Analyzing the complexity of reference post-quantum software

Daniel J. Bernstein¹,²

¹ Department of Computer Science, University of Illinois at Chicago, USA
² Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to

Abstract. Constant-time C software for various post-quantum KEMs has been submitted by the KEM design teams to the SUPERCOP testing framework. The ref/*.c and ref/*.h files together occupy, e.g., 848 lines for ntruhps4096821, 928 lines for ntruhrss701, 1316 lines for sntrup1277, and 2613 lines for kyber1024.

It is easy to see that these numbers overestimate the inherent complexity of software for these KEMs. It is more difficult to systematically measure this inherent complexity.

This paper takes these KEMs as case studies and applies consistent rules to streamline the ref software for the KEMs, while still passing SUPERCOP’s tests and preserving the decomposition of specified KEM operations into functions. The resulting software occupies 381 lines for ntruhps4096821, 385 lines for ntruhrss701, 472 lines for kyber1024, and 478 lines for sntrup1277. This paper also identifies the external subroutines used in each case, identifies the extent to which code is shared across different parameter sets, quantifies various software complications specific to each KEM, and finds secret-dependent timings in kyber*/ref.

Keywords: post-quantum cryptography, lattice-based cryptography, software metrics

1 Introduction

The United Kingdom’s mass-surveillance agency [49] is called the “Government Communications Headquarters” (GCHQ). In 2016, GCHQ introduced a new website called the “National Cyber Security Centre” (NCSC). The GCHQ director

This work was funded by the Intel Crypto Frontiers Research Center; by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments—EXC 2092 CASA—390781972 “Cyber Security in the Age of Large-Scale Adversaries”; by the U.S. National Science Foundation under grant 1913167; and by the Taiwan’s Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

“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). Permanent ID of this document: 07279bdceceec9b085135036f27474755be6e085. Date: 2023.12.17.
later wrote [31, pages 14–15] that “Complete ownership by GCHQ was also key to making the NCSC acceptable to foreign intelligence allies”. In 2023, NCSC issued a statement [41] regarding post-quantum cryptography, in particular

- discouraging immediate deployment of post-quantum software (“operational systems should only use implementations based on the final NIST standards”),
- discouraging deployment of the highest available post-quantum security levels (“require greater processing power and bandwidth, and have larger key sizes or signatures”; “may be useful for key establishment in cases where the keys will be particularly long lived or protect particularly sensitive data that needs to be kept secure for a long period of time”),
- discouraging deployment of the post-quantum signature systems with the strongest security track records (“not suitable for general purpose use”; “the signatures are large”), and
- discouraging deployment of double encryption and double signatures (“no more security than a single post-quantum algorithm but with significantly more complexity and overhead”).

The anti-security recommendations in [41] were surrounded by non-controversial statements (e.g., “For users of commodity IT, such as those using standard browsers or operating systems, the switchover to PQC will be delivered as part of a software update and should happen seamlessly”).

None of the cost claims made in [41] were accompanied by any numbers. Most of the cost claims, such as the claim that “the signatures are large”, would have been very easy to quantify. This quantification would have helped readers compare the costs to the overall costs of their applications (see, e.g., my papers [14, Section 2] and [15])—which, presumably, would have encouraged rapid deployment in many (if not all) applications, evidently not the goal of [41].

The claim of “significantly more complexity” in [41] is different: it is shielded by a lack of literature quantifying this complexity. How complex is the software for a post-quantum KEM? How much more complex is a hybrid KEM that combines a post-quantum KEM with X25519? What about signatures?

Given that no “complexity” metric is specified, one can dismiss the claim of “significantly more complexity” as failing the scientific rule of falsifiability. The claim nevertheless appears in [41] as part of a guide to real-world decisions. It is easy to see how claims regarding complexity can be used to deter cryptographic upgrades and to influence specific choices of cryptographic mechanisms.

1.1. Assessment challenges. Let’s focus on the first question formulated above, the question of how complex post-quantum KEM software is. Software is centrally available for various post-quantum KEMs in the SUPERCOP testing framework from [18]. The real question is how to measure the complexity of that software.

There is an extensive literature on software metrics, for example as predictors of bugs; see, e.g., [46]. There are also reasons for caution in applying these metrics to cryptography:
• 2021 Blessing–Specter–Weitzner [22] evaluated vulnerability announcements (CVEs) since 2010 in OpenSSL, GnuTLS, Mozilla TLS, WolfSSL, Botan, Libgcrypt, LibreSSL, and BoringSSL, finding that “the rate of vulnerability introduction is up to three times as high in cryptographic software as in non-cryptographic software”: e.g., 1.187 CVEs per thousand lines of code added to OpenSSL, compared to 0.403 for Ubuntu.

• Cryptographic software typically replaces secret branch conditions and secret array indices with constant-time arithmetic, since branch conditions and array indices are leaked through timings. Replacing a branch with constant-time arithmetic reduces complexity in typical control-flow metrics such as “cyclomatic complexity”, but does not eliminate bugs.

• A single cryptographic function might be shipped not just as reference software but as dozens of different pieces of optimized software (see, e.g., the official Keccak code package [21]), featuring mathematical optimizations and CPU-specific optimizations such as vectorization. Hopefully any bugs here will eventually be eliminated by formal verification that the optimized software matches the reference software (see, e.g., [4] for recent verification of most of the subroutines in an optimized Kyber implementation), but presumably the complexity of the software has an influence on the verification cost, and on the cost of writing the software in the first place.

This last point suggests a split of analyses into two scenarios:

• This paper focuses on analyzing the complexity of reference software for post-quantum KEMs. The scenario here is that the application simply needs the cryptographic features provided by its selected KEM, and can afford the CPU time for the reference software for that KEM. (See, e.g., [43] and [51].)

• It would also be interesting to analyze the extra complexity of optimized software. The scenario there would be an application where the reference software is not fast enough.

The second scenario is more challenging to analyze—it depends on the target CPUs, depends on the performance targets, naturally involves more code, and raises research questions about tradeoffs between speed and code complexity, whereas existing optimized post-quantum software usually focuses purely on speed—so it makes sense to take the first scenario as an initial case study.

How complex are the reference implementations of post-quantum KEMs? As a starting point, let’s simply count lines in ref/*.c and ref/*.h in SUPERCOP. This produces tallies of, e.g.,

• 848 lines for ntruhps4096821,
• 928 lines for ntruhrss701,
• 1316 lines for sntrup1277, and
• 2613 lines for kyber1024.

All of these pieces of software were submitted to SUPERCOP by the KEM design teams, and all of them are labeled goal-constbranch and goal-constindex, meaning that they are designed to avoid secret branch conditions and secret
array indices. The kyber1024 software does not pass the TIMECOP component of SUPERCOP, but a two-line change makes it pass; see Section 6.

There are many ways to object to these line counts as (1) misleading and (2) clearly not what the ref implementations were designed to optimize. For example, kyber1024/ref/* . c has 77 multi-line comments describing inputs and outputs of functions. As another example, the largest kyber1024/ref/* . c file is fips202.c (774 lines), which implements the SHA-3 family of hash functions; for comparison, ntruhrss701 simply calls SUPERCOP’s crypto_hash_sha3256 subroutine. Counting 774 lines for hash implementations in kyber1024 and 0 for ntruhrss701 is unfair to kyber1024. On the other hand, replacing kyber1024’s fips202.c with calls to subroutines available in SUPERCOP and not tallying the subroutines would be unfair to ntruhrss701, since kyber1024 uses a wider range of SHA-3/SHAKE functions than ntruhrss701 does. (See Section 6.)

Meanwhile sntrup1277 uses crypto_hash_sha512. Is SHA-2 more complex than SHA-3, making this choice a complexity disadvantage? (See Section 3.2.) Or is it a complexity advantage, since practically all environments already provide SHA-2 (for, e.g., TLS) whereas there is a higher risk of having to add SHA-3?

1.2. Contributions of this paper. As case studies, this paper takes the four families of lattice-based KEMs mentioned above: kyber, ntruhps, ntruhrss, and sntrup. Section 2 applies consistent rules to streamline the ref software for these KEMs. Streamlining does not mean pure code-size minimization (“code golfing”): the modified software closely tracks the original software, decomposing the specified KEM operations into functions the same way that ref does.

The new software has been added to SUPERCOP under the name “compact” and passes SUPERCOP’s tests, including TIMECOP. Readers are cautioned, however, that the new software has not been verified and could easily have bugs. SUPERCOP’s tests are more thorough than many other test frameworks but do not eliminate the possibility of bugs.

Section 3 measures the resulting software, tabulating line counts (ranging from 381 lines for ntruhps4096821 through 497 lines for kyber512), further size metrics, and a list of all external subroutines such as crypto_hash_sha3256. Section 3 also measures the number of lines of code needed to merge different parameter sets. Interestingly, even though [6, Section 6.1] claims “scalability” as one of the two “unique advantages” of Kyber, it turns out that kyber* has the largest code-size differences across parameter sets.

