Analyzing the complexity of reference post-quantum software: the case of lattice-based KEMs

Daniel J. Bernstein

University of Illinois at Chicago, Department of Computer Science, USA
Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
Academia Sinica, Institute of Information Science, Taiwan

Abstract. 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 2633 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 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 traces how differences in KEM design goals produced different software complications.

As a spinoff, this paper presents a `kyber512` key-recovery demo exploiting variations in timings of the Kyber reference code.

1 Introduction

The United Kingdom's mass-surveillance agency [**S4.21**] is called the "Government Communications Headquarters" (GCHQ). In 2016, GCHQ registered a new trademark and web site called the "National Cyber Security Centre" (NCSC); see [**I1.16**] and [**W2.17**]. GCHQ's director later wrote [**H2.19**, 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 [**N1.23**] 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"),

E-mail: djb@cr.yp.to (Daniel J. Bernstein)

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: 2024.04.19.

- 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 recommendations in [**N1.23**] against various security measures 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 [**N1.23**] 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 paper [**B2.23**, Section 2])—which, presumably, would have *encouraged* rapid deployment in many (if not all) applications, evidently not the goal of [**N1.23**].

The claim of "significantly more complexity" in [**N1.23**] 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 [**N1.23**] 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 [**B2L24**]. 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., [**R1HTZ13**]. There are also reasons for caution in applying these metrics to cryptography:

- 2021 Blessing–Specter–Weitzner [**B4SW21**] 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 [**B3DHP+23**]), 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., [**A3BBG+23**] 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., [**O23**] and [**W1.23**].)
- 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 2633 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 6.2.

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; meanwhile `ntruhrss701/ref` 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, labeled "`compact`", has been added to SUPERCOP 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 [**A5BDK+21a**, Section 6.1] claims "scalability" as one of the two "unique advantages" of Kyber, it turns out that, among the KEM families considered in this paper, `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 one complication as an indication of one KEM being more complex than another; any particular complication is merely a contributing factor to total complexity.) Section 7 traces how various security and efficiency design goals for these KEMs led to various software complications.

As a spinoff of this analysis, Appendix B presents a demo showing that the complete `kyber512` secret key can often be recovered from timings on a Raspberry Pi 2 of decapsulation with the Kyber reference software.

1.3 Rationale for selection of these case studies

One can, in principle, ask about complexity for a much wider range of cryptographic implementations, but some narrowing is necessary to make a careful analysis feasible. This paper selects three families of lattice KEMs that were submitted to the NIST competition and that are now featured in large real-world deployments. Specifically, see [**O22**] regarding deployment of `sntrup761` from the NTRU Prime submission, [**I2.22**] regarding deployment of `ntruhrss701` from the NTRU submission, and [**EWPWG23**] regarding deployment of `kyber768` from the Kyber submission. These deployments use double encryption, but, as [**N1.23**] illustrates, there is continued pressure to roll out lattice KEMs *without* this protection, making these KEMs natural candidates for studying the "significantly more complexity" claim. Studying multiple lattice KEMs also provides data points regarding the complexity impact of variations in lattice designs. This paper also includes the `ntruhps*` family from the NTRU submission, covering another variation and more choices of security levels.

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, which has 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 `ntrulpr*`.

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.

As noted in Section 1.1, speed is not the point of `ref`, but it may be of interest to note that random sampling of SUPERCOP timings shows `compact` being faster than `ref` for most (KEM, CPU) pairs. One explanation for this is that compiler optimizations such as `-O3` often benefit from code being merged into one file. Systematic speed analysis is outside the scope of this paper.

