OpenSSLNTRU: Faster post-quantum TLS key exchange

Daniel J. Bernstein\textsuperscript{1,2}, Billy Bob Brumley\textsuperscript{3}, Ming-Shing Chen\textsuperscript{2}, and Nicola Tuveri\textsuperscript{3}

authorcontact-opensslntru@box.cr.yp.to

\textsuperscript{1}Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607-7045, USA
\textsuperscript{2}Ruhr University Bochum, Bochum, Germany
\textsuperscript{3}Tampere University, Tampere, Finland

2021.06.15

Abstract

Google’s CECPQ1 experiment in 2016 integrated a post-quantum key-exchange algorithm, newhope1024, into TLS 1.2. The Google-Cloudflare CECPQ2 experiment in 2019 integrated a more efficient key-exchange algorithm, ntruhrss701, into TLS 1.3.

This paper revisits the choices made in CECPQ2, and shows how to achieve higher performance for post-quantum key exchange in TLS 1.3 using a higher-security algorithm, sntrup761. Previous work had indicated that ntruhrss701 key generation was much faster than sntrup761 key generation, but this paper makes sntrup761 key generation much faster by generating a batch of keys at once.

Batch key generation is invisible at the TLS protocol layer, but raises software-engineering questions regarding the difficulty of integrating batch key exchange into existing TLS libraries and applications. This paper shows that careful choices of software layers make it easy to integrate fast post-quantum software, including batch key exchange, into TLS with minor changes to TLS libraries and no changes to applications.

As a demonstration of feasibility, this paper reports successful integration of its fast sntrup761 library, via a lightly patched OpenSSL, into an unmodified web browser and an unmodified TLS terminator. This paper also reports TLS 1.3 handshake benchmarks, achieving more TLS 1.3 handshakes per second than any software included in OpenSSL.

1 Introduction

The urgency of upgrading TLS to post-quantum encryption—see Appendix A—has prompted a tremendous amount of work. There were already 69 proposals for post-quantum cryptography (PQC) submitted to NIST’s Post-Quantum Cryptography Standardization Project in 2017, including 49 proposals for post-quantum encryption. Each proposal included complete software implementations of the algorithms for key generation, encryption, and decryption. Given the cryptographic agility of TLS, one might imagine that TLS software can simply pick a post-quantum algorithm and use it. However, there are many constraints making this more difficult than it sounds, including the following:

- Performance: Post-quantum algorithms can send much more data than elliptic-curve cryptography (ECC), and can take many more CPU cycles. Performance plays a “large role” [44] in the NIST standardization project.
- Integration: Many assumptions about how cryptography works are built into the TLS protocol and existing TLS software. These range from superficial assumptions about the sizes of objects to more fundamental structural assumptions such as the reliance of TLS 1.3 upon “Diffie–Hellman”—a key-exchange data flow not provided by any of the proposals for NIST standardization.
- Security: 30 of the 69 proposals were broken by the end of 2019 [10]. New attacks continue to appear: e.g., [6] uses under a single second of CPU time to break any ciphertext sent by the “Round2” lattice-based proposal.

“Hybrid” approaches that encrypt with ECC and with a post-quantum system are recommended (see, e.g., [15, 43]) because they retain the security of ECC even if the post-quantum system is broken—but, for an attacker with a future quantum computer, a ciphertext encrypted with ECC and with a broken post-quantum system is a ciphertext encrypted with two broken systems.

In July 2020, the NIST project began its third round [1], selecting 4 “finalist” encryption proposals for consideration for standardization at the end of the round, and 5 “alternate” encryption proposals for consideration for standardization after a subsequent round. Meanwhile, there have been various experiments successfully integrating post-quantum encryption systems into TLS. The proposals that have attracted the most attention, and that are also the focus of this paper, are “small” lattice proposals. These include

- three of the finalist proposals (Kyber [4], NTRU [19], and SABER [5]), although NIST says it will standardize at most one of these three;
- one of the alternate proposals (NTRU Prime);
• the newhope1024 algorithm [2] used inside Google’s CECPQ2 experiment in 2016; and
• the ntruhrss701 algorithm (a variant of one of the algorithms in the NTRU proposal) used inside the Google-Cloudflare CECPQ2 experiment in 2019.

These are called “small” because they use just a few kilobytes for each key exchange—much more traffic than ECC, but much less than many other post-quantum proposals.

CECPQ2 actually included two experiments: CECPQ2a used ntruhrss701, while CECPQ2b used an isogeny-based proposal. Compared to ntruhrss701, the isogeny-based proposal had smaller keys and smaller ciphertexts, but used much more CPU time, so it outperformed CECPQ2a only on the slowest network connections.

In general, the importance of a few kilobytes depends on the network speed and on how often the application creates new TLS sessions. A typical multi-megabyte web page is unlikely to notice a few kilobytes, even if it retrieves resources from several TLS servers. A session that encrypts a single DNS query is handling far less data, making the performance of session establishment much more important. Similar comments apply to CPU time.

1.1 Contributions of this paper

This paper introduces OpenSSLNTRU, an improved integration of post-quantum key exchange into TLS 1.3. OpenSSLNTRU improves upon the post-quantum portion of CECPQ2 in two ways: key-exchange performance and TLS software engineering. These are linked, as explained below. OpenSSLNTRU offers multiple choices of key sizes; for concreteness we emphasize one option, sntrup761 [13], to compare to CECPQ2’s ntruhrss701.

Each of ntruhrss701 and sntrup761 is a “key-encapsulation mechanism” (KEM) consisting of three algorithms: a key-generation algorithm generates a public key and a corresponding secret key; an “encapsulation” algorithm, given a public key, generates a ciphertext and a corresponding session key; a “decapsulation” algorithm, given a secret key and a ciphertext, generates the corresponding session key. The key exchange at the beginning of a TLS session involves one keygen, one enc, and one dec. Before our work, both KEMs already had software optimized for Intel Haswell using AVX2 vector instructions (see Appendix B); keygen was 3.03× slower for sntrup761 than for ntruhrss701, making total keygen+enc+dec 2.57× slower.

One can remove keygen cost by reusing a key for many TLS sessions, but it is commonly recommended to use each key for just one session. There are several reasons for this recommendation: for example, in TLS 1.3 key exchange (followed by both CECPQ2 and OpenSSLNTRU), the client performs key generation, and reusing keys would open up yet another mechanism for client tracking.

<table>
<thead>
<tr>
<th>Cryptosystem</th>
<th>CECPQ2</th>
<th>OpenSSLNTRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key+ciphertext bytes</td>
<td>2276</td>
<td>2197</td>
</tr>
<tr>
<td>Keygen cycles</td>
<td>269191</td>
<td>814608 → 156317</td>
</tr>
<tr>
<td>Enc cycles</td>
<td>26510</td>
<td>48892 → 46914</td>
</tr>
<tr>
<td>Dec cycles</td>
<td>63375</td>
<td>59404 → 56241</td>
</tr>
<tr>
<td>Core-SVP security</td>
<td>$2^{156}$</td>
<td>$2^{153}$</td>
</tr>
<tr>
<td>Cyclotomic concerns</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

This paper instead directly addresses the speed problem with sntrup761 key generation, by making sntrup761 key generation much faster. Our sntrup761 software outperforms the latest ntruhrss701 software, and at the same time sntrup761 has a higher security level than ntruhrss701. See Table 1 and Appendix B.

The main bottleneck in sntrup761 key generation is computation of certain types of inverses. This paper speeds up those inversions using “Montgomery’s trick”, the simple idea of computing two independent inverses $1/a$ and $1/b$ as $br$ and $ar$ respectively, where $r = 1/ab$. Repeating this trick converts, e.g., 32 inversions into 1 inversion plus 93 multiplications.

This paper generates a batch of 32 independent keys, combining independent reciprocals across the batch. This batch size is large enough for inversion time to mostly disappear, and yet small enough to avoid creating problems with latency, cache misses, etc. We designed new algorithms and software to optimize sntrup761 multiplications, since the multiplications used previously were “big×small” multiplications while Montgomery’s trick needs “big×big” multiplications; see Section 3.

A new key sent through TLS could have been generated a millisecond earlier, a second earlier, or a minute earlier; this does not matter for the TLS protocol. However, for TLS software, batching keys is a more interesting challenge, for two reasons. First, key generation is no longer a pure stateless subroutine inside one TLS session, but rather a mechanism sharing state across TLS sessions. Second, the TLS software ecosystem is complicated (and somewhat ossified), with many different applications using many different libraries, so the same state change needs to be repeated in many different
pieces of TLS software.

To address the underlying problem, this paper introduces a new choice of software layers designed to decouple the fast-moving post-quantum software ecosystem from the TLS software ecosystem. The point of these layers is that optimization of post-quantum software does not have to worry about any of the complications of TLS software, and vice versa. As a case study demonstrating the applicability of these layers, this paper describes successful integration of its new sntrup761 library, including batch key generation, into an existing web browser communicating with an existing TLS terminator, using OpenSSL on both ends. This demo involves no changes to the web browser, no changes to the TLS terminator, and very few changes to OpenSSL.

The integration of OpenSSLNTRU into TLS means that, beyond microbenchmarks, we can and do measure full TLS handshake performance. The bottom line is that, in a controlled and reproducible end-to-end lab experiment, sntrup761 completes more sessions per second than commonly deployed pre-quantum X25519. This remains true even when we replace sntrup761 with higher-security sntrup857.

One should not conclude that sntrup761 and sntrup857 cost less than ECC overall. This experiment used an unloaded high-bandwidth local network, making communication essentially invisible, whereas reasonable estimates of communication costs (see generally Appendix D) say that every lattice system costs more than ECC. Nevertheless, eliminating 70% of the sntrup761 CPU time significantly reduces total sntrup761 costs, in effect assigning higher decision-making weight to size and, most importantly, security.

2 Background

2.1 Polynomial rings in NTRU Prime

Streamlined NTRU Prime [13], abbreviated sntrup, uses arithmetic in finite rings \( R/3 = (\mathbb{Z}/3)[x]/(x^3 - x - 1) \) and \( R/q = (\mathbb{Z}/q)[x]/(x^3 - x - 1) \), where \( R = \mathbb{Z}[x]/(x^3 - x - 1) \). The parameters \( p, q \) are chosen so that \( R/q \) is a field.

Short means the set of polynomials in \( R \) that are small, meaning all coefficients in \( \{-1, 0, 1\} \), and weight \( w \), meaning that exactly \( w \) coefficients are nonzero, where \( w \) is another parameter. The parameters \( (p, q, w) \) are \((653, 4621, 288), (761, 4591, 286), (857, 5167, 322)\) for the KEMs sntrup653, sntrup761, sntrup857 respectively.

2.2 Montgomery’s trick for batch inversion

In this section, we review Montgomery’s trick for batch inversion [32] as applied to many inputs. The algorithm batchInv takes \( n \) elements \((a_1, a_2, \ldots, a_n)\) in a ring, and outputs their multiplicative inverses \((a_1^{-1}, a_2^{-1}, \ldots, a_n^{-1})\). Montgomery’s trick for batch inversion proceeds as follows:

1. Let \( b_1 = a_1 \) and compute \( b_i = a_i \cdot b_{i-1} \) for \( i \in (2, \ldots, n) \).
   After \( n - 1 \) multiplications, we obtain 
   \[
   (b_1, b_2, \ldots, b_n) = (a_1, a_1 \cdot a_2, a_1 \cdot a_2 \cdot a_3, \ldots, \Pi_{i=1}^n a_i) 
   \]

2. Compute the single multiplicative inverse 
   \[
   t_n = b_n^{-1} = (\Pi_{i=1}^n a_i)^{-1} 
   \]

3. Compute \( c_i = t_i \cdot b_{i-1} \) and \( t_{i-1} = t_i \cdot a_i \) for \( i \in (n, \ldots, 2) \).
   After \( 2n - 2 \) multiplications, we have two lists 
   \[
   (c_n, \ldots, c_2) = (a_n^{-1}, \ldots, a_2^{-1}) 
   \]
   \[
   (t_{n-1}, \ldots, t_2, t_1) = ((\Pi_{i=1}^{n-1} a_i)^{-1}, \ldots, (a_1 \cdot a_2)^{-1}, a_1^{-1}) 
   \]

4. Output \((a_1^{-1}, a_2^{-1}, \ldots, a_n^{-1})\).

In summary, the algorithm uses \( 3n - 3 \) multiplications and one inversion to compute \( n \) inverses.