Sections 4, 5, and 6 look more closely at various aspects of the KEM software, in particular giving quantified examples of inherent software complications specific to each family of lattice-based KEMs. (Note that it is invalid to select any particular complication as an indication of one KEM being more complex than another; these complications are merely contributing factors to total complexity.) As a spinoff, Section 4.2 identifies secret-dependent timings in kyber*/ref.

Finally, Section 7 looks at how various security and efficiency design goals for these KEMs led to these software complications.
2 Streamlining the reference software

This section specifies the rules used to convert the existing ref implementations into this paper’s compact implementations. An overarching principle behind this section’s rules is the principle of sharing any existing streamlining: if a particular type of streamlining is visible in one KEM’s software, and makes sense for other KEMs, then it should be applied to the other KEMs.

This section is organized in roughly decreasing order of code-size impact. Verifying whether this is actually decreasing order would take extra work, since this paper’s software was not written in this order.

2.1. Using external subroutines. All of ntruhps*/ref, ntruhrss*/ref, and sntrup*/ref reuse external hash subroutines, so this paper also replaces the SHA-3/SHAKE functions in kyber*/ref with calls to external hash subroutines. See Section 6 for further information on how hashing is used.

Both ntruhps*/ref and sntrup*/ref also call an external sorting subroutine, but there were no evident opportunities to use sorting in kyber*/ref and ntruhrss*/ref. Each kyber*/ref has two lines calling memcpy, but this does not appear to qualify as streamlining given the need to include string.h.

For integer types, some of the ref software uses, e.g., int16_t from stdint.h, which is more streamlined than using crypto_int16 from SUPERCOP’s crypto_int16.h, so this paper always uses stdint.h.

SUPERCOP provides various further subroutines that could have been used at a few spots in each KEM, such as crypto_int16_nonzero_mask and crypto_verify_*, but none of the ref software uses these subroutines.

2.2. One parameter set. ntruhps2048677/ref has some code that does not appear in ntruhps4096821/ref, and vice versa. sntrup*/ref instead merges code across parameter sets: almost all files are shared across parameter sets, except for an 11-line paramsmenu.h file and the 4-line api.h file required by SUPERCOP. Similarly, kyber*/ref shares all files across parameter sets except for a 48-line params.h file with one line changing across parameter sets.

This paper reports various separate measurements of the code for each parameter set. With separate measurements, code for a single parameter set is more streamlined than merged code, so this paper applies the same streamlining to each parameter set for kyber* and sntrup*. Concretely, this means eliminating code that applies only to other parameter sets, eliminating macros that control the code inclusion, and specializing api.h to 4 lines of precomputed numbers. For example, code in kyber* to support kyber90s* is removed, as is code in sntrup* to support ntruplr*.

For applications that support multiple parameter sets, it is also interesting to measure the extent to which code is shared across parameter sets. Section 3.3 reports the size of code merged across various pairs of parameter sets.

2.3. One file. A KEM can split subroutines into many *.c files, but this requires extra code in *.h files to declare each subroutine. Most of these declarations are skipped by sntrup1277/ref, which puts almost all functions
into a single `kem.c`, except for a few general-purpose utility functions for integer arithmetic, integer encoding, and integer decoding. The number of `ref/*.c` files is 5 for `sntrup1277`, 10 for `kyber1024`, and 13 for `ntruhrss701`.

This paper merges the code for each parameter set into a single `kem.c` file (plus the 4-line `api.h` file required by SUPERCOP), eliminating the need for subroutine declarations (and namespace declarations). Sometimes this eliminates multiple identical `static` subroutines: for example, `ntruhrss701/ref` has three `mod3` subroutines. Actually, one of those `mod3` subroutines is a variant, skipping some initial lines that are no-ops in context; this paper eliminates that variant.

2.4. Integer and polynomial operations. Sections 4 and 5 identify specific differences in how the `ref` code for different KEMs (and sometimes within a single KEM) carries out various operations on integers and polynomials. This paper consistently applies the more concise approach across KEMs.

2.5. Comments. There is a very long history of debates regarding the proper levels of comments in code and in separate documentation. None of the `ref` implementations consistently have (1) comments on code sections (`kyber*` and `ntru*` skip this) and (2) comments on each function (`ntru*` usually skips this; `sntrup*` sometimes skips this) and (3) comments on any potentially interesting step in a function (`kyber*` skips this; `sntrup*` usually skips this).

Code-measurement tools often say that they disregard comments. To ensure that comments do not influence any of the numbers in Section 3, this paper simply removes all comments from the source code.

2.6. Unused code. Some code turns out to be unused, before or after the other changes applied in this paper. For example, `kyber*/ref/reduce.h` defines a `MONT` macro that is not used except in a comment in `kyber*/ref/ntt.c`; `kyber*/ref/kem.h` defines a `CRYPTO_ALGNAME` macro that matters only for a NIST test program, not inside SUPERCOP; and `ntruhrss*/ref/owcpa.c` includes a `uint16_t t = 0` initializer that is followed immediately by setting `t` to `ciphertext[NTRU_CIPHERTEXTBYTES-1]`. This paper eliminates any detected unused code, although this is not necessarily comprehensive.

Some abstraction layers are, in the context of handling just one parameter set (see Section 2.2), simply renaming `X` as `Y` for some `X` and `Y`. In these cases, this paper merges the names `X` and `Y` into a single name, eliminating the renaming step. For example, `kyber1024/ref/cbd.c` has functions `poly_cbd_etal1` and `poly_cbd_etal2` that, for `kyber1024`, simply call `cbd2`, so this paper eliminates those functions in favor of calling `cbd2` directly. This paper does not merge multi-step functions into their callers, even when there is just one caller: this would go beyond renaming into changing functional decomposition.

For the same reason, this paper does not generally eliminate macros. However, some macros are redundant, and are eliminated, given the availability of SUPERCOP macros such as `crypto_kem_CIPHERTEXTBYTES`. 
2.7. Formatting. The ref software for these KEMs varies in code formatting. For example, ntruhrss701/ref/*.c puts opening braces on separate lines—

```c
for(i=0; i<NTRU_N; i++)
{
    ...
}
```

—whereas kyber1024/ref/*.c puts opening braces on the previous line (this is widely known as the “One True Brace Style”), sometimes even for functions:

```c
void ntt(int16_t r[256]) {
    ...
}
```

Streamlining is often measured by line counts, so this paper always puts opening braces on the previous line.

More broadly, this paper reformat all code with clang-format using Google style (which, among other things, always puts opening braces on the previous line), except for “SortIncludes: false” to avoid sorting #include directives (this simplifies comparisons to ref), and “ColumnLimit: 9999” to allow very long lines (otherwise the line count is sensitive to how long the variable names are and whether pointers as function parameters are expressed with array lengths). This can be viewed as too generous to kyber* since it compresses

```c
const int16_t zetas[128] = {
    -1044, -758, -359, -1517, 1493, 1422, 287, 202,
    -171,  622,  1577,  182,  962, -1202, -1474, 1468,
    573, -1325,  264,  383, -829, 1458, -1602,  130,
    -681,  1017,  732,  608, -1542,  411, -205, -1571,
    1223,  652, -552, 1015, -1293, 1491, -282, -1544,
    516,  -8, -320, -666, -1618, -1162, 126, 1469,
    -853,  -90, -271,  830, 107, -1421, -247, -951,
    -398,  961, -1508, -725,  448, -1065,  677, 1275,
    -1103,  430,  555,  843, -1251,  871, 1550,  105,
    422,  587,  177, -235, -291, -460, 1574, 1653,
    -246,  778,  1159, -147, -777, 1483,  602, 1119,
    -1590,  644, -872,  349,  418,  329,  156,  -75,
    817,  1097,  603,  610, 1322, -1285, -1465,  384,
    -1215, -136, 1218, -1335, -874,  220, -1187, -1659,
    -1185, -1530, -1278,  794, -1510, -854,  870,  478,
    -108,  308,  996,  991,  958, -1460, 1522, 1628
};
```

into a single 787-character line, but this is an exceptional case. The byte counts in Section 3 show that the average line lengths are similar across KEMs.

This paper removes all blank lines inside functions, leaves only one blank line between functions, and collects macros at the top of each file with no blank lines between macros.
3 Measurements of the streamlined software

This section applies various metrics to the compact software produced by the rules in Section 2; lists the external subroutines used by this software; and tallies the number of lines used when software for different parameter sets is merged within the same KEM family.

This tables in this section were produced by various scripts attached to this PDF, except that Table 3.2.1 was assembled by hand.

