OpenSSLNTRU: Faster post-quantum TLS key exchange

Daniel J. Bernstein, *University of Illinois at Chicago and Ruhr University Bochum;* Billy Bob Brumley, *Tampere University;* Ming-Shing Chen, *Ruhr University Bochum;* Nicola Tuveri, *Tampere University*

https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

OpenSSLNTRU: Faster post-quantum TLS key exchange

Daniel J. Bernstein[1,2], Billy Bob Brumley[3], Ming-Shing Chen[2], and Nicola Tuveri[3]
authorcontact-openssllntru@box.cr.yp.to
[1]*Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607-7045, USA*
[2]*Ruhr University Bochum, Bochum, Germany*
[3]*Tampere University, Tampere, Finland*

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 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. Constraints that make this more difficult than it sounds include 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" [27] 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 [9]. 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.

In July 2020, the NIST project began its third round [1], selecting 4 "finalist" and 5 "alternate" encryption proposals to consider for standardization at the end of the round and 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 [17], 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 CECPQ1 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.

Table 1: Cryptographic features of the post-quantum components of CECPQ2 (previous work) and OpenSSLNTRU (this paper). Core-SVP in the table is pre-quantum Core-SVP (see [12, Section 6]); post-quantum Core-SVP has 10% smaller exponents. See [13] regarding cyclotomic concerns. The `ntruhrss701` cycle counts are from `supercop-20210423` [10] on `hiphop` (Intel Xeon E3-1220 V3). The `sntrup761` cycle counts are old→**new**, where "old" shows the best `sntrup761` results before our work and "**new**" shows results from this paper's freely available software; Appendix A presents the slight enc and dec speedups, and Section 3 presents the large keygen speedup.

	CECPQ2	OpenSSLNTRU
cryptosystem	`ntruhrss701`	`sntrup761`
key+ciphertext bytes	2276	2197
keygen cycles	269191	814608→**156317**
enc cycles	26510	48892→**46914**
dec cycles	63375	59404→**56241**
Core-SVP security	2^{136}	2^{153}
cyclotomic concerns	yes	no

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` [12], to compare to CECPQ2's `ntruhrss701`.

Each of `ntruhrss701`/`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; 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 (see Section 2.5.2). 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.

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 NIST P-256, and even completes more sessions per second than commonly deployed pre-quantum X25519 (see Section 4.4). This remains true even when we replace `sntrup761` with higher-security `sntrup857`.

2 Background

2.1 Polynomial rings in NTRU Prime

Streamlined NTRU Prime [12], abbreviated `sntrup`, uses arithmetic in finite rings $\mathcal{R}/3 = (\mathbb{Z}/3)[x]/(x^p - x - 1)$ and $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$, where $\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1)$.

The parameters p, q are chosen so that $\mathcal{R}/q$ is a field.

Short means the set of polynomials in $\mathcal{R}$ 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 [24] 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
$$(c_n, \ldots, c_2) = (a_n^{-1}, \ldots, a_2^{-1}) \text{ and}$$
$$(t_{n-1}, \ldots, t_2, t_1) = ((\Pi_{i=1}^{n-1} a_i)^{-1}, \ldots, (a_1 \cdot a_2)^{-1}, a_1^{-1}) \ .$$
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 $((\mathbb{Z}/q)[x]/(x^{n/2} - b)) \times ((\mathbb{Z}/q)[x]/(x^{n/2} + b))$. Due to the Chinese remainder theorem (CRT), the mapping is invertible when $2b$ is invertible. Specifically, let $f = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \in (\mathbb{Z}/q)[x]/(x^n - b^2)$. The trick maps f to

$$\begin{aligned}&(f \bmod (x^{n/2}+b) \ , f \bmod (x^{n/2}-b))\\ =&((f_0 - bf_{n/2}) + \cdots + (f_{n/2-1} - bf_{n-1})x^{n/2-1},\\ &(f_0 + bf_{n/2}) + \cdots + (f_{n/2-1} + bf_{n-1})x^{n/2-1})\end{aligned}$$

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 $\hat{f} = (\hat{f}_0, \ldots, \hat{f}_j, \ldots, \hat{f}_{n-1}) \in (\mathbb{Z}/q)^n$ where $\hat{f}_j = f \bmod (x - \psi^j) = \sum_{i=0}^{n-1} f_i \psi^{ij}$, 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 [31] 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 $((\mathbb{Z}/q)[x][y]/(y^{2n} - 1))/(x^m - y)$. Then, it lifts f to $(\mathbb{Z}/q)[x][y]/(y^{2n} - 1)$, 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 [28] 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 $((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]$, and maps to $((\mathbb{Z}/q)[y]/(y^{2n} + 1))[x]/(x^{2n} - 1)$ 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 [26], we optimize `sntrup` 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 `ymm` registers; each `ymm` register splits into two 128-bit `xmm` lanes. The instruction set treats data in `ymm` registers as lanes (independent partitions) of 32×8-bit, 16×16-bit, 8×32-bit, etc.; every instruction operates simultaneously on the partitioned data in the `ymm` registers. In 2013, the Haswell architecture extended AVX to AVX2 for enhanced integer operations.

2.5 Related works

2.5.1 NTT-based multiplication in other PQC finalists

Among the lattice based KEM of NIST's finalists, Kyber [4] operates in a radix-2 NTT friendly polynomial ring and implements NTT-based multiplication in the proposal. SABER [5] and NTRU [17] operate in polynomial rings with a power-of-two modulus which are considered NTT-unfriendly. The earlier implementations of two schemes used a combination of Toom-4 and Karatsuba based polynomial multiplication.

Recently, [18] showed that NTT-based multiplication outperforms previous Toom-Cook multiplication for implementing NTT-unfriendly SABER and most parameters of NTRU. To use NTT-based multiplication in an NTT-unfriendly ring, they raise the coefficients to a combination of several NTT-friendly polynomial rings, perform several NTT-based multiplications, and map back to original ring with CRT. For NTRU on the AVX2 platform, they reported significant improvement for parameters with polynomials of degree greater then 700. For Saber, they also reported a pronounced performance gain although the degree of polynomials are only 255. It is because the matrix-vector multiplication allows them to save the input NTT transforms for the elements in the common vector which performs inner products with different rows in the matrix.