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_eta1` and `poly_cbd_eta2` 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—

```
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:

```
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 reformats all code with `clang-format` using Google style (which, among other things, always puts opening braces on the previous line) plus the following two `clang-format` options: "`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

```
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 and line 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 (see Appendix A), 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 [Y2.23]:

- "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;

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.

KEM	bytes	bytesw	tokens	bytesz	byteswz	lines	loc	funloc	cyc	funs
`kyber512`	17176	9116	4651	4183	1378	497	425	411	116	49
`kyber768`	16403	8635	4401	4105	1363	469	400	386	110	46
`kyber1024`	16548	8779	4477	4137	1373	472	403	389	114	46
`ntruhps2048509`	13151	7279	4030	2820	1097	393	338	329	93	34
`ntruhps2048677`	13152	7279	4030	2821	1097	393	338	329	93	34
`ntruhps4096821`	12856	7051	3911	2775	1066	381	326	317	91	34
`ntruhrss701`	13322	7441	4232	2864	1144	385	333	325	90	33
`sntrup653`	13306	8630	4660	3205	1266	478	424	415	125	41
`sntrup761`	13308	8630	4660	3205	1266	478	424	415	125	41
`sntrup857`	13308	8630	4660	3204	1266	478	424	415	125	41
`sntrup953`	13308	8630	4660	3229	1276	478	424	415	125	41
`sntrup1013`	13309	8630	4660	3208	1266	478	424	415	125	41
`sntrup1277`	13309	8630	4660	3207	1266	478	424	415	125	41

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.

KEM	bytes	bytesw	tokens	bytesz	byteswz	lines	loc	funloc	cyc	funs
`kyber512`	85312	47043	9504	19547	5909	2633	1290	1084	180	79
`kyber768`	85312	47043	9504	19547	5909	2633	1290	1084	180	79
`kyber1024`	85312	47043	9504	19547	5909	2633	1290	1084	180	79
`ntruhps2048509`	26369	14786	6138	9372	3767	918	589	515	98	38
`ntruhps2048677`	26370	14786	6138	9373	3767	918	589	515	98	38
`ntruhps4096821`	23247	12416	4730	8846	3600	848	525	451	89	38
`ntruhrss701`	27069	15372	6254	9636	3959	928	585	512	96	37
`sntrup653`	26426	15114	6441	8128	3163	1316	744	693	161	66
`sntrup761`	26428	15114	6441	8127	3163	1316	744	693	161	66
`sntrup857`	26428	15114	6441	8130	3163	1316	744	693	161	66
`sntrup953`	26428	15114	6441	8129	3163	1316	744	693	161	66
`sntrup1013`	26428	15114	6441	8130	3163	1316	744	693	161	66
`sntrup1277`	26428	15114	6441	8128	3163	1316	744	693	161	66

- "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.

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.

KEM	external subroutines
`kyber*`	`SHA3_256`, `SHA3_512`, `SHAKE256`, four `Sponge*` subroutines
`ntruhps*`	`crypto_hash_sha3256`, `crypto_sort_int32`
`ntruhrss*`	`crypto_hash_sha3256`
`sntrup*`	`crypto_hash_sha512`, `crypto_sort_uint32`

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.

subroutine	bytes	bytesw	tokens	bytesz	byteswz	lines	loc	funloc	cyc	funs
`hash/sha512`	4466	1902	636	1916	379	65	49	45	17	4
`hash/sha3256`	1778	1401	715	736	374	67	59	57	23	4
`sort/int32`	754	564	264	393	229	31	28	26	11	2
`sort/uint32`	846	634	304	416	239	33	30	28	13	2

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 measurements 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 the metrics from Section 3.1 for this paper's `compact` versions of various subroutines. These implementations are streamlined using the rules from Section 2, starting from `crypto_hashblocks/sha512/compact4`, `crypto_sort/int32/portable3`, and `crypto_sort/uint32/useint32` in SUPERCOP, plus `crypto_xof/shake256/tweet` in libmceliece [**B2C23**], which in turn is based on TweetFIPS202 [**B2SV15**].

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.

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*`. These pairs can be viewed as quantifying the following comment from [**C1DHH+19**, page 4]: "We have unified all aspects of the designs except for the use of fixed-weight sampling." Caution is required in comparing these pairs to the other pairs: these pairs cross KEM families, while the other pairs are instead quantifying code differences across parameter sets within one family. The table does not cover `kyber*` vs. `kyber90s*`, or `sntrup*` vs. `ntrulpr*`.

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.

KEM 1	KEM 2	lines 1	lines 2	lines merged
kyber512	kyber768	493	465	512
kyber512	kyber1024	493	468	574
kyber768	kyber1024	465	468	531
ntruhps2048509	ntruhps2048677	389	389	393
ntruhps2048509	ntruhps4096821	389	377	410
ntruhps2048677	ntruhps4096821	389	377	410
ntruhps2048509	ntruhrss701	389	381	476
ntruhps2048677	ntruhrss701	389	381	476
ntruhps4096821	ntruhrss701	377	381	448
sntrup653	sntrup761	474	474	480
sntrup653	sntrup857	474	474	480
sntrup653	sntrup953	474	474	480
sntrup653	sntrup1013	474	474	480
sntrup653	sntrup1277	474	474	480
sntrup761	sntrup857	474	474	480
sntrup761	sntrup953	474	474	480
sntrup761	sntrup1013	474	474	480
sntrup761	sntrup1277	474	474	480
sntrup857	sntrup953	474	474	480
sntrup857	sntrup1013	474	474	480
sntrup857	sntrup1277	474	474	480
sntrup953	sntrup1013	474	474	480
sntrup953	sntrup1277	474	474	480
sntrup1013	sntrup1277	474	474	480

Recall from Section 1 that [**A5BDK+21a**, 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 $\mathsf{Compress}_q$ 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 [**A3BBG+23**] supports only `kyber512` and `kyber768`.

4 Subroutines for arithmetic

In the 1998 NTRU cryptosystem [**H3PS2.1998**], a ciphertext has the form $Gb + d$. Here b, d are secret integer vectors with small entries, and G is a public linear transformation on vectors of integers mod q. The modulus q is a cryptosystem parameter.