3.1. Metrics. This section uses the following metrics for the *.c and *.h files, in some cases computed by the lizard tool from [53]:

- “bytes”: total number of bytes;
- “bytesw”: total number of bytes after replacement of each alphanumeric stretch (including underscore) with a single letter;
- “tokens”: total number of tokens reported by lizard across all functions;
- “bytesz”: total number of bytes after gzip -9 compression of each file;
- “byteswz”: total number of bytes after gzip -9 compression of the results of replacing each alphanumeric stretch;
- “lines”: total number of lines;
- “loc”: number of lines of code (nloc) reported by lizard (this excludes blank lines between functions, macros outside functions, etc.);
- “funloc”: total of the number of lines of code reported by lizard within functions;
- “cyc”: total across functions of the per-function cyclomatic complexity reported by lizard (i.e., number of functions plus number of branches);
- “funs”: number of functions reported by lizard.

Table 3.1.1 tallies the results of applying these numerical metrics to the compact software. Also, to illustrate the effect of (not) streamlining, Table 3.1.2 tallies the results of applying the same numerical metrics to the original ref software.

3.2. Measuring external subroutines. Table 3.2.1 lists the external subroutines called by the compact software.

The complexity of these subroutines is not accounted for in the metrics from Table 3.1.1. This complexity is of interest for environments where these subroutines are not already available. More broadly, this complexity should be weighted by the extent to which these subroutines are shared by other applications.

Table 3.2.2 reports metrics for this paper’s compact versions of various subroutines. These implementations are streamlined as in Section 2, starting from crypto_hashblocks/sha512/compact4, crypto_sort/int32/portable3, and crypto_sort/uint32/useint32, plus crypto_xof/shake256/tweet in libmceliece [17], which in turn is based on TweetFIPS202 [19].

There are two exceptionally long lines in sha512/compact: 1875 characters for a round array, and 421 characters for an iv array. There are also 8 lines of macros carrying out computations in sha512/compact; if these were rewritten as functions then formatting would cost 7 lines.
Table 3.1.1. Numerical measurements of this paper’s compact software for each parameter set for each KEM. See text for description of the columns.

<table>
<thead>
<tr>
<th>KEM</th>
<th>bytes</th>
<th>bytesw</th>
<th>tokens</th>
<th>bytesz</th>
<th>byteswz</th>
<th>lines</th>
<th>loc</th>
<th>funloc</th>
<th>cyc</th>
<th>funs</th>
</tr>
</thead>
<tbody>
<tr>
<td>kyber512</td>
<td>17176</td>
<td>9110</td>
<td>4645</td>
<td>4169</td>
<td>1377</td>
<td>497</td>
<td>425</td>
<td>411</td>
<td>116</td>
<td>49</td>
</tr>
<tr>
<td>kyber768</td>
<td>16403</td>
<td>8629</td>
<td>4395</td>
<td>4092</td>
<td>1362</td>
<td>469</td>
<td>400</td>
<td>386</td>
<td>110</td>
<td>46</td>
</tr>
<tr>
<td>kyber1024</td>
<td>16549</td>
<td>8773</td>
<td>4471</td>
<td>4121</td>
<td>1371</td>
<td>472</td>
<td>403</td>
<td>389</td>
<td>114</td>
<td>46</td>
</tr>
<tr>
<td>ntruhps2048509</td>
<td>13151</td>
<td>7279</td>
<td>3911</td>
<td>2775</td>
<td>1066</td>
<td>385</td>
<td>333</td>
<td>325</td>
<td>93</td>
<td>34</td>
</tr>
<tr>
<td>ntruhps2048577</td>
<td>13152</td>
<td>7279</td>
<td>3911</td>
<td>2775</td>
<td>1066</td>
<td>385</td>
<td>333</td>
<td>325</td>
<td>93</td>
<td>34</td>
</tr>
<tr>
<td>ntruhps4096821</td>
<td>12856</td>
<td>7051</td>
<td>3911</td>
<td>2775</td>
<td>1066</td>
<td>385</td>
<td>333</td>
<td>325</td>
<td>93</td>
<td>34</td>
</tr>
<tr>
<td>ntruhrss701</td>
<td>13322</td>
<td>7441</td>
<td>4232</td>
<td>2864</td>
<td>1144</td>
<td>385</td>
<td>333</td>
<td>325</td>
<td>93</td>
<td>34</td>
</tr>
<tr>
<td>sntrup653</td>
<td>13306</td>
<td>8630</td>
<td>4660</td>
<td>3205</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
<tr>
<td>sntrup761</td>
<td>13308</td>
<td>8630</td>
<td>4660</td>
<td>3205</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
<tr>
<td>sntrup857</td>
<td>13308</td>
<td>8630</td>
<td>4660</td>
<td>3205</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
<tr>
<td>sntrup953</td>
<td>13308</td>
<td>8630</td>
<td>4660</td>
<td>3205</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
<tr>
<td>sntrup1013</td>
<td>13309</td>
<td>8630</td>
<td>4660</td>
<td>3208</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
<tr>
<td>sntrup1277</td>
<td>13309</td>
<td>8630</td>
<td>4660</td>
<td>3207</td>
<td>1266</td>
<td>478</td>
<td>424</td>
<td>415</td>
<td>125</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 3.1.2. Numerical measurements of the existing ref software for each parameter set for each KEM. See text for description of the columns.

<table>
<thead>
<tr>
<th>KEM</th>
<th>bytes</th>
<th>bytesw</th>
<th>tokens</th>
<th>bytesz</th>
<th>byteswz</th>
<th>lines</th>
<th>loc</th>
<th>funloc</th>
<th>cyc</th>
<th>funs</th>
</tr>
</thead>
<tbody>
<tr>
<td>kyber512</td>
<td>85105</td>
<td>46836</td>
<td>9488</td>
<td>19479</td>
<td>5902</td>
<td>2613</td>
<td>1270</td>
<td>1064</td>
<td>180</td>
<td>79</td>
</tr>
<tr>
<td>kyber768</td>
<td>85105</td>
<td>46836</td>
<td>9488</td>
<td>19479</td>
<td>5902</td>
<td>2613</td>
<td>1270</td>
<td>1064</td>
<td>180</td>
<td>79</td>
</tr>
<tr>
<td>kyber1024</td>
<td>85105</td>
<td>46836</td>
<td>9488</td>
<td>19479</td>
<td>5902</td>
<td>2613</td>
<td>1270</td>
<td>1064</td>
<td>180</td>
<td>79</td>
</tr>
<tr>
<td>ntruhps2048509</td>
<td>26369</td>
<td>14786</td>
<td>6138</td>
<td>9372</td>
<td>3767</td>
<td>918</td>
<td>589</td>
<td>515</td>
<td>98</td>
<td>38</td>
</tr>
<tr>
<td>ntruhps2048577</td>
<td>26370</td>
<td>14786</td>
<td>6138</td>
<td>9373</td>
<td>3767</td>
<td>918</td>
<td>589</td>
<td>515</td>
<td>98</td>
<td>38</td>
</tr>
<tr>
<td>ntruhps4096821</td>
<td>23247</td>
<td>12416</td>
<td>4730</td>
<td>8846</td>
<td>3600</td>
<td>848</td>
<td>525</td>
<td>451</td>
<td>89</td>
<td>38</td>
</tr>
<tr>
<td>ntruhrss701</td>
<td>27069</td>
<td>15372</td>
<td>6254</td>
<td>9636</td>
<td>3959</td>
<td>928</td>
<td>585</td>
<td>512</td>
<td>96</td>
<td>37</td>
</tr>
<tr>
<td>sntrup653</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8128</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
<tr>
<td>sntrup761</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8128</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
<tr>
<td>sntrup857</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8130</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
<tr>
<td>sntrup953</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8130</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
<tr>
<td>sntrup1013</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8130</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
<tr>
<td>sntrup1277</td>
<td>26426</td>
<td>15114</td>
<td>6441</td>
<td>8128</td>
<td>3163</td>
<td>1316</td>
<td>744</td>
<td>693</td>
<td>161</td>
<td>66</td>
</tr>
</tbody>
</table>

Further hashing subroutines are needed for kyber* and are not listed in Table 3.2.2. These subroutines would be able to share most of their code with the sha3256 code.

### 3.3. Line counts for merging parameter sets

Table 3.3.1 lists, for various pairs of parameter sets, the number of lines for a merged version of kem.c across the pairs (not considering api.h). This table covers all pairs of kyber* parameter sets, all pairs of ntruhps* parameter sets, and all pairs of sntrup* parameter sets.

The table also covers pairs crossing ntruhps* and ntruhrss*, given that [24, page 4] says “We have unified all aspects of the designs except for the use of...
Table 3.2.1. List of external subroutines called by this paper’s compact software for each KEM beyond memcpy (twice in kyber*), crypto_declassify (once in kyber* and once in sntrup*), and randombytes. For kyber*, the “four Sponge* subroutines” listed in the table are described in Section 6.