2.5.2 Integrating cryptographic primitives

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

The Open Quantum Safe (OQS) project [34] 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. While [34] focused primarily on key agreement, the OQS OpenSSL fork does also support signatures and certificates using post-quantum algorithms, and their negotiation in TLS. See [29] for a study, conducted using OQS, benchmarking post-quantum TLS authentication. We also note that the end-to-end experiment we present in this paper is limited to one candidate and two sets of parameters (`sntrup761` and `sntrup857`), while the OQS project provides implementations for all finalists.

Similarly, the PQClean project [22] collects a number of implementations for the candidates. However, it does not aim to include integration into higher-level applications or protocols.

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.

Schwabe, Stebila, and Wiggers [32] 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. Therefore, showcasing how at the protocol level our experiment does not alter the TLS 1.3 message flow, in Figure 1 we only highlight the cryptographic operations and material involved in the key exchange—carried in the `ClientHello` and `ServerHello` messages—while keys and signatures used for authentication—as part of the `Certificate` and `CertificateVerify` messages—do not address post-quantum concerns.

Our approach to OpenSSL integration via an `ENGINE` module is based on the methodology suggested in [36], where the authors instantiated `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`, `engNTRU`, builds upon `libbecc` [15], which is itself derived from `libsuola`. Both previous works applied the `ENGINE` framework to integrate alternative ECC implementations. The latter is particularly close to `engNTRU`, as it also featured a transparent mechanism to handle batch key generation. Section 4.2 details how `engNTRU` evolved from these works and the unique features it introduces.

Shacham and Boneh [33] 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.

Comparison table. Based on the previous discussions in this section, Table 2 compares select TLS integration experiments regarding post-quantum algorithms.

The "Hybrid" criterion tracks approaches that simultaneously protect the key agreement with "traditional" (usually ECC) and post-quantum encryption (see, e.g., [14, 35]). This paper does not make recommendations for or against hybrids; our performance and software-engineering contributions are equally applicable to hybrid and non-hybrid scenarios. Figure 1 illustrates how any NIKE system can be transformed

Table 2: Comparison of select TLS integration experiments.

	OQS	CECPQ2	KEMTLS	OpenSSLNTRU
Hybrid[1]	opt.[7]	yes	no	no
PFS[2]	yes	yes	yes	yes
PQ-sec. key agmt.[3]	yes	yes	yes	yes
PQ-sec. auth.[4]	opt.	no	yes	no
TLS 1.3 compat.[5]	yes	yes	no	yes
Binary compat.[6]	no	no	no	yes

[1] Key-agreement uses ECC *and* post-quantum encryption.
[2] Key-agreement provides Perfect Forward Secrecy.
[3] Post-quantum security over TLS key agreement.
[4] Post-quantum security over TLS authentication, inherently limited by access to PQ PKI.
[5] Requires no breaking changes to the TLS 1.3 message flow.
[6] ABI compatible, to easily integrate into the existing software ecosystem.
[7] [34] presents experimental results for both post-quantum and hybrid KEMs. Using the OQS fork of OpenSSL, the choice of supported KEMs and order of preference is left to developers and system administrators.

into an equivalent KEM construction; a protocol that supports key exchange via KEM can support hybrid handshakes by simply composing two or more underlying KEMs to obtain a hybrid KEM.

The "PFS" criterion tracks approaches that provide the traditional notion of Perfect Forward Secrecy w.r.t. the key agreement phase of the handshake. Different experiments take post-quantum security into consideration at different cryptosystem components, tracked by the "key agreement" and "authentication" criteria. The latter comes with the caveat that the extent to which PQ authentication is achieved is inherently limited by access to a fully post-quantum Public Key Infrastructure (PKI). In the specific case of the Internet Web PKI, client and server need to share a chain of certificates up to a common root of trust, entirely signed with PQ algorithms. Some experiments require breaking changes to the TLS 1.3 message flow, depicted in Figure 1; "compatibility" tracks this criterion. Lastly, our work is the only experiment we are aware of that achieves ABI compatibility ("Binary") to easily integrate into the existing software ecosystem.

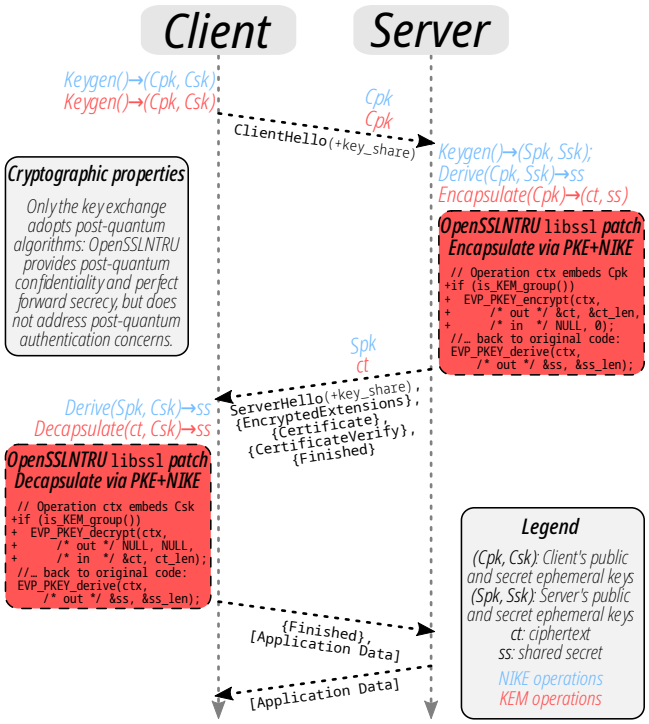

Figure 1: Overview of a full TLS 1.3 handshake. In OpenSSLNTRU, the traditional NIKE operations are replaced with KEM operations. Besides dedicated `NamedGroup` codepoints, this is transparent to TLS 1.3 messages as `key_share` payloads are opaque. As OpenSSL `1.1.1` does not offer an API for KEM operations, our patch uses the described PKE+NIKE workaround when one of the supported KEM groups is negotiated.