2.3 NTT-based multiplication

This section reviews techniques for polynomial multiplication commonly used in lattice-based cryptography. We adopt terminology from [7].

The number theoretic transform (NTT) algorithm maps an element in a polynomial ring into values by lifting the ring element to a polynomial and evaluating the polynomial on a particular set. An NTT-based multiplication algorithm applies NTTs to two input elements in the polynomial ring, performs component-wise multiplication for the transformed values, and applies an inverse NTT, converting the multiplied values back to the product in the same form of inputs.

Computing a size-\( n \) NTT, where \( n \) is a power of 2, comprises \( \log_2 n \) stages of the radix-2 FFT trick. Given a polynomial ring \((\mathbb{Z}/q)[x]/(x^n - b^2)\) where \( b \in \mathbb{Z}/q \), the FFT trick maps elements in \((\mathbb{Z}/q)[x]/(x^n - b^2)\) to \(((\mathbb{Z}/q)[x]/(x^n - b)) \times ((\mathbb{Z}/q)[x]/(x^n + b)))\). Due to the Chinese remainder theorem (CRT), the mapping is invertible when \( 2b \) is invertible. Specifically, let \( f = f_0 + f_1x + \cdots + f_{n-1}x^{n-1} \in (\mathbb{Z}/q)[x]/(x^n - b^2) \). The trick maps \( f \) to 

\[
(f \text{ mod } (x^{n/2} + b), f \text{ mod } (x^{n/2} - b))
\]

\[
= ((f_0 - b f_{n/2}) + \cdots + (f_{n/2-1} - b f_{n-1})x^{n/2-1},
(f_0 + b f_{n/2}) + \cdots + (f_{n/2-1} + b f_{n-1})x^{n/2-1})
\]

with \( n \) multiplications by \( b \), \( n/2 \) additions, and \( n/2 \) subtractions. Setting \( b = 1 \), by recursively applying the FFT trick, an NTT transforms \( f \) into a list \( \hat{f} = (f_0, \ldots, f_{n/2}, \ldots, f_{n-1}) \in (\mathbb{Z}/q)\) where \( \hat{f}_j = f \text{ mod } (x - \psi^j) = \sum_{i=0}^{n/2} f_i \psi^j \), and \( \psi \in \mathbb{Z}/q \) is a primitive \( n\)-th root of unity, i.e., \( \psi^{n/2} = -1 \).
When \( \mathbb{Z}/q \) lacks appropriate roots of unity, Schönhage’s trick [39] manufactures them by introducing an intermediate polynomial ring. Given \( f \in (\mathbb{Z}/q)[x]/(x^{2mn} - 1) \), the trick first introduces a new variable \( y = x^m \) and maps \( f \) from \( (\mathbb{Z}/q)[x]/(x^{2mn} - 1) \) to \( (\mathbb{Z}/q)[y]/(y^{2n} - y) \). Then, it lifts \( f \) to \( (\mathbb{Z}/q)[y]/(y^{2n} - 1) \), which is a polynomial in variable \( y \) with coefficients in \( (\mathbb{Z}/q)[x] \). Since the coefficients of \( f \) are polynomials with degree less than \( m \), it is safe to map them to \( (\mathbb{Z}/q)[x]/(x^{2m} + 1) \) such that coefficient multiplication needs no reduction by \( x^{2m} + 1 \). Now \( x \in (\mathbb{Z}/q)[x]/(x^{2m} + 1) \) is a primitive \( 4m \)-th root of unity, since \( x^{2m} = -1 \).

Nussbaumer’s trick [35] is another method to manufacture roots of unity. Given \( f \in (\mathbb{Z}/q)[x]/(x^{2mn} - 1) \), the trick maps \( f \) to \( ((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^m - y) \), lifts to \( ((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^{2m} - y) \), and maps to \( ((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^{2m} - 1) \) for \( n \geq m \). As noted in [7], Nussbaumer’s trick sometimes uses slightly smaller ring extensions than Schönhage’s trick, but Schönhage’s trick is more cache-friendly, since it uses contiguous data in \( x \).

### 2.4 The AVX2 instruction set

Since NIST specified Intel Haswell CPU as its highest priority platform for performance evaluation [34], we optimize snttrup for the Haswell architecture in this work.

Specifically, we target the Advanced Vector Extensions 2 (AVX2) instruction set. AVX is a single-instruction-multiple-data (SIMD) instruction set in modern (decade or less) x86 CPUs. It provides sixteen 256-bit \( \text{ymm} \) registers; each \( \text{ymm} \) register splits into two 128-bit \( \text{xmm} \) lanes. The instruction set treats data in \( \text{ymm} \) registers as lanes (independent partitions) of \( 32 \times 8 \)-bit, \( 16 \times 16 \)-bit, \( 8 \times 32 \)-bit, etc.; every instruction operates simultaneously on the partitioned data in the \( \text{ymm} \) registers. In 2013, the Haswell architecture extended AVX to AVX2 for enhanced integer operations.

### 2.5 Integrating cryptographic primitives

Related to OpenSSLNTRU, several previous works studied integrations between post-quantum implementation and real world applications.

The Open Quantum Safe (OQS) project [42] includes a library of quantum-resistant cryptographic algorithms, and prototype integrations into protocols and applications. It also includes (and requires) a fork of the OpenSSL project. Conversely, in our contribution we apply a minimal patchset, striving to maintain API and ABI compatibility with the OpenSSL version available to the end-users. This avoids the need of recompiling existing applications to benefit from the new library capabilities. We also note that the experiment we present in this manuscript is limited to one candidate and two sets of parameters (\texttt{snttrup761} and \texttt{snttrup857}), while the OQS project provides implementations for all finalists.

Similarly, the PQClean project [28] collects a number of implementations for the candidates. However, it does not aim to include integration into higher-level applications or protocols.

Schwabe, Stbila, and Wiggers [40] present an alternative to the TLS 1.3 handshake to solve both key exchange and authentication using post-quantum KEM. In contrast, for our experiment we aimed at full compatibility with the TLS 1.3 ecosystem, focusing exclusively on the key exchange. This ensures post-quantum confidentiality, but does not address the post-quantum authentication concerns.

Our approach to OpenSSL integration via an ENGINE module is based on the methodology suggested in [45], where the authors instantiated \texttt{libsuola}. In this context, an ENGINE module is a a dynamically loadable module. Using a dedicated API, such a module is capable of injecting new algorithms or overriding existing ones. The implementations it provides can be backed by a hardware device, or be entirely software based. Our new ENGINE, \texttt{engNTRU}, builds upon \texttt{libbecc} [16], which is itself derived from \texttt{libsuola}. Both previous works applied the ENGINE framework to integrate alternative ECC implementations. The latter is particularly close to \texttt{engNTRU}, as it also featured a transparent mechanism to handle batch key generation. Section 4.2 details how \texttt{engNTRU} evolved from these works and the unique features it introduces.

Shacham and Boneh [41] integrated RSA batching to improve SSL handshake performance already in 2001. However, their methodology required integrating changes directly in the server application. In contrast, OpenSSLNTRU acts on the middleware level, transparent to client and server applications.

### 3 Batch key generation for snttrup

This section presents batch key generation for snttrup and its optimization. Section 3.1 shows the batch key generation algorithm with Montgomery’s inversion-batching trick. Section 3.2 and Section 3.3 present our polynomial multiplication and its optimization in \( (\mathbb{Z}/3)[x] \) and \( (\mathbb{Z}/q)[x] \), respectively. Section 3.4 shows the benchmark results.

#### 3.1 Batch key generation

The snttrup key generation algorithm KeyGen outputs an snttrup key pair. It proceeds as follows:

1. Generate a uniform random small element \( g \in \mathcal{R} \). Repeat this step until \( g \) is invertible in \( \mathcal{R}/3 \).
2. Compute \( 1/g \) in \( \mathcal{R}/3 \).
3. Generate a uniform random \( f \in \text{Short} \).
4. Compute \( h = g/(3f) \) in \( \mathcal{R}/q \).
5. Output \((h, (f, 1/g))\) where \( h \) is the public key and \((f, 1/g) \in \text{Short} \times \mathcal{R}/3 \) is the secret key.
Algorithm 1 (BatchKeyGen) batches sntrup key generation. We use two lists for storing $n$ batches of $g \in \mathcal{R}$ and $f \in \text{Short}$, then process the $n$ batches of computation in one subroutine. The key idea is to replace the $2n$ inversions by two batchInv for $\mathcal{R}/3$ and $\mathcal{R}/q$, respectively. As seen in Section 2.2, batchInv turns $n$ inversions into $3n - 3$ multiplications and one inversion. Considering performance, ring multiplication then becomes the critical part. Hence, Section 3.2 and Section 3.3 present optimized ring multiplication implementations, used in batchInv.

Another difference is the invertibility check in $\mathcal{R}/3$ for the element $g$. Previous NTRU Prime software checks invertibility as a side effect of computing $1/g$ with a constant-time algorithm [12] for extended GCD. Calling batchInv removes this side effect and requires a preliminary check for invertibility of each $g$. In Section 3.1.1 we optimize an isInvertible subroutine for this test.

Algorithm 1 BatchKeyGen

Input: an integer $n$
Output: $n$ key pairs of sntrup

1: $G \leftarrow \varnothing$ \Comment*{an empty list}
2: $F \leftarrow \varnothing$
3: while $\text{len}(G) < n$ do
4: \hspace{1em} $g \leftarrow \mathcal{R}/3$ \Comment*{$S$: uniform random}
5: \hspace{1em} if not isInvertible($g$) : continue
6: \hspace{1em} $f \leftarrow \text{Short}$
7: \hspace{1em} $G$.append($g$)
8: \hspace{1em} $F$.append($f$)
9: end while
10: $\bar{G} \leftarrow \text{batchInv}(G)$
11: $\bar{F} \leftarrow \text{batchInv}([3 \cdot f \text{ for } f \in F])$
12: $H \leftarrow [g \cdot \bar{f} \in \mathcal{R}/q \text{ for } g \in G, \bar{f} \in \bar{F}]$
13: return $(h, (f, g))$ for $h \in H, f \in F, g \in \bar{G}$

$q_s = \lfloor x^{D_b}/f_1 \rfloor$, where the floor function $\lfloor \cdot \rfloor$ removes the negative-degree terms from a series. In the online step, compute $h = \lfloor g \cdot q_s/x^{D_b} \rfloor = \lfloor g/f_1 \rfloor$, i.e., the quotient of the division $g \cdot q_s/x^{D_b}$. Finally, return the remainder $r = g - h \cdot f_1$. We show this gives the correct $r$ in Appendix E.

Some observations about the degree of polynomials help to accelerate the computation. While computing $h = \lfloor g \cdot q_s/x^{D_b} \rfloor$, we compute only terms with degree in the interval $[0, D_f)$, since $r = g - h \cdot f_1$ uses terms exclusively from this interval for $\deg(r) < \deg(f_1)$.

