	Bob	Alice1
Frodo	7.04	14.73	0.13
NewHope	0.20	0.64	0.19

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 Bod'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.

	Average running time (ms)		
	Key Generation	Encryption	Decryption
Lizard	209.99	1.43	0.41
Ring-Lizard	3.10	4.66	2.44
Kyber	0.80	1.20	0.49
	Average running time (ms)		
	Alice0	Bob	Alice1
Frodo	6.88	10.46	0.09
NewHope	0.75	1.32	0.43

Table 6: Performance results on Pacifista.

	Average running time (ms)		
	Key Generation	Encryption	Decryption
Lizard	1301.54	24.67	5.76
Ring-Lizard	43.77	34.24	17.97
Kyber	7.39	4.49	1.78
	Average running time (ms)		
	Alice0	Bob	Alice1
Frodo	35.80	43.96	0.75
NewHope	2.57	9.27	3.08

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

```
<!-- lizard.html -->
<!DOCTYPE html>
<html>
<head><title>Lizard</title>
<meta charset="UTF-8">
<script type="text/ javascript "
  src ="../ Utils /prng.js"></script>
<script type="text/ javascript "
  src ="lizard_random_values.js"></script>
<script type="text/ javascript "
  src ="lizard.js"></script>
</head>
<body></body>
</html>
```

Pseudocode 1

```
// the parameters can be changed
var m = 960, n = 608, l = 256, t = 2, p = 256, q = 1024; // h,r,...
function testlizard() { //main function
  //...
  randomPlaintext();
  keyGeneration(l,m,n,q);
  encrypt(l,n,p,q);
  decrypt(l,q,t);
  //...
}
testlizard(); // invoke the main function
```

Pseudocode 2

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:

```
var tessel = require('tessel');
//functions
```

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
```

osascript

```
$ osascript new_lizard.js
```

Pacifista

```
$ bin/pacifista scripts/new_lizard.js
```

Display. To measure the running time, we can invoke `Date.now()` function or `new Date().getTime()` function, but WSH only supports the latter.

`console.log()` writes a message to the console. If the code is executed on WSH, we should invoke `WScript.Echo()` to display the message:

```
function print(message) {
    //WScript.Echo(message);
    console.log(message);
}
```

The result will be outputted as follows (the binary plaintext is generated randomly) (see Pseudocode 4).

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 onCreat (Bundle savedInstanceState ) {
    super.onCreate ( savedInstanceState );
    webview = new WebView (this);
    webview.getSettings ()
      . 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 =
0,0,1,0,1,0,0,1,1,1,0,0,1,0,0,1,0,1,1,1,1,//...
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

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] J. Bernstein, J. Buchmann, and E. Dahmen, *Post-Quantum Cryptography*, Springer, 2009.

[3] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, pp. 84–93, ACM, Baltimore, Md, USA, May 2005.

[4] C. Peikert, V. Vaikuntanathan, and B. Waters, "A framework for efficient and composable oblivious transfer," in *Proceedings of the 28th Annual International Cryptology Conference – (CRYPTO '08)*, vol. 5157 of *Lecture Notes in Computer Science*, pp. 554–571, 2008.

[5] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology—EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings*, H. Gilbert, Ed., vol. 6110 of *Lecture Notes in Computer Science*, pp. 1–23, Springer, Berlin, Germany, 2010.

[6] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption," in *Topics in Cryptology—CT-RSA 2011*, vol. 6558 of *Lecture Notes in Computer Science*, pp. 319–339, Springer, Heidelberg, Germany, 2011.

[7] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 553–570, 2015.

[8] T. Pöppelmann and T. Güneysu, "Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware," in *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America – (LATINCRYPT '12)*, vol. 7533 of *Lecture Notes in Computer Science*, pp. 139–158, 2012.

[9] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: a signature scheme for embedded systems," in *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems – (CHES '12)*, vol. 7428, pp. 530–547, 2012.

[10] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, "On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 512–529, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] R. De Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede, "Efficient software implementation of ring-LWE encryption," in *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition, DATE 2015*, pp. 339–344, 2015.

[12] Y. Yuan, K. Fukushima, S. Kiyomoto, and T. Takagi, "Memory-constrained implementation of lattice-based encryption scheme on standard Java Card," in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 47–50, 2017.

[13] Y. Yuan, C. Cheng, S. Kiyomoto, Y. Miyake, and T. Takagi, "Portable Implementation of Lattice-based Cryptography using JavaScript," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 309–327, 2016.

[14] Y. Yuan, J. Xiao, K. Fukushima et al., "Portable implementation of post-quantum encryption schemes and key exchange protocols on JavaScript-enabled platforms," in *Proceedings of the Symposium on Cryptography and Information Security (SCIS '18)*, pp. 1–8, 2018, https://www.iwsec.org/scis/2018/program.html.

[15] J. H. Cheon, D. Kim, J. Lee, and Y. Song, "Lizard: Cut off the tail! Practical post-quantum public-key encryption from LWE and LWR," IACR Cryptology ePrint Archive 2016/1126, 2016.

[16] J. Bos, L. Ducas, E. Kiltz et al., "CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM," IACR Cryptology ePrint Archive 2017/634, 2017.

[17] J. Bos, C. Costello, L. Ducas et al., "Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*, pp. 1006–1018, Austria, October 2016.

[18] E. Alkim, L. Ducas, T. Pöppelmann et al., "Post-quantum key exchange - a new hope," in *Proceedings of the 25th USENIX Security Symposium*, pp. 327–343, 2016.

[19] M. R. Albrecht, B. R. Curtis, A. Deo et al., "Estimate all the (LWE, NTRU) schemes!," IACR Cryptology ePrint Archive 2018/331, 2018.

[20] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[21] R. Primas, P. Pessl, and S. Mangard, "Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption," in *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems – (CHES '17)*, vol. 10529 of *Lecture Notes in Computer Science*, pp. 513–533, 2017.

[22] O. Tobias, T. Schneider, T. Pöppelmann, and T. Güneysu, "Practical CCA2-secure and masked ring-LWE implementation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 142–174, 2018.

[23] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers," in *Proceedings of the 4nd International Conference on Cryptology and Information Security in Latin America – (LATINCRYPT '15)*, vol. 9230 of *Lecture Notes in Compuer Science*, pp. 346–365, Springer, Cham, 2015.

[24] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Cryptology and network security*, vol. 10052 of *Lecture Notes in Comput. Sci.*, pp. 124–139, Springer, Cham, 2016.

[25] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with rounding, revisited," in *Proceedings of the 33rd Annual International Cryptology Conference – (CRYPTO '13)*, vol. 8042, pp. 57–74, 2013.

[26] V. Lyubashevsky, "Lattice signatures without trapdoors," in *Advances in Cryptology—EUROCRYPT 2012. EUROCRYPT 2012*, vol. 7237 of *Lecture Notes in Computer Science*, pp. 738–755, Springer, 2012.

[27] S. Sinha Roy, F. Vercauteren, and I. Verbauwhede, "High Precision Discrete Gaussian Sampling on FPGAs," in *Selected Areas in Cryptography – SAC 2013*, vol. 8282 of *Lecture Notes in Computer Science*, pp. 383–401, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[28] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor, "Gaussian random number generators," *ACM Computing Surveys*, vol. 39, no. 4, pp. 1–38, 2007.

[29] D. E. Knuth and A. C. Yao, "The complexity of nonuniform random number generation," in *Algorithms and Complexity: New Directions and Recent Results*, pp. 357–428, Academic Press, 1976.

[30] J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden, "Discrete Ziggurat: A Time-Memory Trade-Off for Sampling from a Gaussian Distribution over the Integers," in *Selected Areas in Cryptography – SAC 2013*, vol. 8282 of *Lecture Notes in Computer Science*, pp. 402–417, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[31] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3 of *Special issue on innovations in theoretical computer science 2012*, pp. 1–36, Part II edition, 2014.

[32] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Designs, Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[33] C. Peikert, "Lattice cryptography for the internet," in *Post-Quantum Cryptography*, vol. 8772 of *Lecture Notes in Computer Science*, pp. 197–219, Springer International Publishing, Cham, Switzerland, 2014.

[34] E. Alkim, P. Jakubeit, and P. Schwabe, "A new hope on ARM Cortex-M," IACR Cryptology ePrint Archive 2016/758, 2016.

[35] D. Cabarcas, P. Weiden, and J. Buchmann, "On the efficiency of provably secure NTRU," in *Proceedings of the 6th International Workshop on Post-Quantum Cryptography – (PQCrypto 2014)*, vol. 8772 of *Lecture Notes in Computer Science*, pp. 22–39, 2014.

[36] S. S. Roy, F. Vercauteren, N. Mentens et al., "Compact ring-LWE cryptoprocessor," in *Proceedings of the 16th International Conference on Cryptographic Hardware and Embedded Systems (CHES '14)*, vol. 8731 of *Lecture Notes in Computer Science*, pp. 371–391, 2014.