<table>
<thead>
<tr>
<th>subroutine</th>
<th>bytes</th>
<th>bytesw</th>
<th>tokens</th>
<th>bytesz</th>
<th>lines</th>
<th>loc</th>
<th>funloc</th>
<th>cyc</th>
<th>funs</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash/sha512</td>
<td>4466</td>
<td>1902</td>
<td>636</td>
<td>1916</td>
<td>379</td>
<td>65</td>
<td>49</td>
<td>45</td>
<td>17</td>
</tr>
<tr>
<td>hash/sha3256</td>
<td>1778</td>
<td>1401</td>
<td>715</td>
<td>736</td>
<td>374</td>
<td>67</td>
<td>59</td>
<td>57</td>
<td>23</td>
</tr>
<tr>
<td>sort/int32</td>
<td>754</td>
<td>564</td>
<td>264</td>
<td>393</td>
<td>229</td>
<td>31</td>
<td>28</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>sort/uint32</td>
<td>846</td>
<td>634</td>
<td>304</td>
<td>416</td>
<td>239</td>
<td>33</td>
<td>30</td>
<td>28</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 3.2.2. Numerical measurements of this paper’s compact software for some of the subroutines used in KEM software. See text for description of the columns.

fixed-weight sampling”. Beware that these pairs are not directly comparable to the others. The table does not cover kyber* vs. kyber90s*, or sntrup* vs. ntruhps*.

Recall from Section 1 that [6, Section 6.1] claims “scalability” as one of the two “unique advantages” of Kyber. The full quote is as follows:

**Scalability:** Switching from one Kyber parameter set to another only requires changing the matrix dimension (i.e., a `#define` in most C implementations), the noise sampling, and the rounding of the ciphertext via different parameters to the `Compress_q` function.

However, Table 3.3.1 shows that merging the streamlined code for kyber512 and kyber1024 increases the line counts from 493 and 468 to 574, jumps of 81 lines and 106 lines respectively, whereas the maximum jump for ntruhps is 33 lines and the jump for sntrup is just 6 lines.

Part of the issue here is that the kyber512 code uses specific noise-sampling functions not used in kyber768 and kyber1024, namely cbd3 (14 lines) and, inside that, load24_littleendian (6 lines). A larger part of the issue is how parameter sets vary in functions for encoding and decoding; see Section 5. It is also interesting to observe that the Kyber software from [4] supports only kyber512 and kyber768.

### 4 Subroutines for arithmetic

In the 1998 NTRU cryptosystem [33], a ciphertext has the form $Gb + d$. Here $b, d$ are secret integer vectors with small entries, and $G$ is a public linear
<table>
<thead>
<tr>
<th>KEM 1</th>
<th>KEM 2</th>
<th>lines 1</th>
<th>lines 2</th>
<th>lines merged</th>
</tr>
</thead>
<tbody>
<tr>
<td>kyber512</td>
<td>kyber768</td>
<td>493</td>
<td>465</td>
<td>512</td>
</tr>
<tr>
<td>kyber512</td>
<td>kyber1024</td>
<td>493</td>
<td>468</td>
<td>574</td>
</tr>
<tr>
<td>kyber768</td>
<td>kyber1024</td>
<td>465</td>
<td>468</td>
<td>531</td>
</tr>
<tr>
<td>ntruhps2048509</td>
<td>ntruhps2048677</td>
<td>389</td>
<td>389</td>
<td>393</td>
</tr>
<tr>
<td>ntruhps2048509</td>
<td>ntruhps4096821</td>
<td>389</td>
<td>377</td>
<td>410</td>
</tr>
<tr>
<td>ntruhps2048677</td>
<td>ntruhps4096821</td>
<td>389</td>
<td>377</td>
<td>410</td>
</tr>
<tr>
<td>ntruhps2048509</td>
<td>ntruhrs701</td>
<td>389</td>
<td>381</td>
<td>476</td>
</tr>
<tr>
<td>ntruhps2048677</td>
<td>ntruhrs701</td>
<td>389</td>
<td>381</td>
<td>476</td>
</tr>
<tr>
<td>ntruhps4096821</td>
<td>ntruhrs701</td>
<td>377</td>
<td>381</td>
<td>448</td>
</tr>
<tr>
<td>sntrup653</td>
<td>sntrup761</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup653</td>
<td>sntrup857</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup653</td>
<td>sntrup953</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup653</td>
<td>sntrup1013</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup653</td>
<td>sntrup1277</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup761</td>
<td>sntrup857</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup761</td>
<td>sntrup953</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup761</td>
<td>sntrup1013</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup761</td>
<td>sntrup1277</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup857</td>
<td>sntrup953</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup857</td>
<td>sntrup1013</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup857</td>
<td>sntrup1277</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup953</td>
<td>sntrup1013</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup953</td>
<td>sntrup1277</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
<tr>
<td>sntrup1013</td>
<td>sntrup1277</td>
<td>474</td>
<td>474</td>
<td>480</td>
</tr>
</tbody>
</table>

Table 3.3.1. Line counts for merges of compact/kem.c across two parameter sets. The 4 lines in each compact/api.h are not included here.

transformation on vectors of integers mod \( q \). The modulus \( q \) is a cryptosystem parameter.

All of the KEMs considered in this paper reuse the same basic structure, but with differences in the details. This section looks at various details that matter for code complexity.

4.1. Modular reduction. In ntru*, the modulus \( q \) is chosen as a power of 2, and reduction modulo \( q \) is simply a mask in ntru*/ref:

```c
#define MODQ(X) ((X) & (NTRU_Q - 1))
```

However, there is also arithmetic modulo 3, which ntru*/ref carries out as in Figure 4.1.1. The first few lines reduce \( r \) to the range \{0, 1, 2, 3, 4, 5\}. The last line uses various logic operations to select either \( r - 3 \) or \( r \), after a two-complement shift \( \gg 15 \) to convert negative integers into \( -1 \) and nonnegative integers into 0. (The C language does not guarantee two-complement arithmetic, but SUPERCOP always sets the compiler’s \(-fwrapv\) option, which guarantees two-complement arithmetic.) One can easily test that this function works for all possible inputs.
static uint16_t mod3(uint16_t a)
{
    uint16_t r;
    int16_t t, c;
    r = (a >> 8) + (a & 0xff); // r mod 255 == a mod 255
    r = (r >> 4) + (r & 0xf); // r' mod 15 == r mod 15
    r = (r >> 2) + (r & 0x3); // r' mod 3 == r mod 3
    t = r - 3;
    c = t >> 15;
    return (c&r) ^ (~c&t);
}

Fig. 4.1.1. Example of a modular-reduction function from ntru*/ref, reducing a 16-bit input mod 3.

In sntrup*, the modulus $q$ is chosen as a prime (4591 for sntrup653, for example, and 7879 for sntrup1277) rather than as a power of 2; there is also arithmetic modulo 3. The sntru*/ref code includes a general-purpose int32_mod_uint14 function used for reduction mod $q$ and reduction mod 3.

In kyber*, there is arithmetic modulo the prime $q = 3329$ for every parameter set (and no arithmetic modulo 3). There is a reduction function in kyber*/ref that reduces 16-bit inputs modulo $q$ (with outputs between $-(q-1)/2$ and $(q-1)/2$) by multiplying by an approximation to $2^{26}/q$:

```c
int16_t barrett_reduce(int16_t a) {
    int16_t t;
    const int16_t v = ((1L << 26) + KYBER_Q/2)/KYBER_Q;
    t = ((int32_t)v*a + (1<<25)) >> 26;
    t *= KYBER_Q;
    return a - t;
}
```

This is more concise than int32_mod_uint14 (which uses similar ideas but allows an unnecessarily wide input range) or mod3, so this paper applies the same streamlining across all the KEMs (also rearranging barrett_reduce to be more concise). The F3_freeze function in sntru*/compact does

```
return x - 3 * ((10923 * x + 16384) >> 15);
```

to reduce modulo 3 with inputs between $-2^{14}$ and $2^{14} - 1$ and outputs in $\{-1, 0, 1\}$. The mod3 function in ntru*/compact does

```
return x - 3 * ((10923 * x) >> 15);
```
to reduce modulo 3 with inputs between 0 and $2^{15} - 1$ and outputs in \{0, 1, 2\}. The \texttt{Fq\_freeze} function in \texttt{sntrup*/compact} does

\begin{verbatim}
const int32_t q16 = (0x10000 + q / 2) / q;
const int32_t q20 = (0x100000 + q / 2) / q;
const int32_t q28 = (0x10000000 + q / 2) / q;
x -= q * ((q16 * x) >> 16);
x -= q * ((q20 * x) >> 20);
return x - q * ((q28 * x + 0x80000000) >> 28);
\end{verbatim}

to reduce modulo odd $q < 2^{13}$ with inputs between $-2q^2$ and $2q^2$ and outputs in \{-$(q - 1)/2$, \ldots, $(q - 1)/2$\}.