In the case of sntrup761, the polynomial $f = x^{761} - x - 1 \in (\mathbb{Z}/3)[x]$ has three factors, with degrees $\deg(f_1) = 19$, $\deg(f_2) = 60$, and $\deg(f_3) = 682$, respectively. We choose $D_1 = 32$, $D_2 = 64$, and $D_3 = 768$ for computing $g \mod f_0$ and $g \mod f_1$. For computing $g \mod f_2$, we note the pre-computed quotient $q_s = \lfloor x^{682}/(x^{682} + \cdots) \rfloor$ satisfies $\deg(q_s) = 88$. Hence, the multiplication $h = g \cdot q_s/x^{682}$ involves $\deg(g) = 768$ and $\deg(q_s) = 88$ polynomials. By partitioning the longer polynomial into several shorter segments, we perform the multiplication by several polynomial multiplications of length equal to the shorter polynomial (less than 128). Therefore, to check invertibility, we use polynomial multiplications in $(\mathbb{Z}/3)[x]$ with lengths in $\{32, 64, 128\}$.

3.2 Polynomial multiplication in $(\mathbb{Z}/3)[x]$

In this section, we describe our multiplication in $(\mathbb{Z}/3)[x]$ for sntrup, and its optimization in the AVX2 instruction set.

Based on the polynomial lengths, we implement polynomial multiplication with different algorithms. We build a 16 $\times$ 16 polynomial multiplier as a building block for schoolbook multiplication. We then use Karatsuba to build longer multipliers, such as $32 \times 32, 64 \times 64$, and further $2^i \times 2^i$. For $3 \cdot 256 \times 3 \cdot 256$ multiplications, we start from Bernstein’s 5-way recursive algorithm [8] for $(\mathbb{Z}/2)[x]$ and optimize the same idea for $(\mathbb{Z}/3)[x]$.

3.2.1 Base polynomial multiplier

For representing $(\mathbb{Z}/3)[x]$ polynomials, we adjust the values of coefficients to unsigned form and store polynomials as byte arrays, with one coefficient per byte. For example, we store the polynomial $a_0 + \cdots + a_{15}x^{15} \in (\mathbb{Z}/3)[x]$ as a byte array $(a_0, a_1, \ldots, a_{15})$ in a 16-byte xmm register.

Besides a byte array, we can view a polynomial as an integer by translating the monomial $x = 256$. For example, a degree-3 polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$ maps to the 32-bit integer $a_0 + a_1 \cdot 2^8 + a_2 \cdot 2^{16} + a_3 \cdot 2^{24}$.

In this 32-bit format, we can perform a $4 \times 4 \rightarrow 8$ polynomial multiplication using a $32 \times 32 \rightarrow 64$ integer multiplication, taking care to control the coefficient values. While calculating the polynomial product $(a_0 + a_1x + a_2x^2 + a_3x^3) \cdot (b_0 + b_1x + b_2x^2 + b_3x^3)$ with a $32 \times 32 \rightarrow 64$ integer multiplication, if all coefficients $a_i, b_i \in \{0, 1, 2\}$, a term’s maximum
possible value is \( \sum_{i+j=3} a_i b_j x^3 \leq 16 \), fitting in a byte. Hence, we use \( 4 \times 4 \) polynomial multiplication (i.e., \( 32 \times 32 \rightarrow 64 \) integer multiplication), as our building block to implement \( 16 \times 16 \) polynomial multiplication with the schoolbook algorithm.

### 3.2.2 Multiplying polynomials of length \( 3n \)

This section reduces a multiplication of \( 3n \)-coefficient polynomials in \( \mathbb{Z}/3[\mathbb{x}] \) to 5 multiplications of \( \approx n \)-coefficient polynomials, while optimizing the number of additions using techniques analogous to Bernstein’s optimizations [8] for \( \mathbb{Z}/2[\mathbb{x}] \). This section also streamlines the computation for \( \leq (3n-1) \)-coefficient polynomials, as in \texttt{strump}.

Take two polynomials \( F_0 + F_1 t + F_2 t^2 \) and \( G_0 + G_1 t + G_2 t^2 \) in \( \mathbb{Z}/3[\mathbb{x}] \), where \( \deg(F_i) < n \), \( \deg(G_i) < n \), and \( t = x^n \). Their product \( H = H_0 + H_1 t + H_2 t^2 + H_3 t^3 + H_4 t^4 \) can be reconstructed by the projective Lagrange interpolation formula

\[
H = H(0) + \frac{(t-1)(t+1)(t-x)}{x} + \frac{H(1)}{x-1} + \frac{H(-1)}{x+1} + \frac{H(\infty)(t-1)(t+1)}{x(x-1)(x+1)} \cdot (t-x).
\]

Here

\[
H(0) = F_0 \cdot G_0, \\
H(1) = (F_0 + F_1 + F_2) \cdot (G_0 + G_1 + G_2), \\
H(-1) = (F_0 - F_1 + F_2) \cdot (G_0 - G_1 + G_2), \\
H(x) = (F_0 + F_1 x + F_2 x^2) \cdot (G_0 + G_1 x + G_2 x^2), \text{ and} \\
H(\infty) = F_2 \cdot G_2
\]

are the only five polynomial multiplications in the algorithm. These polynomials expand from \( n \) to \( 2n \) terms, except \( H(x) \). \( H \) simplifies to

\[
H = U + (H(1) - H(-1)) \cdot t \\
- \frac{H(0) + (H(1) + H(-1)) + H(\infty)}{x} \cdot t^2 + \frac{U \cdot t^3 + H(\infty) \cdot t^4}{x^2 - 1} \tag{1}
\]

where

\[
U = V + \frac{H(0)}{x} - H(\infty) \cdot x
\]

and

\[
V = \frac{(H(1) + H(-1)) \cdot x + (H(1) - H(-1)) + H(x)}{x^2 - 1}.
\]

There are two tricky issues while computing \( V \). First, \( \deg(H(x)) \leq 2n + 2 \), introducing extra complexity since all other polynomials have degree less than \( 2n \). By requiring \( \deg(F_2) \leq n - 2 \) and \( \deg(G_2) \leq n - 2 \), we force \( \deg(H(x)) \leq 2n \). Since \( H(x) \) is only used as \( H(x)/x \) in \( V \), we can always process polynomials with degree less than \( 2n \).

The other issue concerns computing divisions by \( x^2 - 1 \) in \( \mathbb{Z}/3[\mathbb{x}] \). Since long division is a sequential process and not efficient in SIMD settings, we now present a divide-and-conquer method for it.

#### 3.2.3 Division by \( x^2 - 1 \) on \( \mathbb{Z}/3[\mathbb{x}] \)

Dividing a polynomial \( f \) by \( x^2 - 1 \) means producing a representation of \( f = q \cdot (x^2 - 1) + r \), where \( q \) and \( r \) are the quotient and remainder, respectively. Assume that we have recursively divided two \( 2m \)-coefficient polynomials \( f \) and \( g \) by \( x^2 - 1 \), obtaining \( f = q \cdot (x^2 - 1) + r \) and \( g = s \cdot (x^2 - 1) + t \). Then

\[
r \cdot 2^{m} = (r x^{2m-2} + r x^{2m-4} + r x^{2m-6} + \cdots + r)(x^2 - 1) + r
\]

so the result of dividing \( f \cdot 2^{m} + g \) by \( x^2 - 1 \) is

\[
f \cdot 2^{m} + g = [q \cdot 2^{m} + r \cdot 2^{m-2}](x^2 - 1) + (s + r x^{2m-4} + \cdots + r)(x^2 - 1)(t + r) \tag{2}
\]

We carry out these divisions in place as follows: recursively overwrite the array of \( f \) coefficients with \( q \) and \( r \), recursively overwrite the array of \( g \) coefficients with \( s \) and \( t \), and then simply add the lowest two coefficients from the \( f \) array into each coefficient pair in the \( g \) array.

Because the recursive computations for \( f \) and \( g \) are independent, this computation parallelizes. The overall parallel computation for dividing a length-\( n \) polynomial by \( x^2 - 1 \), assuming \( n = 2^i \), proceeds as follows. The computation comprises \( l \) steps. The first step splits the polynomial into \( n/4 \) separate sub-polynomials; each sub-polynomial has degree less than four. We divide a length-four sub-polynomial by \( x^2 - 1 \) by adding two coefficients of higher degrees to the lower two coefficients. We perform these divisions in parallel. In each subsequent step, we double the sub-polynomial sizes, and divide sub-polynomials by \( x^2 - 1 \) by adding two coefficients of lower degree from the higher degree parts to the lower parts of the polynomials as in Equation 2. Since each step performs \( n/2 \) additions, the whole computation costs \( n (\log_2(n) - 1)/2 \) additions.

#### 3.2.4 AVX2 optimization for the \( \mathcal{R}/3 \) multiplier

Since we use integer arithmetic for \( \mathbb{Z}/3 \) and integers grow, we must control the values to prevent overflow. From the AVX2 instruction set, we use the \texttt{vpshufb} instruction to reduce the values. The instruction reads the lower nibbles as indexes from single-byte lanes of a register, then replaces the lane values with those from a 16-entry table, using the four-bit indexes. Thus, we use \texttt{vpshufb} to reduce integers in \( [0, 16) \) to integers in \( [0, 3) \). We also reduce adjacent nibbles by moving them to lower positions using bit-shift instructions.

Our software for \( 16 \times 16 \) polynomial multiplication actually performs two independent \( 16 \times 16 \) multiplications in the
two xmm lanes of ymm registers, respectively. The approach avoids the high latency for moving data between different xmm lanes in Haswell CPUs (see [23, p. 237] for the vperm2128, vextracti128, and vinserti128 instructions). Specifically, our AVX2 multiplier takes two ymm registers as input and outputs products in two ymm registers. A ymm register comprises two polynomials \((a, c)\) where \(a, c \in (\mathbb{Z}/3)[x]\) are stored in different xmm lanes. Given two ymm inputs \((a, c)\) and \((b, d)\), the multiplier outputs \((ab_1 + ab_2 \cdot x^{16}, cd_1 + cd_2 \cdot x^{16})\) in two ymm registers, where \(a \cdot b = ab_1 + ab_2 \cdot x^{16}\) and \(c \cdot d = cd_1 + cd_2 \cdot x^{16}\). Thus, we avoid the data exchange between xmm lanes.

3.3 Polynomial multiplication in \((\mathbb{Z}/q)[x]\)

3.3.1 Problem description and related multiplication

While applying NTT-based multiplication, NTRU Prime faces two issues. First, recalling Section 2.1, NTRU Prime works on the polynomial ring \(\mathcal{R} = (\mathbb{Z}/q)[x]/(x^p - x - 1)\) where \(x^p - x - 1\) is irreducible in \((\mathbb{Z}/q)[x]\); hence, there is no way to apply FFT tricks on the ring. The standard workaround is to lift ring elements in \(\mathcal{R}\) to \((\mathbb{Z}/q)[x]\), and multiply the lifted polynomials with an NTT-based multiplication in \((\mathbb{Z}/q)[x]/(x^N - 1)\) where \(N \geq 2p\). Since two input polynomials have degree less than \(p\), their product will not overflow the degree \(N\). After the polynomial multiplication, the product is reduced with a division by \(x^p - x - 1\) for the result in \(\mathcal{R}/q\).

Secondly, \(q\) from the NTRU Prime parameter set is not a radix-2 NTT friendly prime. For example, \(q = 4591\) in sntrup761, and since \(4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17\), no simple root of unity is available for recursive radix-2 FFT tricks. Alkim, Cheng, Chung, Evkan, Huang, Hwang, Li, Niederhagen, Shih, Waldé, and Yang [3] presented a non-radix-2 NTT implementation on \((\mathbb{Z}/4591)[x]/(x^{530} - 1)\) for embedded systems. They performed radix-3, radix-5, and radix-17 NTT stages in their NTT. We instead use a radix-2 algorithm that efficiently utilizes the full ymm registers in the Haswell architecture.