3 Batch key generation for `sntrup`

This section presents batch key generation for `sntrup` 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 `sntrup` key generation algorithm `KeyGen` outputs an `sntrup` 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 $f \in \mathsf{Short}$.
4. Compute $h = g/(3f)$ in $\mathcal{R}/q$.
5. Output $(h,(f,1/g))$ where h is the public key and $(f,1/g) \in \mathsf{Short} \times \mathcal{R}/3$ 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 $f \in \mathsf{Short}$, 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 [11] for extended GCD. Calling `batchInv` removes this side effect and requires a preliminary check for invertibil-

ity 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`

$G \leftarrow [\cdot]$ ▷ an empty list
$F \leftarrow [\cdot]$
while `len`$(G) < n$ **do**
  $g \xleftarrow{\$} \mathcal{R}/3$ ▷ \$: uniform random
  if not `isInvertible`(g) **: continue**
  $f \xleftarrow{\$}$ Short
  G`.append`(g)
  F`.append`(f)
end while
$\bar{G} \leftarrow$ `batchInv`(G)
$\bar{F} \leftarrow$ `batchInv`$([3 \cdot f$ **for** $f \in F])$
$H \leftarrow \left[g \cdot \bar{f} \in \mathcal{R}/q \text{ for } g \in G, \bar{f} \in \bar{F}\right]$
return $\left[(h,(f,\bar{g}))\right.$ **for** $\left.h \in H, f \in F, \bar{g} \in \bar{G}\right]$

3.1.1 Invertibility check for elements in $\mathcal{R}/3$

At a high level, we check the invertibility of an element $g \in \mathcal{R}/3$ by computing its remainder of division by the irreducible factors of $x^p - x - 1$ modulo 3, as suggested in [12, p. 8]. This section optimizes this computation.

For convenience, we always lift the ring element g to its polynomial form $g \in (\mathbb{Z}/3)[x]$ in this section. In a nutshell, if $g \bmod f_i = 0$ for any factor f_i of $x^p - x - 1$, then g is not invertible in $\mathcal{R}/3$.

We calculate the remainder of $g \bmod f_i$ with Barrett reduction [23]. Suppose the polynomial $x^p - x - 1 \in (\mathbb{Z}/3)[x]$ has m irreducible factors $(f_1, \ldots, f_m)$, i.e., $x^p - x - 1 = \Pi_{i=1}^{m} f_i$. Given a polynomial $g \in (\mathbb{Z}/3)[x]$ and $p > \deg(g) > \deg(f_i)$, we calculate the reminder $r = g \bmod f_i$ as follows. In the *pre-computation* step, choose $D_g > \deg(g)$ and $D_{f_i} > \deg(f_i)$, and calculate q_x as the quotient of the division x^{D_g}/f_i, i.e., $q_x = \lfloor x^{D_g}/f_i \rfloor$, where the floor function $\lfloor \cdot \rfloor$ removes the negative-degree terms from a series. In the *online* step, compute $h = \lfloor g \cdot q_x / x^{D_g} \rfloor = \lfloor g/f_i \rfloor$, i.e., the quotient of the division $g \cdot q_x / x^{D_g}$. Finally, return the remainder $r = g - h \cdot f_i$. We show this gives the correct r in Appendix C.

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

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 $D_{f_1} = 32$, $D_{f_2} = 64$, and $D_g = 768$ for computing $g \bmod f_0$ and $g \bmod f_1$. For computing $g \bmod f_3$, we note the pre-computed quotient $q_x = \lfloor x^{768}/(x^{682} + \cdots) \rfloor$ satisfies $\deg(q_x) = 88$. Hence, the multiplication $h = \lfloor g \cdot q_x / x^{768} \rfloor$ involves $\deg(g) = 768$ and $\deg(q_x) = 88$ 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 16×16 polynomial multiplier as a building block for schoolbook multiplication. We then use Karatsuba to build longer multipliers, such as 32×32, 64×64, 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 $4 \times 4 \to 8$ polynomial multiplication using a $32 \times 32 \to 64$ *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 $32 \times 32 \to 64$ 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 4×4 polynomial multiplication (i.e., $32 \times 32 \to 64$ integer multiplication), as our building block to implement 16×16 polynomial multiplication with the schoolbook algorithm.

3.2.2 Multiplying polynomials of length $3n$

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

Take two polynomials $F_0+F_1t+F_2t^2$ and $G_0+G_1t+G_2t^2$ in $(\mathbb{Z}/3)[x]$, where $\deg(F_i)<n$, $\deg(G_i)<n$, and $t=x^n$. Their product $H=H_0+H_1t+H_2t^2+H_3t^3+H_4t^4$ can be reconstructed by the projective Lagrange interpolation formula

$$\begin{aligned}H =& H(0)\frac{(t-1)(t+1)(t-x)}{x}+H(1)\frac{t(t+1)(t-x)}{x-1}\\ &+H(-1)\frac{t(t-1)(t-x)}{x+1}+H(x)\frac{t(t-1)(t+1)}{x(x-1)(x+1)}\\ &+H(\infty)t(t-1)(t+1)(t-x)\ .\end{aligned}$$

Here

$$\begin{aligned}H(0) &= F_0\cdot G_0,\\ H(1) &= (F_0+F_1+F_2)\cdot(G_0+G_1+G_2),\\ H(-1) &= (F_0-F_1+F_2)\cdot(G_0-G_1+G_2),\\ H(x) &= (F_0+F_1x+F_2x^2)\cdot(G_0+G_1x+G_2x^2),\text{ and}\\ H(\infty) &= F_2\cdot G_2\end{aligned}$$