4.2. Protecting against timing attacks. The reader might be wondering why the reference code does not simply use C’s built-in division and mod operators (“/” and “\%”). The usual answer is as follows:

- Compilers often convert these operators directly into the CPU’s division instructions—and division instructions typically take variable time, perhaps leaking secret information to attackers through timing. (Compilers might instead convert divisions into multiplication instructions, but one can’t rely on this happening. For example, testing various recent versions of \texttt{gcc}, such as version 11.4.0 in current Ubuntu LTS, shows that some optimization options convert divisions into multiplication instructions, but also shows that the -Os option for size optimization produces division instructions.)
- Consequently, all of the KEM software avoids these operators—except for computations on public data, such as the \texttt{kyber*/ref} computation of $v$ displayed in Section 4.1.

But is it actually true that KEM software uses these operators only for public data? Scanning for “/KYBER\_Q” in \texttt{kyber*/ref/*} finds some divisions where the numerator is a run-time variable, not just a cryptosystem parameter. In at least one case, this variable is derived from secrets: \texttt{indcpa\_dec} combines the secret key with a ciphertext and then calls \texttt{poly\_tomsg} (see Section 5.4), which, when the work for this paper began, had a line

\begin{verbatim}
t = (((t << 1) + KYBER\_Q/2)/KYBER\_Q) & 1;
\end{verbatim}

dividing secret results by $q$. (The line counts and other measurements reported in this paper are after patches to remove this division.)

Checking (not comprehensively) Kyber implementations listed in [8] shows that the same division was copied into at least [39, “kyber.cpp”], [5, “poly.rs”], [26, “Poly.java"], [38, “poly.go”], and [50, “kyber512.js”, “kyber768.js”, and “kyber1024.js"], although in the case of [50] the choice of JavaScript raises larger questions about constant-time behavior; see generally [47].

There are many tools available to check for timing variations. See [37] for a survey. Some tools, such as \texttt{saferewrite} from [11], check for division instructions. But most tools don’t, and TIMECOP doesn’t, and in any case the
tools don’t help if they aren’t used. As noted in Section 1, the submitted kyber* code doesn’t even pass TIMECOP.

This paper’s investigation led to the discovery of this variable-time division on 14 December 2023 and an announcement on 15 December 2023. It turned out that this division had been eliminated in the official Kyber software repository two weeks earlier, with credit to Goutam Tamvada, Karthikeyan Bhargavan, and Franziskus Kiefer—without a vulnerability announcement, and without notice to downstream projects such as [39], [5], [26], [38], [50], and SUPERCOP.

In response to the 15 December 2023 announcement, the maintainer of the official Kyber software asked whether there was in fact a time variation “on any particular CPU” for the range of numerators in this division (namely \((q-1)/2 = 1664\) through \(5(q-1) = 8320\)). The answer is yes. For example, AMD Zen 2 takes an extra cycle when the numerator is 8192 or larger; and SiFive U74 (RISC-V) takes extra cycles starting at 4096 and at 8192. CacheBleed [52] is an example of a timing attack exploiting single-cycle variations; the only safe presumption is that this Kyber division is also exploitable.

Division is not the only potential issue. The C language does not guarantee that any instructions take constant time. The ref code assumes, for example, that multiplication of an integer type such as int32_t takes constant time; C compilers typically compile each integer multiplication to a single CPU multiplication instruction; but some CPUs have variable-time multipliers. See, e.g., [29] and [44]. Eliminating variable-time multiplications is outside the scope of this paper.

4.3. Concise polynomial multiplication. In ntru*, the multiplication of \(G\) by \(b\), as part of building a ciphertext \(Gb + d\), is a multiplication of two polynomials mod \(x^n - 1\) (where \(n\) is a parameter), where the polynomial coefficients are integers mod \(q\). The code for this in ntru*/ref is as follows:

```c
void poly_Rq_mul(poly *r, const poly *a, const poly *b)
{
    int k,i;
    for(k=0; k<NTRU_N; k++)
    {
        r->coeffs[k] = 0;
        for(i=1; i<NTRU_N-k; i++)
            r->coeffs[k] += a->coeffs[k+i] * b->coeffs[NTRU_N-i];
        for(i=0; i<k+1; i++)
            r->coeffs[k] += a->coeffs[k-i] * b->coeffs[i];
    }
}
```

The indices in the first \(i\) loop incorporate reduction modulo \(x^n - 1\). Reductions mod \(q\) are delayed until ciphertext encoding (Section 5.2).

In sntrup*, \(x^n - 1\) is replaced with \(x^p - x - 1\), so the coefficient of \(x^{p+k}\) has to be added to the coefficients of both \(x^k\) and \(x^{k+1}\). The multiplication code in sntrup*/ref carries out this reduction as a separate step, but reduces
mod $q$ in each arithmetic operation. Delaying the reduction until the end of the multiplication is more concise. Here is $Rq\_mult\_small$ in $sntrup*/compact$:

```c
static void Rq_mult_small(Fq *h, const Fq *f, const small *g) {
  int32_t fg[p + p - 1];
  int i, j;
  for (i = 0; i < p + p - 1; ++i) fg[i] = 0;
  for (i = 0; i < p; ++i)
    for (j = 0; j < p; ++j)
      fg[i + j] += f[i] * (int32_t)g[j];
  for (i = p; i < p + p - 1; ++i)
    fg[i - p] += fg[i];
  for (i = p; i < p + p - 1; ++i)
    fg[i - p + 1] += fg[i];
  for (i = 0; i < p; ++i)
    h[i] = Fq_freeze(fg[i]);
}
```

This is 10 lines; $poly_{Rq\_mul}$ in $ntru*/compact$ is shorter, 8 lines.

In $Rq\_mult\_small$, the $f$ coefficients are between $-q/2$ and $q/2$, and the $g$ coefficients are between $-1$ and $1$, so the polynomial product $fg$ has coefficients between $-pq/2$ and $pq/2$, or between $-3pq/2$ and $3pq/2$ after reduction (see [16, Theorem 1] for better bounds), safely inside the range $-2q^2$ through $2q^2$.

Both $ref$ and $compact$ rely on similar range calculations for $R3\_mult$ in $sntrup*$, and for $poly_{S3\_mul}$ in $ntru*$. Internally, $poly_{S3\_mul}$ is only 4 lines, reusing $poly_{Rq\_mul}$ (and taking advantage of the fact that $poly_{Rq\_mul}$ does not actually reduce mod $q$), while $R3\_mult$ is another 10 lines. It would be possible to similarly merge $R3\_mult$ and $Rq\_mult\_small$, either through macro-based templates (a form of streamlining that none of the $ref$ implementations use) or through eliminating the small type, but this would deviate from the functional decomposition in $ref$.

### 4.4. NTTs and matrices

The multiplication code in $kyber*/compact$ is longer than in $ntru*/compact$ or $sntrup*/compact$, for two reasons.

First, polynomial multiplication in $kyber*/compact$ transforms each of the input polynomials modulo $x^{256}+1$ to “NTT domain” (13 lines for $ntt$, and 1 long line for the $zetas$ array quoted in Section 2), carries out “base multiplications” in NTT domain (4 lines for $basemul$), and transforms the product back from NTT domain (15 lines for $invntt$).

NTTs are used the same way in $kyber*/ref$; but why use NTMs when other multipliers are more concise? The answer comes from an interesting feature of this KEM family: namely, ciphertexts et al. are sent in NTT domain. This limits the implementor’s choices of multiplication algorithms.

As an example of this limit, consider [1], which has been listed since at least 2021 on the Kyber page [7] as one of the “third-party implementations of Kyber”. The paper [1] uses an existing big-integer multiplier on an SLE 78 smart card to multiply polynomials, and reports “Kyber768 key generation in 79.6 ms, encapsulation in 102.4 ms and decapsulation in 132.7 ms”. However, a closer look shows that the cryptosystem in [1] is actually “not interoperable with Kyber”, in particular because “Kyber explicitly requires the usage of the
Number Theoretic Transform (NTT), which we cannot realise efficiently with our approach.

The same limit rules out the possibility of kyber*/compact having a multiplier as simple as the multipliers in ntru*/compact or sntrup*/compact. Having ciphertexts sent in NTT domain also removes multiplication as an abstraction layer: there are 10 lines calling ntt and invntt in each kyber*/compact.

The second reason that the multiplication code in kyber*/compact is longer is that \( b \) is actually a length-\( k \) vector of polynomials modulo \( x^{256} + 1 \), and \( G \) is actually a \( k \times k \) matrix of polynomials modulo \( x^{256} + 1 \), where \( k \) is 2 or 3 or 4 for kyber512 or kyber768 or kyber1024 respectively. The resulting polyvec abstraction layer includes 34 lines (2 of which are calls to ntt and invntt mentioned above) in functions for encoding, decoding, NTT, inverse NTT, dot products, reduction, and addition. There are further length-\( k \) loops for, e.g., matrix handling. Loops of length \( k \) end up appearing 17 times in each kyber*/compact (and 19 times in each kyber*/ref).