The fastest Haswell sntrup software before our work dealt with the radix-2-NTT-unfriendly \(q\) by lifting the coefficients to \(\mathbb{Z}\) and then multiplying in \((\mathbb{Z}/7681)[x]\) and \((\mathbb{Z}/10753)[x]\). Both 7681 and 10753 are NTT-friendly. This suffices for "big×small" multiplications for all specified NTRU Prime parameters: one input is a small element of \(\mathcal{R}/q\), with coefficients in \([-1, 0, 1]\), and the maximum coefficient of a "big×small" product is below 7681·10753/2 in absolute value.

However, Montgomery’s trick involves general "big×big" multiplications in \(\mathcal{R}/q\). Even if each coefficient for, e.g., \(q = 4591\) is fully reduced to the range \([-2295, 2295]\), the product here can have coefficients as large as \(2295 \cdot 2295 \cdot 761 > 7681 \cdot 10753\). One way to handle these multiplications would be to use more NTT-based multiplications over small moduli, for example multiplying in \((\mathbb{Z}/7681)[x]\) and \((\mathbb{Z}/10753)[x]\) and \((\mathbb{Z}/12289)[x]\), but this means 50% more NTTs, plus extra reductions since 12289 is larger than 10753. We take a different approach described below.

3.3.2 Our polynomial multiplication

In this section, we present a multiplication for polynomials in \((\mathbb{Z}/q)[x]\) with degree less than 1024. We first map polynomials to \((\mathbb{Z}/q)[x]/(x^{2048} - 1)\). Rather than switching from \(q\) to an NTT-friendly prime, we use Schönhage’s trick (Section 2.3) to manufacture roots of unity for radix-2 NTTs.

Specifically, define \(K\) as the ring \((\mathbb{Z}/q)[x]/(x^{64} + 1)\). We map \((\mathbb{Z}/q)[x]/(x^{2048} - 1)\) to \((\mathbb{Z}/q)[y]/(y^{64} - 1))\(x/(x^{32} - y)\), lift to \((\mathbb{Z}/q)[x][y]/(y^{64} - 1)\), and then map to \(K[y]/(y^{64} - 1)\). Each 32 consecutive terms of a polynomial in \((\mathbb{Z}/q)[x]\) are thus viewed as an element of \(K\). We segment the original polynomial of 1024 terms in \(x\) into 32 elements in \(K\), associating each element in \(K\) to a new indeterminate \(y\) with different degrees. The remaining problem is to multiply elements of the ring \(K[y]/(y^{64} - 1)\).

We use NTTs to multiply in \(K[y]/(y^{64} - 1)\), using \(x\) as a primitive 128-th root of unity in \(K\). NTT-based multiplication applies two NTTs for the input polynomials, performs component-wise multiplication for the transformed values, and applies one inverse NTT for the final product. Each NTT converts one input element in \(K[y]/(y^{64} - 1)\) into 64 elements in \(K\), using additions, subtractions, and multiplications by powers of \(x\). Multiplication by a power of \(x\) simply raises the degree of the polynomial in \((\mathbb{Z}/q)[x]\), and then replaces \(x^{64+i}\) by \(-x^i\), using negations without any multiplications in \(\mathbb{Z}/q\).

After transforming the input polynomials into a list of elements in \(K\), we perform the component-wise multiplication for the transformed vectors. The problem now is to multiply two elements of \(K = (\mathbb{Z}/q)[x]/(x^{64} + 1)\).

We use Nussbaumer’s trick (Section 2.3) to manufacture further roots of unity: map \(K\) to \(((\mathbb{Z}/q)[y]/(y^{64} + 1))[x]/(x^8 - y), lift to \(((\mathbb{Z}/q)[y]/(y^{64} + 1))[x]/(x^8 + 1))\)\(x/(x^{64} - 1)\), and map to \(((\mathbb{Z}/q)[y]/(y^{64} + 1))[x]/(x^{64} - 1)\). The polynomial ring \((\mathbb{Z}/q)[y]/(y^{64} + 1)\) supports a radix-2 NTT of size 16 with a primitive root of unity \(y\).

Since the polynomials are short, we choose Karatsuba’s algorithm for component-wise multiplication in \((\mathbb{Z}/q)[y]/(y^{64} + 1)\). We use Montgomery multiplication [33] to calculate modular products in \(\mathbb{Z}/q\).

For sntrup761 and sntrup653, the input polynomials have degree less than 768, so we truncate some computations in the NTT algorithm: we apply NTT on the ring \(K[y]/((y^{32} + 1)(y^{16} - 1))\) instead of the original \(K[y]/(y^{64} - 1)\). We map the input polynomials to degree-24 polynomials in \(K[y]\), and calculate the product with a truncated inverse NTT of 48 values. Our NTT sizes are within 18%, 1%, and 20% of optimal for 653, 761, and 857 respectively; further truncation is possible at the expense of some complication in the calculations.
3.3.3 AVX2 optimization for the $\mathcal{R}/q$ multiplier

Since the component-wise multiplication step comprises 48 or 64 multiplications on $K$, we perform the multiplications simultaneously in different 16-bit lanes ofymm registers. Our software stores the first $\mathbb{Z}/q$ coefficient of 16 elements in $K$ in a ymm register, stores their second coefficients in a second register, and so on. In this way, we avoid data movement between the 16-bit lanes inside a ymm register.

To apply this optimization, we first rearrange the coefficients of a polynomial to different registers with a $16 \times 16$ matrix transposes. Given sixteen degree-15 polynomials $(a_0^{(0)} + a_1^{(0)} x + \cdots + a_{15}^{(0)} x^{15}), \ldots, (a_0^{(15)} + \cdots + a_{15}^{(15)} x^{15})$, data in (...) represents one ymm register, and we treat a polynomial in one ymm register as a row of a $16 \times 16$ matrix. Transposing this matrix rearranges the data to $(a_0^{(0)}, \ldots, a_0^{(15)}), \ldots, (a_{15}^{(0)}, \ldots, a_{15}^{(15)})$. Thus, we can fetch a specific coefficient by accessing its corresponding ymm register, while parallelizing 16 polynomial multiplications for the transposed data.

We use the method in [46] for matrix transposition. The technique transposes a $2 \times 2$ matrix by swapping its two off-diagonal components. For transposing matrices with larger dimensions, e.g. $4 \times 4$, it first swaps data between two $2 \times 2$ off-diagonal sub-matrices, and then performs matrix transpose for all its four sub-matrices.

3.4 Microbenchmarks: arithmetic

We benchmark our implementation on an Intel Xeon E3-1275 v3 (Haswell), running at 3.5 GHz, with Turbo Boost disabled. The numbers reported in this section are medians of 3 to 63 measurements, depending on the latency of the operation under measurement. We omit the numbers of multiplication for sntrup653 because it actually uses the same multiplier as sntrup761.

3.4.1 Benchmarks for $\mathcal{R}/3$

We compare the cycle counts for $\mathcal{R}/3$ multiplication between our implementation and the best previous sntrup implementation, round2 in [11], in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Implementation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>sntrup761</td>
<td>this work (Section 3.2)</td>
<td>8183</td>
</tr>
<tr>
<td></td>
<td>this work (Section 3.2, Appendix C)</td>
<td>8827</td>
</tr>
<tr>
<td></td>
<td>NTRUP round2 (NTT, [11])</td>
<td>9290</td>
</tr>
<tr>
<td>sntrup857</td>
<td>this work (Section 3.2)</td>
<td>12840</td>
</tr>
<tr>
<td></td>
<td>this work (Section 3.2, Appendix C)</td>
<td>12533</td>
</tr>
<tr>
<td></td>
<td>NTRUP round2 (NTT, [11])</td>
<td>12887</td>
</tr>
</tbody>
</table>

The best results are from our Karatsuba-based polynomial multiplication for smaller parameters, and from our NTT improvements for larger parameters.

We can see the cost of three multiplications and one invertibility check is less than half of a single inversion in $\mathcal{R}/3$. It is clear that batch inversion costs less than pure ring inversion, even for the smallest possible batch size of two.

3.4.2 Benchmarks for $\mathcal{R}/q$

The following table shows the cycle counts of big $\times$ big multiplication and big $\times$ small multiplication in $\mathcal{R}/q$, comparing with the previous best software [11].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Implementation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>sntrup761</td>
<td>this work (Section 3.3, big $\times$ big)</td>
<td>25113</td>
</tr>
<tr>
<td></td>
<td>this work (Appendix C, big $\times$ small)</td>
<td>16992</td>
</tr>
<tr>
<td></td>
<td>NTRUP round2 [11], big $\times$ small</td>
<td>18080</td>
</tr>
<tr>
<td>sntrup857</td>
<td>this work (Section 3.3, big $\times$ big)</td>
<td>32265</td>
</tr>
<tr>
<td></td>
<td>this work (Appendix C, big $\times$ small)</td>
<td>24667</td>
</tr>
<tr>
<td></td>
<td>NTRUP round2 [11], big $\times$ small</td>
<td>25846</td>
</tr>
</tbody>
</table>

The results show the absolute cycle count of big $\times$ big is larger than big $\times$ small multiplication. To evaluate the efficiency of big $\times$ big multiplication, consider if we extend the big $\times$ small multiplication to big $\times$ big multiplication, by applying more internal NTT multiplications. It will result in multiplications of roughly $3/2$ times the current cycle counts, i.e., slower than big $\times$ big multiplication presented in this work.

Since big $\times$ small multiplication is faster than big $\times$ big, we use the former as much as possible in batchinv for $\mathcal{R}/q$. Recall that Montgomery’s trick for batch inversion replaces one inversion in $\mathcal{R}/q$ by roughly three ring multiplications and one amortized ring inversion. From the batchinv algorithm in Section 2.2, we can see the three ring multiplications are $a_i \cdot b_{i-1}, a_i \cdot t_i,$ and $t_i \cdot b_{i-1}$. Since the input $a_i$ is a small element, it turns out that only the last is big $\times$ big multiplication. Since the costs for inverting one element in $\mathcal{R}/q$ are 576989, 785909, and 973318 cycles for sntrup653, sntrup761, and sntrup857, respectively, the cost of two big $\times$ small and one big $\times$ big multiplication is clearly much less than one inversion operation.
3.4.3 Benchmarks for batch key generation

We show the benchmark results for batch key generation (BatchKeyGen) in Figure 1. See also Table 2.

The figure shows how increasing \( n \), the key generation batch size, amortizes the ring inversion cost. Generating a few dozen keys at once already produces most of the throughput benefit: for example, generating \( n = 32 \) keys takes a total of 1.4 milliseconds for sntrup761 at 3.5GHz. Generating \( n = 128 \) keys takes a total of 5.2 milliseconds for sntrup761 at 3.5GHz, about 10% better throughput than \( n = 32 \).

We adopt BatchKeyGen with batch size \( n = 32 \) in our library, resulting in 156317 Haswell cycles per key.

4 New TLS software layering

At the application level, the goals of our end-to-end experiment are to demonstrate how the new results can be deployed in real-world conditions, transparently for the end users, and meet the performance constraints of ubiquitous systems. For this reason, we developed patches for OpenSSL 1.1.1 to support post-quantum key exchange for TLS 1.3 connections. We designed our patches so that any existing application built on top of OpenSSL 1.1.1 can transparently benefit from the PQC enhancements with no changes, as the patched version of OpenSSL retains API/ABI compatibility with the original version and acts as a drop-in replacement. This works for any application dynamically linking against libssl as the backend to establish TLS 1.3 connections. Among them, for our demonstration, we picked a web browser\(^1\), a custom application (tls_timer, described later), and a TLS proxy.\(^2\) After installing our patched version of OpenSSL, users can establish secure and fast post-quantum TLS connections.

Appendix G provides relevant technical background regarding the OpenSSL software architecture. The rest of this section describes, with more detail, our work to achieve the goals of our experiment, and provides rationale for the most relevant design decisions.