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

H simplifies to

$$\begin{aligned}H =& H(0)-[U+(H(1)-H(-1))]\cdot t\\ &-[H(0)+(H(1)+H(-1))+H(\infty)]\cdot t^2\\ &+U\cdot t^3+H(\infty)\cdot t^4\ ,\end{aligned}\tag{1}$$

where $U=V+H(0)/x-H(\infty)\cdot x$ and

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

There are two tricky issues while computing V. First, $\deg(H(x))\le 2n+2$, introducing extra complexity since all other polynomials have degree less than $2n$. By requiring $\deg(F_2)\le n-2$ and $\deg(G_2)\le n-2$, we force $\deg(H(x))\le 2n$. 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 $(\mathbb{Z}/3)[x]$. 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 $(\mathbb{Z}/3)[x]$

Dividing a polynomial f by x^2-1 means producing a representation of $f=q\cdot(x^2-1)+r$, where q and $r=r_1x+r_0$ 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

$$r\cdot x^{2m}=(rx^{2m-2}+rx^{2m-4}+rx^{2m-6}+\cdots+r)(x^2-1)+r\ ,$$

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

$$\begin{aligned}f\cdot x^{2m}+g =& \left[q\cdot x^{2m}+r\cdot x^{2m-2}\right](x^2-1)\\ &+(s+rx^{2m-4}+\cdots+r)(x^2-1)+(t+r)\ .\end{aligned}\tag{2}$$

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 every 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 $n=2^l$, proceeds as follows. The computation comprises $l-1$ 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 `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 `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 16×16 polynomial multiplication actually performs two independent 16×16 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 [21, p. 237] for the `vperm2i128`, `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_l,cd_l) and (ab_h,cd_h) in two `ymm` registers, where $a\cdot b=ab_l+ab_h\cdot x^{16}$ and $c\cdot d=cd_l+cd_h\cdot x^{16}$. Thus, we avoid the data exchange between `xmm` lanes.

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

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 $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$ 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 $\mathcal{R}/q$ 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, Wälde, and Yang [3] presented a non-radix-2 NTT implementation on $(\mathbb{Z}/4591)[x]/(x^{1530} - 1)$ 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 $\mathcal{R}/q$, coefficients in $\{-1, 0, 1\}$; the maximum coefficient of a "big×small" product is below $7681 \cdot 10753/2$ 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.

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 $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]/(x^8 - y)$, lift to $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]$, and map to $((\mathbb{Z}/q)[y]/(y^8 + 1))[x]/(x^{16} - 1)$. The polynomial ring $(\mathbb{Z}/q)[y]/(y^8 + 1)$ 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 $(\mathbb{Z}/q)[y]/(y^8 + 1)$. We use Montgomery multiplication [25] 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.

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 of `ymm` 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 16×16 matrix transposition. Given sixteen degree-15 polynomials $(a_0^{(0)} + a_1^{(0)}x + \cdots + a_{15}^{(0)}x^{15}), \cdots, (a_0^{(15)} + \cdots + a_{15}^{(15)}x^{15})$, data in $(\ldots)$ represents one `ymm` register, and we treat a polynomial in one `ymm` register as a row of a 16×16 matrix. Transposing this matrix rearranges the data to $(a_0^{(0)}, \cdots, a_0^{(15)}), \cdots, (a_{15}^{(15)}, \cdots, a_{15}^{(15)})$. 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 [37] for matrix transposition. The

technique transposes a 2×2 matrix by swapping its two off-diagonal components. For transposing matrices with larger dimensions, e.g. 4×4, it first swaps data between two 2×2 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 benchmarks here for `sntrup653` because it actually uses the same multiplier as `sntrup761`.

Benchmarks for $\mathcal{R}/3$. We compare cycle counts for $\mathcal{R}/3$ multiplication between our implementation and the best previous `sntrup` implementation, `round2` in [10], in the following table.

Parameter	Implementation	Cycles
`sntrup761`	this work (Section 3.2)	8183
	this work (NTT, Appendix A)	8827
	NTRUP `round2` (NTT, [10])	9290
`sntrup857`	this work (Section 3.2)	12840
	this work (NTT, Appendix A)	12533
	NTRUP `round2` (NTT, [10])	12887

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

Another question is the efficiency of Montgomery's trick for inversion in $\mathcal{R}/3$. Recall that, roughly, the trick replaces one multiplicative inversion by three ring multiplications, one amortized ring inversion, and one check for zero divisors. We show benchmarks of these operations in the following table.

Parameter	Operation	Cycles
`sntrup653`	Ring inversion	95025
	Invertibility check	22553
	Ring multiplication	8063
`sntrup761`	Ring inversion	114011
	Invertibility check	9668
	Ring multiplication	8183
`sntrup857`	Ring inversion	160071
	Invertibility check	12496
	Ring multiplication	12533

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.

Benchmarks for $\mathcal{R}/q$. The following table shows the cycle counts of big×big multiplication and big×small multiplication in $\mathcal{R}/q$, comparing with the previous best software [10].

Parameter	Implementation	Cycles
`sntrup761`	this work (Section 3.3), big×big	25113
	this work (Appendix A), big×small	16992
	NTRUP `round2` [10], big×small	18080
`sntrup857`	this work (Section 3.3), big×big	32265
	this work (Appendix A), big×small	24667
	NTRUP `round2` [10], big×small	25846

The results show the absolute cycle count of big×big is larger than big×small multiplication. To evaluate the efficiency of big×big multiplication, consider if we extend the big×small multiplication to big×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×big multiplication presented in this work.

Since big×small multiplication is faster than big×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×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×small and one big×big multiplication is clearly much less than one inversion operation.

Benchmarks for batch key generation. We show the benchmark results for batch key generation (`BatchKeyGen`) in Figure 2. See also Table 3.