It is difficult to come up with a total line count for multiplication in the kyber* software because components of multiplication are spread through so many different functions, but 75 lines are mentioned above in ntt, zetas, basemul, invntt, the calls to those functions, and the polyvec abstraction layer.

4.5. Polynomial inversion. In sntrup*/compact, R3_recip (36 lines) inverts a polynomial mod \( x^p - x - 1 \) with coefficients mod 3, and Rq_recip3 (36 lines) inverts a polynomial mod \( x^p - x - 1 \) with coefficients mod \( q \), also multiplying the result by 3.

Inversion is more complicated in ntru*/compact because \( q \) is not prime: there are functions poly_S3_inv (35 lines) and poly_R2_inv (34 lines) working mod 3 and mod 2, but there is also poly_Rq_inv (5 lines), built from poly_R2_inv and poly_R2_inv_to_Rq_inv (14 lines). There is no polynomial inversion in kyber*.

The four inversion functions with prime moduli (R3_recip and Rq_recip3 for sntrup*; poly_S3_inv and poly_R2_inv for ntru*) are all very similar, using the inversion algorithm from [20]. As in Section 4.3, it would be possible to merge these.

4.6. Further arithmetic. This section is not meant to tally all arithmetic operations in these KEMs. For example, each ntru* has a poly_lift function, which is 27 lines in ntruhrss*/compact and 5 lines in ntruhps*/compact.

5 Subroutines for encoding and decoding

These KEMs use various types of conversions between byte strings and vectors of integers in various ranges:

- Vectors of small integers are saved inside secret keys. See Section 5.1.
- Vectors of integers mod \( q \) are communicated as public keys and as ciphertexts. See Section 5.2.
- Byte strings are converted to integers used in noise generation. See Section 5.3. This is only decoding, not encoding.
• For kyber*, a 32-byte message is converted to and from a vector of 256 integers in \{0, (q + 1)/2\}, where “from” rounds other integers mod \(q\) to 0 or to \((q + 1)/2\). See Section 5.4.

Not all of the details are required for interoperability. In particular, changing secret-key formats would preserve interoperability; also, for ntru* and sntrup*, changing noise-decoding methods would preserve interoperability. Investigating alternatives could be interesting but is outside the scope of this paper.

5.1. Encoding and decoding small integers. For kyber*, secret vectors have entries in \{-3, -2, -1, 0, 1, 2, 3\} (and more specifically in \{-2, -1, 0, 1, 2\} for kyber768 and kyber1024). However, the vectors are encoded in secret keys as integers mod \(q = 3329\) without regard to their smallness. Each integer mod \(q\) is encoded as 12 bits, and 2 integers are packed into 3 bytes. In kyber*/compact, encoding and decoding of 256 integer coefficients are handled by poly_tobytes (12 lines) and poly_frombytes (7 lines); see also Section 4.4 regarding the matrix layer on top of the polynomial layer.

For sntrup*, secret vectors have entries in \{-1, 0, 1\}. Each integer is encoded as 2 bits, and 4 integers are packed into a byte. In sntrup*/compact, encoding and decoding of \(p\) small integer coefficients are handled by Small_encode (9 lines) and Small_decode (8 lines).

For ntru*, 5 elements of \{0, 1, 2\} are packed into a byte in radix 3 as \(v_0 + 3v_1 + 9v_2 + 27v_3 + 81v_4\). In ntru*/compact, encoding and decoding of \(n\) small integer coefficients are handled by poly_S3_tobytes (13 lines) and poly_S3_frombytes (19 lines). The ntru*/ref code for this is larger since various loops are unrolled; this paper consistently rolls loops for conciseness.

5.2. Encoding and decoding big integers. For ntru*, public keys and ciphertexts are vectors of \(n - 1\) integers mod \(q\); recall that \(q\) is a power of 2, such as 2048 for ntruhps2048677. Each integer is encoded as \(\log_2 q\) bits, so each stretch of 8 integers fits into \(\log_2 q\) bytes. The code for handling 8 integers is unrolled in ntruhps2048*/ref, producing the 47-line encoding function in Figure 5.2.1 and a 31-line decoding function.

The ntruhrss*/ref code is longer: it has \(q = 8192\), with 13 bytes at a time rather than 11. The ntruhps4096*/ref code is shorter: it has \(q = 4096\), and packs each 2 integers into 3 bytes. In ntru*/compact, poly_Sq_tobytes (see Figure 5.2.2) and poly_Sq_frombytes loop over bits and each take just 5 lines. One line in poly_Sq_tobytes is long enough to be split into two lines in Figure 5.2.2, but a large part of the length comes from the length of macro names, illustrating Section 2.7’s rationale for allowing long lines.

For sntrup*, a public key is a vector of \(p\) integers mod \(q\), where again \(q\) depends on the parameter set but now \(q\) is a prime rather than a power of 2. A ciphertext is a vector of \(p\) integers rounded to multiples of 3, effectively an integer mod \((q + 2)/3\) rather than mod \(q\). There is a 31-line general-purpose Encode function that uses multiplications to encode a sequence of integers for any specified moduli, and there is a 45-line general-purpose Decode function; on
void poly_Sq_tobytes(unsigned char *r, const poly *a)
{
    int i, j;
    uint16_t t[8];

    for(i=0; i<NTRU_PACK_DEG/8; i++)
    {
        for(j=0; j<8; j++)
            t[j] = MODQ(a->coeffs[8*i+j]);

        r[11 * i + 0] = (unsigned char) (t[0] & 0xff);
        r[11 * i + 1] = (unsigned char) ((t[0] >> 8) | ((t[1] & 0x1f) << 3));
        r[11 * i + 3] = (unsigned char) ((t[2] >> 2) & 0xff);
        r[11 * i + 7] = (unsigned char) ((t[5] >> 1) & 0xff);
    }

    for(j=0; j<NTRU_PACK_DEG-8*i; j++)
        t[j] = MODQ(a->coeffs[8*i+j]);
    for(; j<8; j++)
        t[j] = 0;

    switch(NTRU_PACK_DEG&0x07)
    {
        // cases 0 and 6 are impossible since 2 generates (Z/n)* and
        // p mod 8 in {1, 7} implies that 2 is a quadratic residue.
        case 4:
            r[11 * i + 0] = (unsigned char) (t[0] & 0xff);
            r[11 * i + 1] = (unsigned char) ((t[0] >> 8) | ((t[1] & 0x1f) << 3));
            r[11 * i + 3] = (unsigned char) ((t[2] >> 2) & 0xff);
            break;
        case 2:
            r[11 * i + 0] = (unsigned char) (t[0] & 0xff);
            r[11 * i + 1] = (unsigned char) ((t[0] >> 8) | ((t[1] & 0x1f) << 3));
            break;
    }
}

Fig. 5.2.1. Example of an encoding function from ntruhps2048*/ref, packing \(n-1\) 11-bit integers into bytes. Compare Figure 5.2.2.
static void poly_Sq_tobytes(unsigned char *r, const poly *a) {
    int i;
    for (i = 0; i < crypto_kem_PUBLICKEYBYTES; i++) r[i] = 0;
    for (i = 0; i < NTRU_LOGQ * NTRU_PACK_DEG; i++)
        r[i / 8] |= (1 & (a->coeffs[i / NTRU_LOGQ] >> (i % NTRU_LOGQ))) << (i % 8);
}

Fig. 5.2.2. Example of an encoding function from ntruhps2048*/compact, packing \( n-1 \) 11-bit integers into bytes. The last loop is broken into two lines for display here. Compare Figure 5.2.1.

The top of these are four 7-line functions for encoding and decoding of vectors mod \( q \) and of rounded vectors mod \( q \).

For \texttt{kyber*}, there are three different formats. There is a format for the public key, handled by the same \texttt{poly_tobytes} and \texttt{poly_frombytes} as in Section 5.1. There is a format for part of the ciphertext, rounding to 10 bits for \texttt{kyber512} and \texttt{kyber768} or 11 bits for \texttt{kyber1024} and then packing the resulting integers into bytes; \texttt{kyber*/compact} uses 16 lines for \texttt{polyvec_compress}, and 12 lines for \texttt{polyvec_decompress}. There is also a format for another part of the ciphertext, rounding to 4 bits for \texttt{kyber512} and \texttt{kyber768} or 5 bits for \texttt{kyber1024}; \texttt{kyber*/compact} uses 12 or 14 lines respectively for \texttt{poly_compress} (not to be confused with \texttt{polyvec_compress}), and 8 or 9 lines respectively for \texttt{poly_decompress}.

5.3. Decoding for noise generation. In \texttt{ntruhps*/compact}, there is a function \texttt{sample_fixed_type} (11 lines) that generates a secret vector with entries 0, 1, 2 as follows: decode an array of secret bytes into \( n-1 \) integers, each integer having 30 bits; convert each integer \( i \) into \( 4i + 1 \) in the first \( q/16-4 \) positions, \( 4i+2 \) in the next \( q/16-4 \) positions, and \( 4i \) in the remaining positions; sort the array; and extract the bottom 2 bits at each position. This is another case where the \texttt{ref} code is strikingly less concise, unrolling the conversion of 15 bytes into 4 integers.