Figure 2 depicts a high-level overview of our end-to-end experiment, highlighting the boundary between the unmodified software ecosystem and our novel contributions. This section details, in particular, our OpenSSL patches and our new ENGINE component. libsntrup761 and libsntrup857 provide the new optimized implementations of sntrup761 and sntrup857 operations (Section 3), through a simple standardized API that is independent from OpenSSL, and reusable by other cryptographic software components.

---

\(^1\)GNOME Web (a.k.a. epiphany)— https://wiki.gnome.org/Apps/Web

\(^2\)stunnel — https://www.stunnel.org/
4.1 OpenSSL patches

Figure 4 depicts an architecture diagram of our end-to-end experiment, highlighting with red boxes inside the libcrypto and libssl modules, the patched OpenSSL components.

**libssl changes.** Within libssl, conceptually three elements need to be changed:

- Modify the server-side handling of the *Key Share* extension in an outgoing *ServerHello* to optionally use a KEM *Encapsulate()* operation for KEM groups;
- Modify the client-side handling of the *Key Share* extension in an incoming *ServerHello* to optionally use a KEM *Decapsulate()* operation for KEM groups;
- Hardcode private NamedGroup TLS 1.3 codepoints to negotiate sntrup761 or sntrup857 groups for key exchange.

As OpenSSL 1.1.1 does not provide an abstraction for KEM primitives, we implemented the first two changes as a workaround, to which we refer as PKE+NIKE. It maps the KEM operations as a combination of *public-key encryption* (PKE) and *non-interactive key exchange* (NIKE).

3 We combine the use of *EVP_PKEY_encrypt()* with NULL input, followed by *EVP_PKEY_derive()* to mimic *Encapsulate()* and *EVP_PKEY_decrypt()* with NULL output, followed by *EVP_PKEY_derive()* for *Decapsulate()*.

Due to the structure of the PKE+NIKE workaround, on both sides of the handshake, handling the *Key Share* extension for KEM groups finishes with the call for *EVP_PKEY_derive()* before updating the protocol key schedule. This is also the case in the original code that supports traditional NIKE. Therefore, the new code only affects the handling of the opaque *Key Share* content transmitted over the wire.

On the server side, traditional NIKE groups generate an ephemeral keypair, sending the encoded public key as the payload of the extension. With our patch, if the group is flagged as a KEM group, instead of key generation we execute *EVP_PKEY_encrypt()* under the client’s public key with NULL input. We then send the resulting ciphertext as the payload of the *Key Share* extension. As a side effect, per our PKE+NIKE workaround, *EVP_PKEY_encrypt()* also stores the shared secret plaintext within the internal state of the server-side object representing the client’s public key. This plaintext is what is ultimately retrieved upon calling *EVP_PKEY_derive()*.

On the client side, for traditional NIKE groups, the payload of the *Key Share* extensions is parsed as the encoding of the peer’s public key, to be used in the subsequent *EVP_PKEY_derive()*.

3Generally speaking, the *EVP* API supports any NIKE algorithm. But historically, DH and ECDH have been the only implementations included in OpenSSL for this API. Hence, code and documentation tend to refer to such primitives as *DH key exchange* or just *key exchange* rather than NIKE.

4.2 The engNTRU ENGINE

As mentioned in Section 2.5 and depicted in Figure 2 and Figure 4, as part of our end-to-end experiment, we introduce a new ENGINE, dubbed engNTRU.

We followed the methodology suggested in [45], and we defer to it for a detailed description of the ENGINE framework, how it integrates with the OpenSSL architecture (partially illustrated in Figure 4) and general motivations to use...
the ENGINE framework for applied research. In this section, we highlight how this choice has two main benefits. First, it decouples OpenSSL from fast-paced development in the ecosystem of optimized implementations for post-quantum primitives. Finally, at the same time it decouples external libraries implementing novel primitives from the data types and patterns required to provide OpenSSL compatibility.

engNTRU builds upon libbecc [16], which is itself derived from libsuola [45]. Similar to both previous works, engNTRU is also a shallow ENGINE, i.e., it does not contain actual cryptographic implementations for the supported primitives. Instead, it delegates actual computations to libsntrup761 and libsntrup857. The functionality provided by engNTRU includes:

- building as a dynamically loadable module, injecting support for novel cryptographic primitives transparently for existing applications;
- supporting generic KEM primitives under the PKE+NIKE workaround;
- dynamically injecting/replacing support for sntrup761 at run-time, delegating to libsntrup761 for optimized computation;
- dynamically injecting support for sntrup857 at run-time, delegating to libsntrup857 for optimized computation;
- mapping the PKE+NIKE workaround back to the standard KEM API adopted by the implementations of NIST PQC KEM candidates, including libsntrup*.

Furthermore, similar to libbecc and libsuola, and using the same terminology, engNTRU supports the notion of multi-threads to interface with the OpenSSL API. Under the serial_lib provider, each Keygen() operation is mapped to crypto_kem_keypair(), generating a new keypair on demand as defined by the NIST PQC KEM API. Alternatively, under the batch_lib provider (which is the default in our experiment), engNTRU supports batch key generation, similar to libbecc. In the case of libsntrup761 and libsntrup857, this allows OpenSSL and applications to transparently take advantage of the performance gains described in Section 3.

Under the batch_lib model, while a process is running, each sntrup* parameter set is associated with a thread-safe heap-allocated pool of keypairs. Every time an application thread requests a new sntrup* keypair, engNTRU attempts to retrieve a fresh keypair from the corresponding pool. For each supported parameter set, it dynamically allocates a pool, initialized the first time a keypair is requested. This includes filling the pool, by calling crypto_kem_sntrup761_keypair_batch() or crypto_kem_sntrup857_keypair_batch(). Otherwise, after the first request, engNTRU normally serves each request by copying (and then securely erasing from the pool buffer) the next fresh entry in the pool. After this, if the consumed keypair was the last in the pool, engNTRU fills it again, by calling the corresponding libsntrup* batch generation function. This happens synchronously, before returning control to the application.

It is easy to see the advantage of batch_lib over serial_lib from our microbenchmarks in Section 3. With serial_lib, each sntrup761 key costs 0.4ms on a 2GHz Haswell core. With batch_lib, within each batch of 32 sntrup761 keys, the first key costs 2.5ms, and the remaining 31 keys each cost 0ms. Note that, according to video-game designers [17], latencies below 20ms are imperceptible. A series of K sntrup761 keys costs 0.4Kms from serial_lib and just (0.08K + 2.5)ms from batch_lib. Similar comments apply to the separate sntrup857 pool.

As long as API/ABI compatibility is maintained in the engNTRU/libsntrup* interfaces, further refinements in the libsntrup* implementations do not require recompiling and reinstalling engNTRU, or OpenSSL, nor other components of the software ecosystem above. At the same time, libsntrup761 and libsntrup857 are isolated from OpenSSL-specific APIs, so they can easily be reused by alternative stacks supporting the NIST PQC KEM API. Moreover, they can retain a lean and portable API, while details like the handling of pools of batch results, or the sharing model to adopt, are delegated to the middleware layer.

4.3 Reaching applications transparently

Consulting Figure 4, the purpose of this section is to describe the extent of the application layer we explored in our study. In these experiments, we investigated two paths to reach libssl and libcrypto (and subsequently engNTRU then libsntrup*). Namely, a networking application dynamically linking directly, and a separate shared library against which even higher level applications dynamically link against. More generally, this approach works for any application which supports TLS 1.3 by dynamically linking against libssl1.1.1, but not for statically linked applications.\(^5\)

**stunnel.** For networking applications that do not natively support TLS, stunnel is an application that provides TLS tunneling. The two most common deployment scenarios for stunnel are client mode and server mode.

In client mode, stunnel listens for cleartext TCP connections, then initiates a TLS session to a fixed server address. A common use case for client mode would be connecting to a fixed SMTPS server to securely transfer email.

In client mode, stunnel listens for cleartext TCP connections, then initiates a TLS session to a fixed server address. A common use case for client mode would be connecting to a fixed SMTPS server to securely transfer email.\(^5\)

\(^5\)Although not part of our end-to-end demo described here, we further validated this by successfully enabling sntrup connections in popular web servers, such as nginx and Apache httpd, and other applications, without changes to their sources or their binary distributions.
In server mode, stunnel listens for TLS connections, then initiates cleartext TCP connections to a fixed server address. A common use case for server mode would be providing a TLS service from a server application that does not support TLS. For example, a user could serve a single static webpage over HTTP with the `netcat` utility, which stunnel would then wrap to serve the content via HTTPS to incoming connections from e.g. browsers. In this light, stunnel server mode is one form of TLS termination.

stunnel links directly to OpenSSL for TLS functionality, hence the intersection with ngNTRU and underlying libsntrup* is immediate. For example, in stunnel server mode, this requires no changes to the server application, which in fact is oblivious to the TLS tunneling altogether.

**glib-networking.** Similar to how the Standard Template Library (STL) and Boost provide expanded functionality for C++ (e.g. data structures, multithreading), Glib is a core C library for GNOME and GTK applications. Bundled as part of Glib, one feature of the Gnome Input/Output (GIO) C library provides an API for networking functionality, including low-level BSD-style sockets. For TLS connections, GIO loads the glib-networking C library, which abstracts away the backend TLS provider, and presents a unified interface to callers. Currently, glib-networking supports two such backends: GnuTLS and OpenSSL. The latter is newer, main-lined in v2.59.90 (Feb 2019) while the current version as of this writing is v2.68.1. This is precisely the place where glib-networking intersects OpenSSL.

To summarize, the modularity of glib-networking regarding TLS backends, coupled with the layered approach of GIO, allows any application utilizing glib-networking for TLS functionality to transparently benefit from ENGINE features, including engNTRU.

One such application, and one highlight of our experiments, is GNOME Web. Neither Google Chrome nor Mozilla Firefox are capable of this level of modularity. Both browsers link directly to TLS backends at build time (BoringSSL, NSS). These do not support dynamically injecting this level of cryptosystem functionality, necessarily extending to the TLS layer as well. In general, all other popular browser implementations (we are aware of) require source-code changes to add any new TLS cipher suite. In our experiments, we are able to make GNOME Web sntrup761- and sntrup857-aware with absolutely no changes to its source code, nor that of glib-networking. Performance-wise, GNOME Web then transparently benefits from the batch key generation in libsntrup* through engNTRU, loaded dynamically by the OpenSSL TLS backend of glib-networking.

### 4.4 Macrobenchmarks: TLS handshakes

To conclude our end-to-end experiment, we investigated the impact of enabling post-quantum key exchanges for TLS 1.3 handshakes, as perceived by end users. We considered an experiment on large-scale deployments like CECPQ1 or CECPQ2 out of scope for this work, as it would be better served by a dedicated study. As an alternative, we decided to evaluate the performance on a smaller and more controlled environment: namely, a client and a server connected over a low-traffic Gigabit Ethernet network. We chose to focus on number of connections per second as the more relevant metric from the point of view of end users, and used easily accessible consumer hardware as the platform, to simulate a small office setup.\(^6\)

To exercise full control over the sampling process, we developed a small (about 300 LOC) TLS client built directly on top of libssl (see Appendix F for a discussion about in-browser benchmarks). Referring to the diagram in Figure 2, the end-to-end benchmark replaces epiphany with this new program, that we dubbed tls_timer. In its main loop, tls_timer records a timestamp, sequentially performs a predetermined number of TLS connections, then records a second timestamp, returning the elapsed wall-clock time. In the above loop, for each connection, it performs a full TLS 1.3 handshake. Then, the client properly shuts down the connection, without sending any application data. Hence, the total elapsed time measured by the client covers the computation time required by client and server to generate and parse the handshake.