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

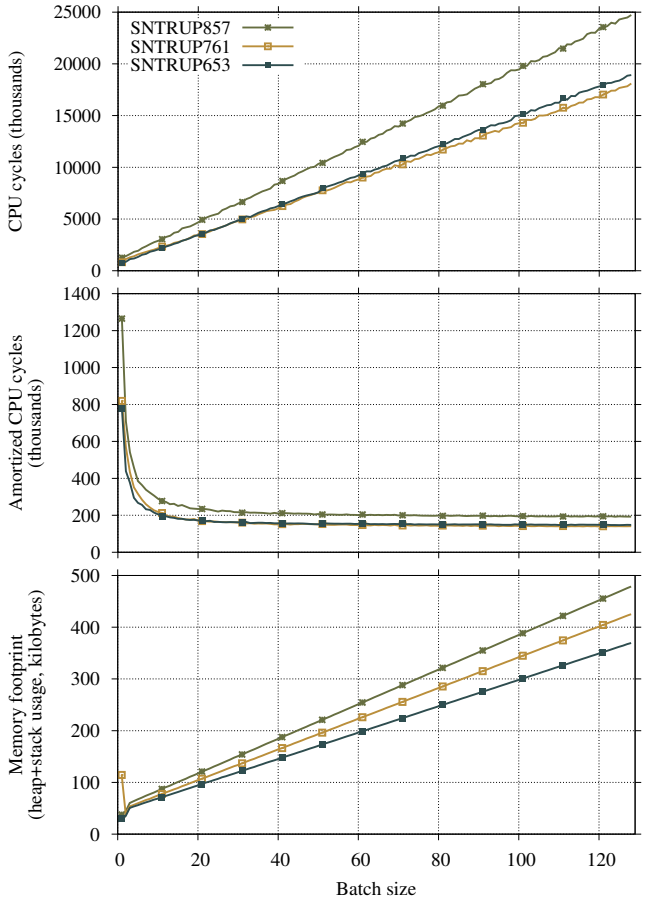

Figure 2: `BatchKeyGen` metrics regarding various batch sizes (n). Top: full batch cost in CPU cycles. Middle: amortized cost in CPU cycles, dividing by n. Bottom: memory footprint, i.e., heap+stack usage, in kilobytes.

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 E 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 3 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`

[1] GNOME Web (a.k.a. `epiphany`) — https://wiki.gnome.org/Apps/Web

[2] `stunnel` — https://www.stunnel.org/

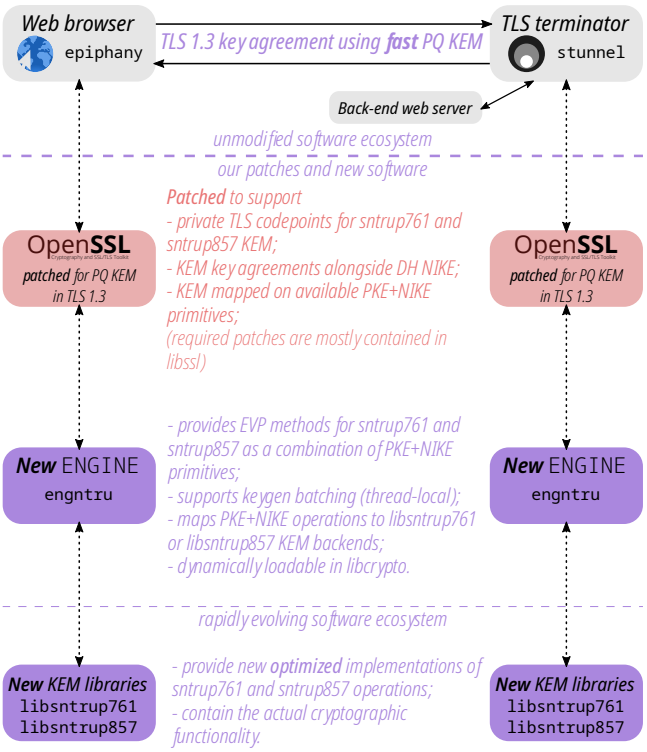

Figure 3: Overview of our end-to-end experiment.

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.

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 conditionally use a KEM *`Encapsulate()`* operation for KEM groups;
- Modify the client-side handling of the `key_share` extension in an incoming `ServerHello` to conditionally 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

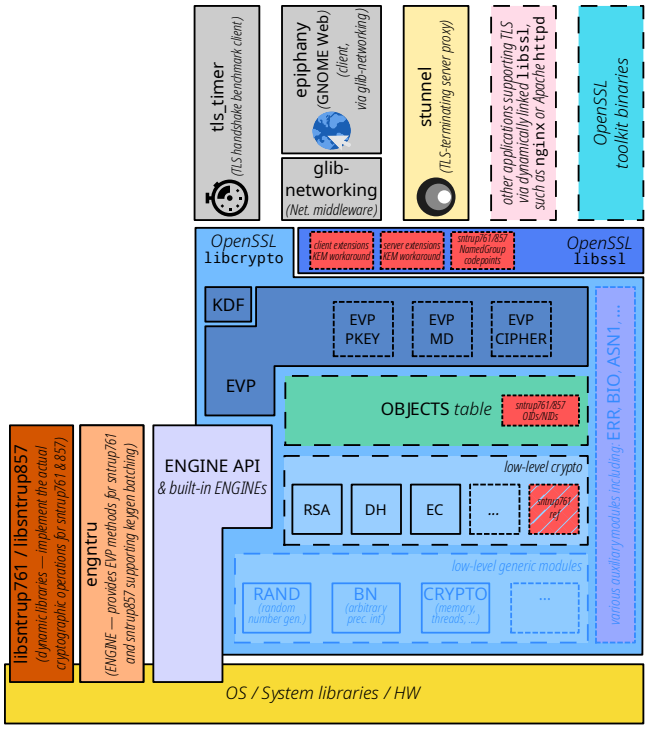

Figure 4: Architecture diagram of the end-to-end experiment, derived from [36, Figure 2]. The red boxes within `libssl` and `libcrypto` represent patches applied to OpenSSL `1.1.1` to enable our post-quantum KEM experiment over TLS 1.3. The striped `sntrup761 ref` box represents the optional patch to also include a reference implementation for `sntrup761` inside `libcrypto`. When loaded, `engNTRU` overrides it with the optimized implementation from `libsntrup761`.

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.