Sorting is used similarly in \texttt{sntrup*/compact} (and not in \texttt{ntruhrss*} or \texttt{kyber*}; see Table 3.2.1), although the usage is split into three subroutines: \texttt{urandom32} (8 lines) generates 4 bytes and then decodes those into a 32-bit integer; \texttt{Short_fromlist} (8 lines) adjusts the bottom 2 bits at each position in an array of 32-bit integers, sorts, and then extracts the bottom 2 bits; \texttt{Short_random} (6 lines) calls \texttt{urandom32} repeatedly and then \texttt{Short_fromlist}.

There is a function \texttt{sample_iid} (5 lines) in \texttt{ntru*/compact} that generates a secret vector with entries 0, 1, 2 in another way: start with a secret array of bytes and reduce each byte mod 3. In \texttt{ntruhrss*/compact}, there is also a function \texttt{sample_iid_plus} (10 lines) that first calls \texttt{sample_iid} and then adjusts the resulting vector to be “positive”; this does not involve further decoding steps.

In \texttt{kyber*/compact}, there is a function \texttt{cbd2} (13 lines) that generates a secret vector with entries in \( \{-2,-1,0,1,2\} \), where each entry is computed as \( a + b - \ldots \)
For $c - d$ for 4 bits $a, b, c, d$. This uses a function $\text{load32\_littleendian}$ (6 lines) that decodes 4 bytes into a 32-bit integer, and arithmetic on 4-bit subsequences of the integer.

In kyber512/compact, along with cbd2, there is a function $\text{cbd3}$ (14 lines) generating secret vectors with entries in $\{-3, -2, -1, 0, 1, 2, 3\}$, where each entry is computed as $a + b + c - d - e - f$ for 6 bits $a, b, c, d, e, f$. This uses another function $\text{load24\_littleendian}$ (6 lines).

5.4. Encoding and decoding messages. In kyber*/compact, there is a function $\text{poly\_frommsg}$ (9 lines) that encodes a 32-byte (256-bit) message as a polynomial mod $x^{256} + 1$, each coefficient being 0 or $(q + 1)/2$. There is also a function $\text{poly\_tomsg}$ (11 lines) that, given a polynomial mod $x^{256} + 1$, rounds each coefficient to 0 or $(q + 1)/2$ to recover a 32-byte message.

There are no analogous functions in ntru* or sntrup*. In those KEMs, the underlying encryption and decryption functions transmit vectors of small integers; there are no separate messages. See Section 7.1.

It is possible to merge encoding and decoding for the kyber* message format with encoding and decoding for two of the three kyber* formats from Section 5.2. This is a case where deviating from ref’s function structure would probably be an improvement: a proliferation of encoding and decoding functions is a risk. As pointed out in Section 4.2, there were secret-dependent timings in divisions in $\text{poly\_tomsg}$ in kyber*/ref until December 2023.

6 Subroutines for hashing

This section looks more closely at how hashing is used in these KEMs.

6.1. Hashing for noise generation. All of these KEMs generate long secret vectors of small random integers. One of the KEM families, kyber*, requires a long secret vector to be generated as a deterministic function of a short secret message; interoperability requires all kyber* software to use this function. Part of this function is the decoding covered in Section 5.3, but there is a preliminary step of applying a hash function to expand the short secret message to a long string provided to the decoding. This expansion is part of the KEM software, separate from whatever expansion is used inside the environment’s RNG.

For example, inside kyber512/compact, $\text{poly\_getnoise\_3}$ (5 lines) converts a 32-byte seed and a 1-byte nonce into 192 bytes of random data by calling a prf function, and then converts those 192 bytes into 256 small integers by calling $\text{cbd3}$ (see Section 5.3). The prf function (6 lines) concatenates its inputs and then calls SHAKE256 from the Keccak code package. The same functions (modulo streamlining) appear in kyber512/ref, along with an implementation of SHAKE256. Internally, SHAKE256 generates 272 bytes in two 136-byte blocks; the first 192 bytes are used.
6.2. Matrix generation. Each use of kyber* deterministically expands a public 32-byte seed into a matrix of integers modulo 3329: in total $2 \cdot 512$ integers for kyber512, $3 \cdot 768$ integers for kyber768, or $4 \cdot 1024$ integers for kyber1024.

The expansion uses SHAKE128 to generate a long output from the 32-byte seed. The output is parsed into 12-bit integers, and then integers $\geq 3329$ are rejected, leaving a sequence of integers between 0 and 3328.

Internally, SHAKE128 “absorbs” the seed into a 200-byte state, applies the Keccak permutation to that state, “squeezes” 168 bytes out of the state, applies Keccak again, “squeezes” 168 more bytes out of the state, etc. Presumably SHAKE128 never ends up stuck in a loop generating only integers $\geq 3329$.

The kyber*/ref software includes various functions for initializing, absorbing, permuting, and squeezing the SHAKE128 state. For kyber*/compact, these are replaced by calls to four external KeccakWidth1600_Sponge subroutines from the official Keccak code package: SpongeInitialize, SpongeAbsorb, SpongeAbsorbLastFewBits, and SpongeSqueeze. There is still some code in kyber*/compact on top of these functions: a 9-line xof_absorb, plus a few state-management lines in gen_matrix.

As noted in Section 1, kyber*/ref does not pass TIMECOP; the above variable-time rejection-sampling loops are the reason. Modifying kyber*/ref to pass TIMECOP is a simple matter of including crypto_declassify.h and inserting crypto_declassify(&state, sizeof state) after the initialization of the SHAKE128 state; kyber*/compact includes this change.

6.3. What API is required? The 20 lines of code mentioned in Sections 6.1 and 6.2 for poly_getnoise_3, prf, and xof_absorb are assuming the best case for kyber*: the environment provides a Keccak library that supports application-selected SHAKE256 output lengths and incremental SHAKE128 squeezing, rather than just a traditional hash-function API generating fixed-length output.

Incrementality is not critical here. One can replace the calls to Sponge* with calls to a simpler SHAKE128 interface that generates enough output all at once. This might also save a few lines of calling code. This would change the functional decomposition of ref; also, one would have to add an analysis of how much output is enough.

6.4. Session keys as hashes. Each KEM produces a 32-byte (256-bit) session key as a hash of a secret plaintext that the receiver recovers by decrypting the ciphertext. This hash function is SHAKE256 (shared with Section 6.1) for kyber*, SHA3-256 for ntru*, and truncated SHA-512 for sntrup*.

The session-key hashing is always one hash call in enc and one in dec, sometimes with a few more lines to assemble inputs (e.g., hashing the plaintext together with the ciphertext); see also the variations in Section 6.6.

6.5. Plaintext confirmation. For sntrup*, another hash of the plaintext is included as an extra component in the ciphertext. This hash is called “plaintext confirmation”. There is no plaintext confirmation for ntru* or kyber*.
The plaintext-confirmation hash for sntrup* includes the public key as an extra input. Actually, this extra input is a hash of the public key, and that hash is cached in the secret key. Furthermore, the plaintext is hashed before it is given to the session hash and to the confirmation hash.

The input to each hash is prefixed by a byte indicating its role. One hash input is byte 4 followed by the public key; one hash input is byte 3 followed by the plaintext; the plaintext-confirmation hash input is byte 2 followed by the plaintext hash and public-key hash; and the session-hash input is byte 1 (or byte 0 for invalid ciphertexts; see Section 6.6) followed by the plaintext hash and ciphertext.

There is code for all of this hashing (and caching), including an 8-line Hash_prefix wrapping SHA-512, a 7-line HashConfirm, a 7-line HashSession, 5 lines of further calls to these functions (including the 2 calls to HashSession mentioned above), and a few more lines handling the cache.

6.6. Reencryption and implicit rejection. After decrypting a ciphertext to produce a plaintext, kyber* and sntrup* reencrypt the plaintext to see whether it produces the same ciphertext. For kyber*/compact, the reencryption is a line in crypto_kem_dec calling indcpa_enc, and the ciphertext comparison is a line calling a 6-line verify function. For sntrup*/compact, the reencryption is a line in crypto_kem_dec calling Hide, and the ciphertext comparison is a line calling a 6-line Ciphertexts_diff_mask function.

For ntru*/compact, decapsulation does not factor in the same way through encryption, but a test with the same effect is handled by a few lines in owcpa_dec, plus a call to owcpa_check_r (11 lines).

In all cases, if the ciphertext does not match, crypto_kem_dec does not report a failure, but instead “implicitly” rejects the ciphertext. This means returning a secretly keyed hash of the ciphertext as a session key, instead of the usual hash of the plaintext. The secret hash key is included in the KEM’s secret key.