![Figure 3: Cumulative distributions of handshake performance under different cryptosystems in a local network. Each curve represents a key-exchange group, for which we collected 100 samples, in terms of average number of connections per second. This metric is extrapolated from measuring the elapsed wall-clock time over 8192 sequentially established connections per sample.](image)

\(^6\)The client side is hosted on an Intel Core i7-6700 workstation, running Ubuntu 20.04.2 with Linux 5.4.0, while the server side is hosted on an AMD Ryzen 7 2700X workstation, running Ubuntu 18.04.5 with Linux 5.4.0. Both peers directly connect to the same Gigabit Ethernet L2 switch via their embedded Gigabit Ethernet NICs.
content of the exchanged messages. It also includes the time spent due to transit of packets over the network, and through userland/kernel space transitions. In particular, with respect to cryptographic computations, during the benchmark the client repeatedly performs `Keygen()` and `Decapsulate()` for the ephemeral key exchange, and RSA-2048 signature verifications to validate the identity of the server against its certificate. During the client-measured interval, the server respectively performs `Encapsulate()` for the ephemeral key exchange, and RSA-2048 signature generation for authentication.

As a baseline for comparisons, we used `tls_timer` to analogously measure the performance of TLS handshakes using the most popular TLS 1.3 groups for key exchange: namely, X25519 and P-256, in their respective ASM-optimized implementations. These are the fastest software implementations of TLS 1.3 key-exchange groups shipped in OpenSSL 1.1.1k, and are widely deployed in production. For these groups, computation on the client and server differs from the description above exclusively on the ephemeral key exchange, as both sides perform their respective NIKE `Keygen()` and `Derive()` operations instead of the listed post-quantum KEM operations.

On the server side `tls_timer` connects to an instance of `stunnel`, configured as described above. Technically `stunnel` is itself connected to an `apache2` HTTP daemon serving static content on the same host, but as `tls_server` does not send any application data, the connection between `stunnel` and `apache2` is short-lived and does not carry data. Finally, to minimize noise in the measurements, we disabled frequency scaling and Turbo Boost on both platforms, terminated most concurrent services and processes on the client and the server, and isolated one physical core exclusively to each benchmark process (i.e., `tls_timer`, `stunnel`, and `apache2`) to avoid biases due to CPU contention.

Figure 3 visualizes our experimental results as cumulative distributions for each tested group. The results show that, in our implementation, both the recommended `sntrup761` parameter set and the higher security `sntrup857` consistently achieve more connections per second than the optimized implementations of pre-quantum alternatives currently deployed at large. The unloaded high-bandwidth network of our experimental environment masks the higher communication costs of the lattice cryptosystems, hence these results do not imply that `sntrup` cost less than ECC algorithms overall, as further discussed in Appendix D. However, our results show that, in terms of computational costs, we achieve new records when compared with the most performant implementations of TLS 1.3 key-exchange groups included in OpenSSL 1.1.1k, while providing higher pre-quantum security levels and much higher post-quantum security levels against all known attacks.

5 Conclusion

NIST’s ongoing Post-Quantum Cryptography Standardization Project poses significant challenges to the cryptology, applied cryptography, and system security research communities, to name a few. These challenges span both the academic and industry arenas. Our work contributes to solving these challenges in two main directions. (1) In Section 3, we propose software optimizations for `sntrup`, from fast SIMD arithmetic at the lowest level to efficient amortized batch key generation at the highest level. These are an essential part of our new `libsntrup761` and `libsntrup857` libraries. (2) In Section 4, we demonstrate how to realize these gains from `libsntrup` by developing `engNTRU`, a dynamically-loadable OpenSSL ENGINE. We transparently expose it to the application layer through a light fork of OpenSSL, augmented with `sntrup` support in TLS 1.3 cipher suites. Our experiments reach the Gnome Web (Epiphany) browser on the client side and `stunnel` as a TLS terminator on the server side—both with no source-code changes. Finally, our end-to-end macrobenchmarks combine (1) and (2) to achieve more TLS 1.3 handshakes per second than any software included in OpenSSL.

CECPQ1 and CECPQ2 were important proof-of-concept experiments regarding the integration of post-quantum algorithms into selected browser and TLS implementations, but those experiments suffered from poor reproducibility: the capabilities and telemetry are only available to major industry players like Google and Cloudflare, so the cryptographic primitive choice and optimization techniques were dictated by them as well. Our work demonstrates that establishing a research environment to provide reproducible results is not only feasible, but achievable with a reasonable workload distribution, using new TLS software-layering techniques to minimize complexity at the architecture and system levels.

Availability. In support of Open Science, we provide several free and open-source software (FOSS) contributions and research artifacts.

We released `libsntrup761`, `libsntrup857`, `engNTRU`, and `tls_timer` as FOSS. We also contributed our FOSS implementations of enc and dec to SUPERCOP; its API does not support batch keygen at this time. Lastly, we published our OpenSSL patches and a detailed, step-by-step tutorial to reproduce our full experiment stack.

Acknowledgments. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy—EXC 2092 CASA—390781972 “Cyber Security in the Age of Large-Scale Adversaries”, by the U.S. National Science Foundation under grant 1913167, by the Cisco University Research Program, and by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 804476). “Any opinions,
findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation” (or other funding agencies).

The tls_timer icon used in Figure 4 is licensed under CC-BY and created by Tomas Knopp for thenounproject.com. All product names, logos, brands and trademarks are property of their respective owners.

The scientific colour map batlow [21] is used in this study to prevent visual distortion of the data and exclusion of readers with colour-vision deficiencies [22].

**Metadata.** Permanent ID of this document: a8f6fc35a5dc11da1f125b9f5225d2a9e4c5b08b.

**References**


[28] Matthias Kannwischer, Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers. PQClean: clean,


[38] Steven Rich and Barton Gellman. NSA seeks to build quantum computer that could crack most types of encryption, 2014. https://tinyurl.com/3msrzpzhh.
A Post-quantum timelines

Given a large enough quantum computer, Shor’s algorithm will rapidly break 2048-bit RSA and 256-bit ECC, the primary algorithms used today for TLS key exchange. This appendix reviews current predictions regarding the timeframe for this event, and the consequences for TLS.

The thousands of logical qubits used in Shor’s algorithm are expected to need millions of physical qubits [24]. This might sound like science fiction, compared to the 53 physical qubits underlying Google’s announcement of “quantum supremacy” in late 2019. However, IBM released a quantum-computing roadmap in September 2020 more than doubling the number of physical qubits per year, and “says it is confident it will make its 2030 deadline for a 1 million-qubit machine” [26].

Google makes similar projections regarding its own progress, according to [26]. Meanwhile, quantum algorithms are continuing to improve: for example, [24] reduced the number of quantum operations by two orders of magnitude in Shor’s algorithm for breaking RSA-2048, and [27] did the same for 256-bit ECC.

Several years ago, leaks showed that NSA was already intercepting “nearly everything a user does on the internet” [25], that the data-minimization procedures imposed on NSA by a secret court had an exception allowing ciphertexts to be stored indefinitely [20], and that NSA already had an $80 million/year internal budget for its “Penetrating Hard Targets” program, with a goal of building a “cryptologically useful quantum computer” [38]. Large-scale adversaries in various countries will be able to use future quantum computers to retroactively break the confidentiality of today’s recorded ciphertexts, and could be years ahead of IBM and Google in building those quantum computers.

If some messages encrypted by TLS need X years of confidentiality, then it is important for TLS to have completed its upgrade to post-quantum encryption X years before any attackers have quantum computers. TLS software encrypting a user’s message does not know how many years of confidentiality are required for that message, so it is important for the switch to be something that the software can afford for all messages.

It is also important for TLS to upgrade to post-quantum signatures before attackers have quantum computers, but this does not raise the same X-year issue. This paper focuses on encryption.

B Algorithm-selection issues

B.1 Speed

NIST says that performance plays a “large role” in its PQC standardization project, as noted in Section 1. Submitters have already put intensive efforts into optimizing every level of their algorithms; see, e.g., the submission documents for ntruhrss701 [19] and sntrup761 [13], and the software in benchmarking frameworks such as SUPERCOP [11] and pqm4 [29]. This existing work already includes CPU-specific optimizations, notably vectorization, which is essential for performance on modern Intel CPUs. NIST specified the Intel Haswell CPU microarchitecture as its first platform for comparisons, and this has been the most common optimization target in the literature, including extensive use of Haswell’s AVX2 vector instructions in software releases for ntruhrss701 and sntrup761. For comparability, this paper also focuses on Haswell.

There are strong arguments that CPU cycles are more important on smaller CPUs, such as the Cortex-A7 CPUs commonly used in low-end smartphones, or the Cortex-M4 microcontrollers commonly used in embedded devices. It is reasonable to extrapolate from this paper’s large sntrup761 speedups on Haswell to large sntrup761 speedups on other CPUs. However, the exact comparison between sntrup761 and ntruhrss701 will depend on the CPU: the cryptosystems use different polynomial rings, allowing different spaces of multiplication algorithms, which in turn interact with differences in instruction sets and in instruction speeds. For example, the sntrup761 speed records in pqm4 are from [3]; the multiplier in [3] uses non-power-of-2 NTTs, so it would be challenging to adapt to a vectorized CPU, and it relies on the prime modulus used in sntrup761, so it would not work for the power-of-2 modulus in ntruhrss701.

B.2 Beyond speed

Security levels. Each post-quantum proposal offers a selection of different key sizes, such as sntrup761 and sntrup857, where larger key sizes are less efficient but (hopefully) more secure. An easy way to reach the goal of “faster post-quantum TLS key exchange” is to take smaller keys—but this reduces the security level, perhaps to something unacceptable.

Sometimes the selection of key sizes is limited. For example, New Hope was structurally incapable of offering options between newhope512, which is at a questionable security level, and newhope1024, which used 1824 bytes for a public key plus 2048 bytes for a ciphertext. Part of the performance improvement from CECPQ1 to CECPQ2 came from the switch from New Hope to NTRU, which offers intermediate key sizes, such as ntruhrss701.

The performance improvements from CECPQ2’s ntruhrss701 to OpenSSL’s sntrup761 do not reduce security level; sntrup761 has higher security level than ntruhrss701. NIST uses a mechanism called “Core-SVP” to compare the security levels of lattice proposals; the algorithm sntrup761 selected here has Core-SVP 2153, while ntruhrss701 has Core-SVP just 2136. (For comparison, newhope1024 has Core-SVP 2257, and newhope512 has Core-SVP just 2112.) These are pre-quantum Core-SVP
levels; post-quantum Core-SVP has an exponent 10% smaller.

**Avoiding cyclotomic concerns.** NTRU Prime, the proposal that includes sntrup761, also avoids security concerns triggered by (1) the “cyclotomic” structure used in Kyber, NTRU, and SABER and (2) the quantum attack in [14] breaking the cyclotomic case of Gentry’s original lattice-based fully-homomorphic-encryption scheme. NIST has expressed “confidence in cyclotomic structures” [1], which is a controversial position: according to [13], 41% of the lattice submissions to the NIST project provided non-cyclotomic options, most of these 41% provided no cyclotomic options, and most of these 41% expressed security concerns regarding cyclotomic options.

**CCA security.** Part of the traffic in lattice-based encryption, and a larger part of the CPU time, comes from mechanisms designed to protect against “chosen-ciphertext attacks”. Some proposals save time by eliminating these mechanisms; this is safe if each new key is used for just one session, which, as noted above, is commonly recommended for TLS. However, TLS and other protocols have a long history of keys being deliberately or accidentally reused; for example, [18] reports that the Microsoft Windows TLS library, SChannel, “caches ephemeral keys for two hours”. CECPQ1 used newhope1024 without protection against chosen-ciphertext attacks, but CECPQ2 used ntruchrss701 with protection against chosen-ciphertext attacks. The rationale for this upgrade stated in [30] concluded that “CPA vs CCA security is a subtle and dangerous distinction, and if we’re going to invest in a post-quantum primitive, better it not be fragile.”