As depicted in Figure 1, on the server side, traditional NIKE groups generate an ephemeral key pair, 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()`. With our patch, if the group is flagged as a KEM group, instead we treat the `key_share` payload as the ciphertext to be used in `EVP_PKEY_decrypt()` under the client's secret key, and with `NULL` output. The resulting plaintext is stored in the internal state of the client-side object representing the client's key pair. The plaintext shared secret is ultimately retrieved via `EVP_PKEY_derive()`.

The last patch alters the `libssl` static table of supported TLS 1.3 groups. It assigns private `NamedGroup` codepoints to negotiate `sntrup761` or `sntrup857` key exchanges, flagged as KEM groups, and links it to static numeric identifiers (`NIDs`) defined within `libcrypto` headers. These identify implementations of `sntrup761` and `sntrup857`, as described in the next paragraph.

`libcrypto` changes. `libcrypto 1.1.1` has the ability to generate `NIDs` dynamically for custom algorithms unknown at OpenSSL build time. In contrast, `libssl 1.1.1` defines supported groups in a static table generated during compilation. It is technically possible to inject KEM functionality (using the PKE+NIKE workaround described above) via a custom `ENGINE` without any change to `libcrypto`. Yet, the limited support for dynamic customization in `libssl` adds the requirement for a `libcrypto` patch to issue static `NIDs` for `sntrup761` and `sntrup857`. This is so they can be included in the `libssl` static table at compile time. For each parameter set, this patch uses the internal OpenSSL tooling to issue a novel static `NID` and associate it with the corresponding `sntrup*` algorithm and a custom *object identifier* (`OID`),[4] required for serializing and deserializing key objects. With this data, the tooling updates the public `libcrypto` headers, adding the relevant `sntrup*` definitions.

Additionally, we include an optional patch for `libcrypto` that adds a reference implementation of `sntrup761` as a new `libcrypto` submodule. Including this patch allows us to test the implementation provided by `engNTRU` against the reference implementation, and also to test the software stack on the server and the client in absence of the `ENGINE`. This eases the debug process during the development of `engNTRU`. For the final users of our end-to-end scenario, this patch is entirely optional, as the dynamic `ENGINE` injects the optimized

[3] Generally 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] https://www.itu.int/en/ITU-T/asn1/Pages/OID-project.aspx

implementation for the cryptographic primitive if it is absent. Otherwise, it overrides the default reference implementation if it is already included in `libcrypto`.

4.2 The `engNTRU` `ENGINE`

As mentioned in Section 2.5.2 and depicted in Figure 3 and Figure 4, as part of our end-to-end experiment, we introduce a new `ENGINE`, dubbed `engNTRU`.

We followed the methodology suggested in [36], 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), security considerations, and general motivations to use the `ENGINE` framework for applied research. In this section, we highlight how this choice has two main benefits: it decouples OpenSSL from fast-paced development in the ecosystem of optimized implementations for post-quantum primitives, and at the same time it decouples external libraries implementing novel primitives from the data types and patterns required for OpenSSL compatibility.

`engNTRU` builds upon `libbecc` [15], which is itself derived from `libsuola` [36]. 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 multiple providers to interface with the OpenSSL API. Under the `serial_lib` provider, each *`Keygen()`* operation is mapped to `crypto_kem_keypair()`, generating a new key pair 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 key pairs. Every time an application thread requests a new `sntrup*` key pair, `engNTRU` attempts to retrieve a fresh one from the corresponding pool. For each supported parameter set, it dynamically allocates a pool, initialized the first time a key pair 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` serves each request by copying (and then securely erasing from the pool buffer) the next fresh entry in the pool. After this, if consuming the key pair emptied the pool, `engNTRU` fills it again, by calling the corresponding `libsntrup*` batch generation function. This happens synchronously, before returning control to the application. Storing keys for deferred use adds security concerns: `engNTRU` addresses them relying on standard OS guarantees for the protection of memory contents across processes and users. On the other hand, the batch strategy decouples the generation of a key pair from its use in the application (e.g., an attacker's connection request), which complicates many implementation attacks, and results in an overall positive security impact.

In terms of performance, 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 [16], latencies below 20ms are imperceptible. A series of K `sntrup761` keys costs $0.4K$ms 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`, nor 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 `libssl 1.1.1`, 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 TLS service from a client application that does not support TLS. For example, a user could execute the `telnet` application (with no TLS support) to connect to a client mode instance of `stunnel`, which would then TLS-wrap the connection to a static SMTPS server to securely transfer email.

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 web page over HTTP with the `netcat` utility, which `stunnel` would then TLS-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 `engNTRU` 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, mainlined 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`.

[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.

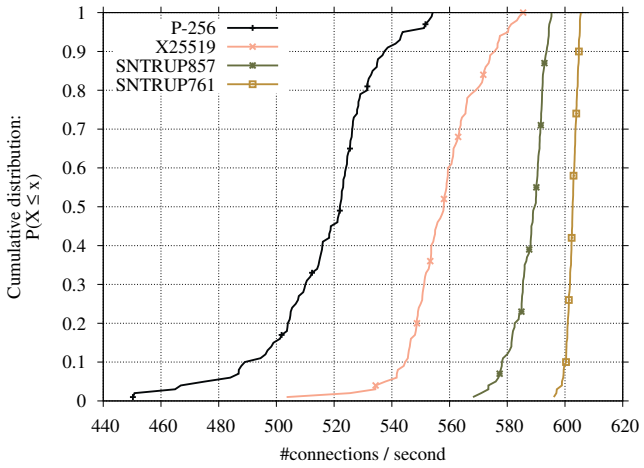

Figure 5: Cumulative distributions of handshake performance under different cryptosystems in a local network. Each curve represents a key-exchange group, for which we collected 100 samples, in terms of average number of connections per second. This metric is extrapolated from measuring the elapsed wall-clock time over 8192 sequentially established connections per sample.

One such application, and one highlight of our experiment, 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 D for a discussion about in-browser benchmarks). Referring to the diagram in Figure 3, 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 content of the exchanged messages. It also includes the time spent due to transit of packets over the network, and through userland/kernelspace 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, as summarized in Figure 1.