For sntrup*/compact, there is a line in crypto_kem_dec overwriting the plaintext with the secret hash key in case of failure, so that the subsequent computation of $H(1, m, c)$ instead computes $H(0, k, c)$. For ntru*/compact, there are 4 lines in crypto_kem_dec computing $H(k, c)$ and using that to overwrite $H(m)$ in case of failure, calling a separate cmov (5 lines). Similar lines in kyber*/compact are subject to change since NIST has expressed plans to remove some of the hashing from kyber*.

7 Design goals for the KEMs

A reader seeing complications in KEM software may be wondering why the complications are there—especially in cases where a complication appears in only one of the studied KEMs. This section looks at how the design goals for the KEMs led to various software complications.
7.1. Encryption and decryption. In all of these KEMs, public keys reveal $A = aG + e$ where $G$ is public and $a, e$ are small secrets. Also, in all of these KEMs, ciphertexts include the traditional NTRU ciphertexts $B = Gb + d$ mentioned in Section 4, where $b, d$ are small secrets. There are two different strategies for decryption:

- In Quotient NTRU, $G$ is chosen as $-e/a$, so $A = 0$. Then $aB = aGb + ad = ad - eb$, which is small and thus does not involve reduction mod $q$. One can choose $a$ to be a multiple of 3, and then dividing by $-e$ mod 3 gives $b$.
- In Product NTRU, there is an extra ciphertext component $C = M + Ab + c$, where $b, c$ are small secrets and $M$ is an encoded message. Then $C - aB = M + (aG + e)b + c - a(Gb + d) = M + eb + c - ad$, which is close to $M$ since $eb + c - ad$ is small. Suitable decoding recovers the message $M$.

The original 1998 NTRU system, ntru*, and sntrup* are examples of Quotient NTRU; kyber* is an example of Product NTRU. For security comparisons, see [10] and [42].

The choice between Quotient NTRU and Product NTRU directly accounts for some of the software complications appearing earlier in this paper:

- Both strategies involve arithmetic mod $q$, but Quotient NTRU also involves arithmetic mod 3. This produces, e.g., extra functions F3_freeze and R3_mult for sntrup*. See Sections 4.1 and 4.3.
- Quotient NTRU involves inversions in key generation, both mod $q$ and mod 3; see Section 4.5. These two functions can be merged into one, and a modified KEM can skip the mod-3 inversion entirely (see, e.g., [32, Algorithm 1]), but there will be at least one inversion function.
- Product NTRU involves encoding the message $M$, and decoding $M + eb + c - ad$ back to $M$. See Section 5.4.
- Product NTRU involves hashing for noise generation, specifically to deterministically derive $b, c, d$ from $M$. See Section 6.1. This is essential for reencryption; see Section 7.4.

7.2. Minimizing size. Product NTRU might seem at first to have keys twice as large as Quotient NTRU, since Quotient NTRU sets $A = 0$ and does not need to transmit $A$. However, NewHope [3] eliminates almost all of the space for $G$ by deterministically computing $G$ from a short seed; kyber* does the same. See Section 6.2.

Product NTRU might also seem to have ciphertexts twice as large as Quotient NTRU, since there are two components $(B, C)$ instead of just one. However, one can choose the errors $d, c$ so that $B$ and $C$ are rounded to limited subsets of the integers mod $q$, and then use this limit to save space in ciphertexts. Decryption requires keeping $eb + c - ad$ small, putting less pressure on $c$ than on $d$ and thus allowing more rounding of $C$ than of $B$; this is why kyber* has two different formats for ciphertext components. See Section 5.2.

One can also choose $B$ to be rounded in Quotient NTRU, and sntrup* does this, accounting for the sntrup* ciphertext format being different from the public-key format. See again Section 5.2.
Size is also the reason that some of the encoders in Section 5 involve multiplications rather than just bit shifts. In particular, the general-purpose \texttt{Encode} and \texttt{Decode} in \texttt{sntrup*} (see Section 5.2) are designed for space efficiency of keys and ciphertexts. There are also multiplications by 3 in an \texttt{ntru*} encoder (see Section 5.1), although this affects only secret-key size.

The above comments should not be viewed as endorsing the idea that any of these size reductions are important for users. For examples of quantifying cryptographic costs in context, see [14, Section 2] and [15].

7.3. Minimizing CPU cycles. The \texttt{kyber*} complications in Section 4.4 arise as follows.

NewHope [3] chooses its dimension $n$ as a power of 2, and chooses its modulus $q$ as a prime for which $x^n + 1$ factors mod $q$ into polynomials of small degree. These choices are copied in \texttt{kyber*}, and allow multiplication mod $x^n + 1$ to be carried out with three size-$n$ NTTs mod $q$ (one NTT of each input, a simpler multiplication in NTT domain, and then an inverse NTT). This uses fewer CPU cycles than various other multiplication methods. Communicating objects in NTT domain then allows some NTTs to be skipped.

Unlike NewHope, \texttt{kyber*} allows just one choice of $n$, namely $n = 256$, and uses matrices to support multiple security levels. The following statement appears in [6, Section 6]:

Optimized implementations only have to focus on a fast dimension-256 NTT and a fast Keccak permutation. This will give very competitive performance for all parameter sets of Kyber.

If “competitive” is understood as comparing to other possible KEMs then this would appear to be a claim that the use of matrices of length-256 polynomials, rather than longer polynomials, is beneficial for performance.

The above comments should not be viewed as endorsing the idea that these are speedups, never mind speedups large enough to be important for users. See, e.g., the aforementioned paper [1] for an environment where these choices appear to hurt performance; [42, Section 6.5] for reasons to believe that these choices will generally hurt hardware performance; and [2] and [25] for fast multiplication software for other KEMs.

7.4. Protection against chosen-ciphertext attacks. There is a long history of chosen-ciphertext attacks against public-key cryptosystems, including lattice-based cryptosystems.

One basic defense against chosen-ciphertext attacks was introduced by Shoup in [48]: namely, the general concept of a KEM, and in particular the structure of hashing a randomly chosen plaintext to obtain a session key (Section 6.4), rather than applying public-key encryption directly to user data. All of the KEMs considered in this paper follow this structure.

The simplest ways to build lattice-based cryptosystems allow ciphertexts to be modified in a way that often produces valid plaintexts. The pattern of successful modifications depends on, and reveals, secret data. Reencryption, plaintext
confirmation, and implicit rejection (see Sections 6.5 and 6.6) are strategies to address chosen-ciphertext attacks. See [13] for a recent attack and a survey of defenses.

Reencryption requires all randomness used in encryption to be recovered in decryption. This forces $b, c, d$ in Product NTRU to be derived deterministically from $M$, as noted in Section 7.1, producing the kyber* complications in Section 6.1. This is also why interoperability requires all kyber* software to use the same decoding function from strings to noise (Section 5.3).

The same randomness-recovery requirement is also what leads to decapsulation in ntru* not factoring through encryption (Section 6.6). The decryption process in ntru* recovers $b$ as in Section 7.1, and then multiplies by $G$ and subtracts from $B$ to recover $d$. The full reencryption process is then optimized down to checking whether $d$ is a valid noise vector; it would be redundant to recompute $Gb + d$ at this point. The situation is different for sntrup*: $Gb$ is simply rounded to obtain $B$, so decryption recovers $b$, and then decapsulation calls encryption as a black box.

7.5. Minimizing morphisms. In 2014, I introduced a “subfield-logarithm” attack [9] exploiting the structure of the algebraic number fields used in some lattice problems. Subsequent developments of the same attack idea have broken various lattice problems: for example, Gentry’s original FHE cryptosystem [27] has been broken in quantum polynomial time for modulus $x^n + 1$ when $n$ is a power of 2. See [42, Section 1.2] for an overview of attacks and further references, and [12] for an example of ongoing developments.

To simplify security review, and in particular to limit the number-theoretic structure given to the attacker, [9] also recommends

- using $x^p - x - 1$ for prime $p$ rather than $x^n \pm 1$,
- using a prime modulus $q$ for which $x^p - x - 1$ is irreducible mod $q$, and
- choosing $q$ large enough to provably eliminate all decryption failures.

These recommendations have, to the extent they have been followed, improved the security of lattice-based cryptography against subsequently published attacks. For example, quantum polynomial-time breaks of Gentry’s system are known for $x^n + 1$ and not for $x^p - x - 1$; meanwhile no proposals have been broken for $x^p - x - 1$ without also being broken for $x^n + 1$. Furthermore, the first version of the Round5 lattice-based KEM was broken [30] by an attack exploiting decryption failures. On the other hand, so far none of the breaks of lattice-based KEMs proposed to NIST have been because of the use of $x^n + 1$.

The recommendations from [9] are used in sntrup*. They account for sntrup* using different primes $q$ for different dimensions $p$, and for extra code to reduce mod $x^p - x - 1$. See Sections 4.1 and 4.3.

References


Analyzing the complexity of reference post-quantum software