**Avoiding patents.** Another advantage of CECPQ2 over CECPQ1, an advantage shared by OpenSSLNTRU, is that there are no known patent threats against NTRU and sntrup, while there are two lines of patents threatening New Hope, Kyber, and SABER. The first line, with US patent 9094189 (expiring 2032) and corresponding international patents, covers the central idea of encryption by “noisy DH + reconciliation”. The second line, with US patent 9246675 (expiring 2033) and corresponding international patents, covers 2× ciphertext compression. Commentators sometimes claim that these patents are invalid and/or inapplicable, but the first patent was declared valid [36] by a European tribunal in response to litigation, and FOIA results [9] show that NIST has been attempting, so far unsuccessfully, to buy out that patent. Perhaps the patent threats will be resolved through buyouts or further litigation, but one cannot simply ignore the potential impact of patents on the deployment of post-quantum cryptography.

**C. Further improvements in NTRU Prime software**

This paper emphasizes a big speedup in sntrup key generation, and new software layers integrating this speedup into TLS software. The speedup relies on changing the key-generation API to generate many keys at once, and providing one key at a time. On top of this requires maintaining state, which is enabled by the new software layers.

This appendix describes other ways that we have improved the NTRU Prime software without changing the API. The software was already heavily tuned before our work, but some further streamlining turned out to be possible, for example reducing sntrup761 enc from 48892 cycles to 46914 cycles and reducing dec from 59404 cycles to 56241 cycles. More important than these quantitative speedups is the software engineering: we considerably simplified the preexisting optimized code base for NTRU Prime, especially with the new NTT compiler described below.

**C.1 Review of NTRU Prime options**


The round-3 NTRU Prime proposal expresses concern that dimension 512 in Kyber and SABER, with pre-quantum Core-SVP level (see Appendix B) below 2120, “will turn out to be inadequate against generic lattice attacks”. The proposal recommends 761 “for an extra security margin”. The proposal says that the objective of dimensions 953, 1013, and 1277 is “bulletproofing” to “prevent NIST from issuing security complaints”.

Users concerned about the risk of lattice attacks continuing to improve enough to break dimensions 653, 761, and 857 (also breaking kyber512 etc.) might be safe with dimension 1277. On the other hand, considering larger dimensions raises the question of whether the user can afford something other than a “small” lattice system. There are other post-quantum proposals that emphasize stability of security analysis as reducing risks.

NTRU Prime specifies two cryptosystems: Streamlined NTRU Prime (sntrup), an example of Quotient NTRU, and NTRU LPrime (ntrulpr), an example of Product NTRU. For example, dimension 761 has both sntrup761 and ntrulpr761. The two cryptosystems are almost identical in key sizes, ciphertext sizes, and Core-SVP security. The ntrulpr cryptosystems avoid the Quotient NTRU inversions and have much faster keygen than sntrup, but they have slower enc and slower dec than sntrup. They are also threatened by the same patents as Kyber and SABER; see Appendix B.

18
C.2 Review of the preexisting NTRU Prime software ecosystem

The official NTRU Prime software includes three software packages that run on Intel CPUs:

- A “reference implementation in Sage”. Sage is Python plus many math libraries. This implementation has the conciseness—and slowness—that one would expect from Python; it also has a warning that it leaks “secret information through timing”. This software supports dimensions 653, 761, 857, 953, 1013, and 1277.
- “Reference C software”. This software is in portable C and is designed to avoid timing attacks. Within these constraints, this software is written to maximize readability without regard to performance. The latest version of this software (see the [13] tarball) supports dimensions 653, 761, 857, 953, 1013, and 1277.
- “Optimized C software”. This software is not portable: it uses AVX2 instructions via `intrin.h` intrinsics. This software is designed to avoid timing attacks and to be as fast as possible. The latest version of this software supports only dimensions 653, 761, 857.

There have also been papers optimizing NTRU Prime for small devices, such as an ARM Cortex-M4 or a Xilinx Artix-7 FPGA.

The reference Sage implementation is a single file. It is mostly self-contained, but it relies on Sage for polynomial arithmetic, and it calls a few subroutines such as `hashlib.sha512`. The choice of parameter set—`sntrup761`, for example, or `ntrulpr857`—is made at run time. Most of the code is shared between `sntrup` and `ntrulpr`, but there are some differences: for example, `sntrup` key generation uses divisions, and `ntrulpr` uses AES.

The reference C implementation is mostly in a single file `kem.c` but includes a few general-purpose subroutines (Decode, Encode, and constant-time integer divisions) in separate files. The implementation also calls separate subroutines: SHA-512, for example. This implementation is conceptually more self-contained than the Sage implementation—for example, this implementation includes its own polynomial arithmetic (without timing leaks)—but is also less concise. The choice of parameter set is made by compile-time macros, such as `S12E761` and `SNTRUP`.

The optimized C implementation is structured differently. There are 18 subroutines factored out of the top-level `crypto_kem_sntrup761` and `crypto_kem_ntrulpr761`: for example, `crypto_core_multsntrup761` multiplies in the ring $\mathbb{R}/q$, and `crypto_encode_761x4591` encodes an element of $\mathbb{R}/q$ as a string. The SUPERCOP benchmarking framework automatically tries various compiler options for each subroutine and selects the fastest option, and automatically runs many test vectors for each subroutine.

Similar comments apply to dimensions 653 and 857, for 54 subroutines overall in the optimized C implementation. Some of the C code is shared across sizes except for compile-time selection of $q$ etc. Some of the C code is generated by Python scripts: for example, there is a Python script generating the `crypto_core_inv3sntrup*/avx` software for inversion in $\mathbb{R}/3$. (Beyond NTRU Prime, a variant of the same script is used in, e.g., `ntruhrss701`.) There is, however, less sharing of the multiplier code across sizes:

- Dimensions 653 and 761 use `mult768.c`, which uses size-512 NTTs to multiply 768-coefficient polynomials.
- Dimension 857 uses `mult1024.c`, which uses size-512 NTTs to multiply 1024-coefficient polynomials.

An underlying `ntt.c` is shared for computing size-512 NTTs, and the same NTT code is used for each of the NTT-friendly primes $r \in \{7681, 10753\}$, but multiplication algorithms vary between `mult768.c` and `mult1024.c`: for example, `mult768.c` uses “Good’s trick” to reduce a size-1536 NTT to 3 size-512 NTTs, taking advantage of 3 being odd, while `mult1024.c` uses a more complicated method to reduce a size-2048 NTT to 4 size-512 NTTs. The NTT API allows these 3 or 4 independent size-512 NTTs to be computed with one function call, reducing per-call overheads and also reducing the store-to-load-forwarding overheads in crossing NTT layers.

C.3 Improvements

We first built a tool to regenerate 653, 761, and 857 in the optimized C implementation from a merged code base. We then added support for 953, 1013, and 1277, which in previous work had only reference code. This meant, among other things, building a new `mult1280.c` to reduce a size-2560 NTT to 5 size-512 NTTs. Good’s trick is applicable here since 5 is odd, but we were faced with a new mini-optimization problem regarding the number of AVX2 instructions needed for 5-coefficient polynomial multiplications modulo $r$. The best solution we found uses 15 modular multiplications, 2 extra reductions, and 34 additions/subtractions.

We then built a new tool to compile concise descriptions of NTT strategies into optimized NTT software. This tool is analogous to SPIRAL [37], but handles the extra complications of NTTs compared to floating-point FFTs, notably the requirement of tracking ranges of intermediate quantities so as to avoid overflows. Note that one should not confuse automated generation of NTTs with automated generation of multipliers: it remains challenging to automate code generation for the type of multipliers that we consider in Section 3.

Armed with this tool, we searched for efficient size-512 NTT strategies to replace the previous `ntt.c`. We found a fully vectorizable strategy that

- avoids all overflows for both $r = 7681$ and $r = 10753$,
- uses just 6656 16-bit multiplications,
• uses just 6976 16-bit additions (counting subtractions as additions),
• stores data only every 3 NTT layers, and
• has only 4 layers of permutation instructions.

To put this in perspective, if each of the 9 NTT layers had 256 modular multiplications, 512 additions, and zero extra modular reductions, then in total there would be 6912 16-bit multiplications and 6912 16-bit additions, since each modular multiplication costs 3 16-bit multiplications and 1 16-bit addition.

At a higher level, we tweaked the order of operations in `mult*.c` as follows. As noted above, an NTT of size $2^N$ in `multN.c` was already decomposed into $2N/512$ size-$512$ FFTs, which were all handled with one call to the NTT API. At a higher level, `mult*.c` used the conventional approach to multiply two polynomials $f,g$ modulo $r$: apply an NTT to $f$, apply an NTT to $g$, multiply pointwise, and apply an inverse NTT—so there was one call to the NTT API for $f$ and one call to the NTT API for $g$. The tweak is that we merged these into a single call, reducing overhead.

Finally, we checked that all of the software follows the standard rules for constant-time cryptographic software. As a double-check, we used `crypto_declassify` to mark the safe rejection-sampling loop in `sntrup` key generation (generating an invertible $g$ in $\mathcal{R}/3$), and then checked that the software passes the TIMECOP tool inside SUPERCOP; TIMECOP is a wrapper around the standard idea of running `valgrind` to check for secret-dependent branches and array indices.

**D Is there any hope for ntruhrss701?**

CECPQ2’s ntruhrss701 keygen, like OpenSSLNTRU’s sntrup761 keygen, is bottlenecked by inversion. Conceptually, everything this paper does for sntrup761 can also be done for ntruhrss701, starting with converting a batch of 32 ntruhrss701 inversions into 1 inversion plus 93 multiplications. This appendix explains how difficult it would be for this to make ntruhrss701 an attractive option compared to sntrup761.

As a starting point, sntrup761 has a considerably higher security level: Core-SVP $2^{153}$ vs. $2^{136}$. Let’s reuse an existing model of lattice performance as being linear in the security level. This says that ntruhrss701 wins if it costs below 88% of sntrup761. In fact, ntruhrss701 was chosen at a “cliff” in the NTRU-HRSS design space and is not so easy to scale up to higher security levels, so this model is biased towards ntruhrss701, but let’s disregard this bias.

Costs include not just the costs of CPU time but also the costs of transmitting data. Notice that ntruhrss701 sends 3.6% more traffic than sntrup761 (see Table 1), meaning that ntruhrss701 is 15% worse per security level in data-communication costs. (In applications where keys are broadcast or pre-communicated through a lower-cost channel, ntruhrss701 transmits only an 1138-byte ciphertext, but sntrup761 transmits only an 1039-byte ciphertext, so ntruhrss701 is 23% worse per security level.) The question is whether ntruhrss701 can save enough CPU time to make up for this.

To put CPU time and communication costs on the same scale, let’s scale costs so that a CPU cycle costs 1, and let’s write $C$ for the cost of communicating a byte of data. Specifically, let’s use the existing estimate $C = 1000$. This estimate says, for example, that a quad-core 3GHz server has the same cost as a 100Mbits Internet connection.

Transmitting the sntrup761 key and ciphertext now costs 2.197 million. Our results reduce sntrup761 keygen+enc+dec to 259472 cycles, for total cost 2.456 million. (Before our work, the total cost was 3.120 million.)

For comparison, transmitting the ntruhrss701 key and ciphertext costs 2.276 million. The current ntruhrss701 keygen+enc+dec is 359076 cycles, for total cost 2.635 million. This is 7% higher cost than sntrup, meaning 21% higher cost per security level than ntruhrss701.

Now imagine ntruhrss701 consuming no CPU time at all. Notice that 2.276 million, the cost of just communication for ntruhrss701, is 93% of 2.456 million, our cost for communication and computation for sntrup761—in other words, 4% worse per security level. All possible room for ntruhrss701 to compete is thus eliminated by

• our keygen speedups,
• the slightly reduced traffic for sntrup761 compared to ntruhrss701, and
• the higher security level of sntrup761 compared to ntruhrss701,

under the reasonable assumption $C = 1000$ regarding the relative costs of communication and computation.