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 5 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.

One should not conclude that `sntrup761` and `sntrup857` cost less than ECC overall. The unloaded high-bandwidth network of our experimental environment masks the higher communication costs of the lattice cryptosystems, whereas reasonable estimates of communication costs (see generally Appendix B) say that every lattice system costs more than ECC. Nevertheless, our results show that, in terms of computational costs, we achieve new records when compared with the fastest 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. This significantly reduces the total `sntrup*` costs, in effect assigning higher decision-making weight to size and, most importantly, security.

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

[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.

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.[7] 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* [19] is used in this study to prevent visual distortion of the data and exclusion of readers with colour-vision deficiencies [20].

Metadata. Date: 2021.10.05. Permanent ID of this document: `a8f6fc35a5dc11da1f125b9f5225d2a9c4c5b08b`.

References

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. Status report on the second round of the NIST Post-Quantum Cryptography Standardization Process, 2020. https://csrc.nist.gov/publications/detail/nistir/8309/final.

[2] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In *USENIX Security 2016*, pages 327–343, 2016.

[3] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde, and B.-Y. Yang. Polynomial multiplication in NTRU Prime: Comparison of optimization strategies on Cortex-M4. *TCHES*, 2021(1):217–238, 2021.

[7] https://opensslntru.cr.yp.to

[4] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[5] A. Basso, J. M. B. Mera, J.-P. D'Anvers, A. Karmakar, S. S. Roy, M. V. Beirendonck, and F. Vercauteren. SABER: Mod-LWR based KEM (round 3 submission), 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[6] M. Bellare, H. Davis, and F. Günther. Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. In *EUROCRYPT 2020*, pages 3–32, 2020. https://eprint.iacr.org/2020/241.

[7] D. J. Bernstein. Multidigit multiplication for mathematicians, 2001. http://cr.yp.to/papers.html#m3.

[8] D. J. Bernstein. Batch binary Edwards. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009.

[9] D. J. Bernstein and T. Lange. Crypto horror stories, 2020. https://hyperelliptic.org/tanja/vortraege/20200206-horror.pdf.

[10] D. J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems, 2021. https://bench.cr.yp.to, accessed 28 May 2021.

[11] D. J. Bernstein and B. Yang. Fast constant-time gcd computation and modular inversion. *TCHES*, 2019(3):340–398, 2019.

[12] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang. NTRU Prime: round 3, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[13] J. Biasse and F. Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *SODA 2016*, pages 893–902, 2016.

[14] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, and D. Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *PQCrypto 2019*, pages 206–226. Springer, 2019.

[15] B. B. Brumley, S. ul Hassan, A. Shaindlin, N. Tuveri, and K. Vuojärvi. Batch binary Weierstrass. In *LATINCRYPT*, pages 364–384. Springer, 2019.

[16] J. Carmack. Latency mitigation strategies, 2013. https://web.archive.org/web/20130225013015/www.altdevblogaday.com/2013/02/22/latency-mitigation-strategies/.

[17] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, T. Saito, P. Schwabe, W. Whyte, T. Yamakawa, K. Xagawa, and Z. Zhang. NTRU: algorithm specifications and supporting documentation, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[18] C. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C. Shih, and B. Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *TCHES*, 2021(2):159–188, 2021.

[19] F. Crameri. Scientific colour maps, February 2021. https://doi.org/10.5281/zenodo.4491293.

[20] F. Crameri, G. E. Shephard, and P. J. Heron. The misuse of colour in science communication. *Nature Communications*, 11 (1):5444, Oct 2020.

[21] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, March 2021. https://www.agner.org/optimize/instruction_tables.pdf, accessed 22 March 2021.

[22] M. Kannwischer, J. Rijneveld, P. Schwabe, D. Stebila, and T. Wiggers. PQClean: clean, portable, tested implementations of postquantum cryptography, 2021. https://github.com/pqclean/pqclean.

[23] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., USA, 1st edition, 1996.

[24] P. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48:243–264, 1987.

[25] P. L. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.

[26] NIST. Guidelines for submitting tweaks for third round finalists and candidates, 2020. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/guidelines-for-sumbitting-tweaks-third-round.pdf.

[27] NIST PQC team. Guidelines for submitting tweaks for Third Round Finalists and Candidates, 2020. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LPuZKGNyQJ0/m/O6UBanYbDAAJ.

[28] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE ASSP*, 28:205–215, 1980.

[29] C. Paquin, D. Stebila, and G. Tamvada. Benchmarking post-quantum cryptography in TLS. In *PQCrypto*, pages 72–91. Springer, 2020.

[30] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005.

[31] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Inf.*, 7(4):395–398, December 1977.

[32] P. Schwabe, D. Stebila, and T. Wiggers. Post-quantum TLS without handshake signatures. In *ACM CCS*, pages 1461–1480. ACM, 2020.

[33] H. Shacham and D. Boneh. Improving SSL handshake performance via batching. In *CT-RSA 2001*, page 28–43, 2001. https://hovav.net/ucsd/papers/sb01.html.

[34] D. Stebila and M. Mosca. Post-quantum key exchange for the Internet and the Open Quantum Safe project. In *SAC*, pages 14–37. Springer, 2016.

[35] D. Stebila, S. Fluhrer, and S. Gueron. Hybrid key exchange in TLS 1.3, April 2021. https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-02.

[36] N. Tuveri and B. B. Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. In *SecDev*, pages 4–19. IEEE, 2019.

[37] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

A 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.

Review of NTRU Prime options. The NTRU Prime proposal specifies various lattice dimensions. Round-1 NTRU Prime specified only dimension 761. Round-2 NTRU Prime specified dimensions 653, 761, and 857. Round-3 NTRU Prime—which appeared after our first announcement of the OpenSSLNTRU results—also specified 953, 1013, and 1277.

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`.

Preexisting AVX2-optimized software. The official NTRU Prime "optimized C software" uses Intel AVX2 instructions and supports dimensions 653, 761, 857. Some of the code is shared across sizes except for compile-time selection of q etc. There is 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.

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 [30], 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.

B Comparing `ntruhrss`

CECPQ2's `ntruhrss701` keygen, like OpenSSLNTRU's `sntrup761`/`sntrup857` keygen, is bottlenecked by inversion. Conceptually, everything this paper does for `sntrup` can also be done for `ntruhrss`, starting with converting a batch of 32 `ntruhrss` inversions into 1 inversion plus 93 multiplications. This appendix explains two factors making this strategy less attractive for `ntruhrss` compared to `sntrup`.

First, a reasonable estimate, based on a close look at the underlying algorithms, is that there would be only about a $2\times$ speedup from `ntruhrss` keygen to batched `ntruhrss` keygen, much less than the speedup we achieve for `sntrup`.

The reason is as follows. Unbatched keygen is bottlenecked by inversion, and `ntruhrss` exploits one of its design decisions—a power-of-2 modulus, which `sntrup` 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. One can see *some* speedup from `sntrup761` to `ntruhrss701` in multiplications (because of the modulus and the lower `ntruhrss701` security level), but the ultimate difference in keygen speeds will be an order of magnitude smaller than the difference in keygen speeds before this paper.

Second, the network-traffic-vs.-security-level trade-off is worse for `ntruhrss` than for `sntrup`. For example, Table 1 shows that `ntruhrss701` sends 3.6% more traffic than `sntrup761`, despite having only 89% of the security level (Core-SVP 2^{136} vs. 2^{153}).

An existing cost model estimates that 1000 CPU cycles have the same cost as communicating a byte of data: e.g., a quad-core 3GHz server has the same cost as a 100Mbps Internet connection. An easy calculation from Table 1 concludes that higher-security `sntrup761` would still cost 1.8% below `ntruhrss701` in this model even after a $2\times$ speedup in `ntruhrss701` keygen. Making `ntruhrss` competitive with `sntrup`, accounting for the security level, would require not just this speedup but also focusing on environments where communication is more than $10\times$ cheaper.

C Barrett reduction correctness

Recalling Section 3.1.1, Barrett reduction estimates g/f_i as $h = \lfloor g/f_i \rfloor$. Then it calculates the remainder as $r = g - g \cdot \lfloor g/f_i \rfloor \cdot f_i$. We compute the difference between $(g/f_i) \cdot f_i$ and $\lfloor g/f_i \rfloor \cdot f_i$. It is the remainder r if the difference has degree less than $\deg(f_i)$.

Using the pre-computation $x^{D_g} = q_x \cdot f_i + r_x$, we have

$$g/f_i = g \cdot (x^{D_g}/f_i) \cdot 1/x^{D_g} = g \cdot (q_x - r_x/f_i) \cdot 1/x^{D_g}.$$

We compute the difference

$$\begin{aligned}&(g/f_i) \cdot f_i - \lfloor g/f_i \rfloor \cdot f_i \\ &\quad = (g \cdot q_x/x^{D_g} - \lfloor g \cdot q_x/x^{D_g} \rfloor) \cdot f_i - g(r_x/f_i) f_i/x^{D_g}.\end{aligned} \tag{3}$$

Define $h = \lfloor g \cdot q_x/x^{D_g} \rfloor = \lfloor g/f_i \rfloor$ and $l = g \cdot q_x/x^{D_g} - h$. The term $(g \cdot q_x/x^{D_g} - \lfloor g \cdot q_x/x^{D_g} \rfloor) \cdot f_i = l \cdot f_i$ in Equation 3 has degree less than $\deg(f_i)$. The other term $g(r_x/f_i) f_i/x^{D_g}$ also has degree less than $\deg(f_i)$, since $\deg(g) < D_g$ and $\deg(r_x) < \deg(f_i)$.

D More on benchmarks

Batch key-generation microbenchmarks. Table 3 shows the performance and key pair storage of `BatchKeyGen` regarding various batch sizes n.

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

Table 3: Performance of `BatchKeyGen` regarding various batch sizes n.

	n	1	2	4	8	16	32	64	128
sntrup653	amortized cost†	778 218	438 714	295 150	229 429	180 863	164 260	152 737	147 821
	latency†	778 218	877 428	1 180 600	1 835 432	2 893 808	5 256 300	9 775 160	18 921 036
	key pair storage‡	2512	5024	10 048	20 096	40 192	80 384	160 768	321 536
	memory footprint‡⋆	30 432	36 184	54 808	65 432	85 880	127 672	211 480	378 424
sntrup761	amortized cost†	819 332	567 996	351 329	242 043	181 274	156 317	147 809	141 411
	latency†	819 332	1 135 992	1 405 316	1 936 340	2 900 380	5 002 124	9 459 748	18 100 592
	key pair storage‡	2921	5842	11 684	23 368	46 736	93 472	186 944	373 888
	memory footprint‡⋆	117 200*	38 608	58 040	70 456	94 584	143 288	240 536	435 512
sntrup857	amortized cost†	1 265 056	708 104	458 562	322 352	255 815	216 618	201 173	193 203
	latency†	1 265 056	1 416 208	1 834 248	2 578 812	4 093 040	6 931 748	12 875 024	24 729 872
	key pair storage‡	3321	6642	13 284	26 568	53 136	106 272	212 544	425 088
	memory footprint‡⋆	38 648	45 520	65 176	78 840	106 392	161 216	270 912	490 336

† (Haswell cycle). ‡ (Byte). ⋆ Benchmark with `'valgrind --tool=massif --stacks=yes'`.

* The implementation uses the 'jump-div-step' optimization in [11], consuming more stack space.

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.

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.

E OpenSSL: software architecture

Section 4.1 details the part of our contributions consisting of 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. See `https://www.openssl.org/docs/man1.1.1/man7/evp.html`.

The `EVP` API, especially for public key cryptography, offers a high degree of *crypto agility*. It defines abstract cryptographic key objects, and functions that operates on them, in terms of generic operations (e.g., `EVP_PKEY_encrypt()`/`EVP_PKEY_decrypt()`). 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.