One could still try to argue that (1) ntruhrss701 is competitive in applications that reuse keys, and (2) sharing software will then justify also using ntruhrss701 in applications that don’t reuse keys, so (3) it is interesting to study the savings from batch keygen in ntruhrss701. But looking at the performance numbers shows that this argument fails at the first step. Compared to total keygen+enc+dec CPU time plus communicating keys+ciphertexts, if one switches to total enc+dec time plus communicating ciphertexts (or, even more extreme, total enc+dec time plus communicating keys+ciphertexts), then communication costs become a larger fraction of total costs. In all of these scenarios, sntrup761 wins because of its higher security level, its lower bandwidth, and our keygen speedups, independently of the ntruhrss701 CPU time.

What if we focus specifically on environments with much smaller values of $C$, say $C = 100$ or even smaller? The Haswell performance numbers before our work (titan0, Intel Xeon E3-1275 V3, supercop=202000000) showed

• ntruhrss701 taking 270548 cycles for keygen, 26936 cycles for enc, and 63900 cycles for dec; and

20
• sntrup761 taking 814608 cycles for keygen, 48892 cycles for enc, and 59404 cycles for dec.

Evidently ntruhrss701’s extra network traffic was buying some wins in CPU time: there was a slight loss of dec speed, but a larger win in enc speed, and a much larger win in keygen speed. If the extra network traffic is cheap, could these wins reduce the total ntruhrss701 cost below 88% of the sntrup761 cost?

Saying that this paper improves sntrup761 keygen from 814608 cycles to 156317 cycles, so sntrup761 now uses less CPU time than ntruhrss701, does not directly answer this question. If batching makes ntruhrss701 keygen 5× faster, shouldn’t it produce a similarly huge speedup in ntruhrss701 keygen?

A closer look at the underlying algorithms shows that it is an error to use the performance of unbatched keygen as a predictor of the performance of batched keygen. Unbatched keygen is bottlenecked by inversion, and ntruhrss701 exploits one of its design decisions—a power-of-2 modulus, which sntrup761 avoids because of security concerns—for a specialized type of inversion algorithm, a “Hensel lift”. Batched keygen is instead bottlenecked by multiplication, and benefits much less from a power-of-2 modulus. The Hensel speedup would still be measurable inside the occasional inversion, but batch size 32 compresses this speedup by a factor 32.

There is still some speedup from sntrup761 to ntruhrss701 in multiplications; this is not surprising, given that sntrup761 uses larger polynomials for its higher security level, and avoids composite moduli because of security concerns. Existing microbenchmarks suggest that one could reasonably hope for a 2× speedup from ntruhrss701 keygen to batched ntruhrss701 keygen. This would make ntruhrss701 cost 87% as many CPU cycles as sntrup761. However, this still would not outweigh the difference in security levels for $C = 100$. Even in an extreme $C = 0$ model, this hypothetical improvement would be only a 3% win for ntruhrss701, taking security levels into account. Meanwhile, for a reasonable $C = 1000$, the same hypothetical 2× improvement would still be a 15% loss for ntruhrss701, again taking security levels into account. For larger $C$, the loss would approach the 17% mentioned above.

### E Barrett reduction correctness

Recalling Section 3.1.1, Barrett reduction estimates the division $\frac{g}{f_i}$ as its quotient $h = \left[\frac{g}{f_i}\right]$. Then it calculates the remainder as $r = g - g \cdot \left[\frac{g}{f_i}\right] \cdot f_i$. We compare the difference between $\frac{g}{f_i} \cdot f_i$ and $\left[\frac{g}{f_i}\right] \cdot f_i$. It is the remainder $r$ if the difference is a polynomial of degree less than $\deg(f_i)$.

1. With an equivalence equation $x^{D_g} = q_s \cdot f_i + r_x$ from the pre-computation, we have the exact form

$$\frac{g}{f_i} = g \cdot \left(\frac{x^{D_g}}{f_i}\right) \cdot \frac{1}{x^{D_g}} = g \cdot (q_s \cdot \frac{r_x}{f_i}) \cdot \frac{1}{x^{D_g}}.$$  

2. Then we compute the difference

$$\frac{g}{f_i} \cdot f_i - \left[\frac{g}{f_i}\right] \cdot f_i = \left(\frac{g \cdot q_s}{x^{D_g}} - \left[\frac{g \cdot q_s}{x^{D_g}}\right]\right) \cdot f_i$$

$$- \frac{g \cdot r_x}{f_i} \cdot \frac{1}{x^{D_g}}.$$

3. Let $\frac{g \cdot q_s}{x^{D_g}} = h + l$ where $h = \left[\frac{g \cdot q_s}{x^{D_g}}\right] = \left[\frac{g}{f_i}\right]$ contains the non-negative degree terms and $l$ is the polynomial with negative degree terms. The term $\left(\frac{g \cdot q_s}{x^{D_g}} - \left[\frac{g \cdot q_s}{x^{D_g}}\right]\right)$ is a polynomial of degree less than $\deg(f_i)$.

4. The last term of Equation 3 $\frac{g \cdot \frac{r_x}{f_i}}{x^{D_g}}$ is also a polynomial of degree less than $\deg(f_i)$, since $\deg(g) < D_g$ and $\deg(r_x) < \deg(f_i)$. Hence, we conclude the difference is a polynomial of degree less than $\deg(f_i)$.

### F More on benchmarks

#### F.1 Batch key-generation microbenchmarks

Table 2 shows the performance and key pair storage of BatchKeyGen regarding various batch sizes $n$.

#### F.2 In-browser handshake macrobenchmarks

Section 4.4 described how we developed tls_timer, a dedicated handshake benchmarking client, to measure the end-to-end performance of OpenSSLNTRU. The need to fully control the sampling process with tls_timer, arose after an initial attempt to measure the end-to-end performance from within the GNOME Web browser. Specifically, we originally designed the experiment to let the browser first connect to a web server via stunnel to retrieve a static HTML page. This in turn embedded JavaScript code to open and time a number of connections in parallel to further retrieve other resources from a web server. We designed these resources to:

- have short URI to minimize application data in the client request, which length-wise is dominated by HTTP headers outside of our control;
- have randomized URI matching a “rewrite rule” on the web server, mapping to the same file on disk. This allows the server to cache the resource and skip repeated file system accesses, while preventing browser optimizations to avoid downloading the same URI repeatedly or concurrently;
- be short, comment-only, JavaScript files, to minimize transferred application data from the server, and, on the browser side, the potential costs associated with parsing and rendering pipelines.
Table 2: Performance of BatchKeyGen regarding various batch sizes $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>sntrup653</td>
<td>amortized cost</td>
<td>778218</td>
<td>438714</td>
<td>295150</td>
<td>229429</td>
<td>180863</td>
<td>152737</td>
<td>147821</td>
</tr>
<tr>
<td></td>
<td>latency</td>
<td>778218</td>
<td>877428</td>
<td>1180600</td>
<td>1835432</td>
<td>2893808</td>
<td>5256300</td>
<td>9775160</td>
</tr>
<tr>
<td></td>
<td>pk storage</td>
<td>994</td>
<td>1988</td>
<td>3976</td>
<td>7952</td>
<td>15904</td>
<td>31808</td>
<td>63616</td>
</tr>
<tr>
<td></td>
<td>sk storage</td>
<td>1518</td>
<td>3036</td>
<td>6072</td>
<td>12144</td>
<td>24288</td>
<td>48576</td>
<td>97152</td>
</tr>
<tr>
<td>sntrup761</td>
<td>amortized cost</td>
<td>819332</td>
<td>567996</td>
<td>351329</td>
<td>242043</td>
<td>181274</td>
<td>156317</td>
<td>147809</td>
</tr>
<tr>
<td></td>
<td>latency</td>
<td>819332</td>
<td>1135992</td>
<td>1405316</td>
<td>1936340</td>
<td>2900380</td>
<td>5002124</td>
<td>9459748</td>
</tr>
<tr>
<td></td>
<td>pk storage</td>
<td>1158</td>
<td>2316</td>
<td>4632</td>
<td>9264</td>
<td>18528</td>
<td>37056</td>
<td>74112</td>
</tr>
<tr>
<td></td>
<td>sk storage</td>
<td>1763</td>
<td>3526</td>
<td>7052</td>
<td>14104</td>
<td>28208</td>
<td>56416</td>
<td>112832</td>
</tr>
<tr>
<td>sntrup857</td>
<td>amortized cost</td>
<td>1265056</td>
<td>708104</td>
<td>458562</td>
<td>322352</td>
<td>255815</td>
<td>216618</td>
<td>201173</td>
</tr>
<tr>
<td></td>
<td>latency</td>
<td>1265056</td>
<td>1416208</td>
<td>1834248</td>
<td>2578812</td>
<td>4093040</td>
<td>6931748</td>
<td>12875024</td>
</tr>
<tr>
<td></td>
<td>pk storage</td>
<td>1322</td>
<td>2644</td>
<td>5288</td>
<td>10576</td>
<td>21152</td>
<td>42304</td>
<td>84608</td>
</tr>
<tr>
<td></td>
<td>sk storage</td>
<td>1999</td>
<td>3998</td>
<td>7996</td>
<td>15992</td>
<td>31984</td>
<td>63968</td>
<td>127936</td>
</tr>
</tbody>
</table>

Unfortunately, this approach proved to be unfruitful, as the recorded measures were too coarse and noisy. This is mostly due to the impossibility of completely disabling caching on the browser through the JavaScript API and developer options, delayed multiplexing of several HTTP requests over a single TLS connection, ignored session keep-alive settings, and, possibly, the effect of intentionally degraded clock measurements when running JavaScript code fetched from a remote origin.

G OpenSSL: software architecture

Section 4.1 details the part of our contributions consisting in a set of patches that applies to the source code of OpenSSL 1.1.1. We designed our patches to provide full API and ABI compatibility with binary distributions of OpenSSL 1.1.1, while transparently enabling linking applications to perform post-quantum key exchanges in TLS 1.3 handshakes. The description of details of our contribution relies on various technical concepts regarding OpenSSL; this appendix reviews this background.

Illustrated in Figure 4, as a library to build external applications, OpenSSL is divided into two software libraries, namely libcrypto and libssl. The former provides cryptographic primitives and a set of utilities to handle cryptographic objects. The latter implements support for TLS 1.3 and other protocols, deferring all cryptographic operations and manipulation of cryptographic objects to libcrypto.

Due to its legacy, libcrypto exposes a wide programming interface to users of the library, offering different levels of abstraction. Currently, the recommended API for external applications and libraries (including libssl) to perform most cryptographic operations is the EVP API.

The EVP API, especially for public key cryptography, offers a high degree of crypto agility. It defines abstract cryptographic key objects, and functions that operate on them, in terms of generic operations primitives (e.g., EVP_PKEY_encrypt()/EVP_PKEY_decrypt(), or EVP_DigestSign()/EVP_DigestVerify(), etc.). This lets the libcrypto framework pick the algorithm matching the key type and the best implementation for the application platform. Using the API appropriately, a developer can write code that is oblivious to algorithm selection. That is, leaving algorithm adoption choices to system policies in configuration files, or in the creation of the serialized key objects fed to libcrypto.

In this work, we patch libssl to support the negotiation of KEM groups over TLS 1.3, mapping KEM operations over the existing EVP API. The API itself does not include abstractions for the Encapsulate() and Decapsulate() KEM primitives.

8https://www.openssl.org/docs/man1.1.1/man7/evp.html
Figure 4: Architecture diagram of the end-to-end experiment, derived from [45, Figure 2]. The red boxes within `libssl` and `libcrypto` represent patches applied to OpenSSL 1.1.1 to enable our post-quantum KEM experiment over TLS 1.3.