```
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
  r = (r >> 2) + (r & 0x3); // r' mod 3 == r mod 3

  t = r - 3;
  c = t >> 15;

  return (c&r) ^ (~c&t);
}
```

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

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.

Specifically, Section 4.1 covers modular reduction, Section 4.2 covers polynomial multiplication, Section 4.3 covers NTTs and matrices, and Section 4.4 covers polynomial inversion. These subsections do not cover *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`.

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`:

```
#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 twos-complement shift `>>15` to convert negative integers into -1 and nonnegative integers into 0. (The C language does not guarantee twos-complement arithmetic, but SUPERCOP always sets the compiler's `-fwrapv` option, which guarantees twos-complement arithmetic.) One can easily test that this function works for all possible inputs.

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 `sntrup*/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 $2^{26}/q$; see `barrett_reduce` in Figure 4.1.2. 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).

```
int16_t barrett_reduce(int16_t a) {
  int16_t t;
  const int16_t v = ((1U << 26) + KYBER_Q/2)/KYBER_Q;

  t  = ((int32_t)v*a + (1<<25)) >> 26;
  t *= KYBER_Q;
  return a - t;
}
```

Figure 4.1.2: Example of a modular-reduction function from `kyber*/ref`, reducing a 16-bit input mod $q = 3329$.

Specifically, the `F3_freeze` function in `sntrup*/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 `Fq_freeze` function in `sntrup*/compact` does

```
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 + 0x8000000) >> 28);
```

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

For readers wondering why the reference code does not simply use C's built-in division and mod operators ("`/`" and "`%`"): See Appendix B.1.

4.2 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:

```
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`:

```
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 [**B2BCC+20**, 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`.

4.3 NTTs and matrices

The multiplication code in `kyber*/compact` is longer than the multiplication code 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 NTTs when other multipliers are more concise? The answer comes from an interesting feature of this KEM family: namely, public keys are sent in NTT domain. This limits the implementor's choices of multiplication algorithms.

As an example of this limit, consider [**A1HHP+19**], which has been listed since at least 2021 on the Kyber page [**A5BDK+21b**] as one of the "third-party implementations of Kyber". The paper [**A1HHP+19**] 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 [**A1HHP+19**] 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 public keys 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, in the ciphertext formula $Gb+d$, Kyber's b is actually a length-k vector of polynomials modulo $x^{256}+1$, and Kyber's 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.

4.4 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 to each other, using the inversion algorithm from [**B2Y19**]. As in Section 4.2, it would be possible to merge these.

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 encoding/decoding 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 these 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 4.3 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 top of these are four 7-line functions for encoding and decoding of vectors mod q and of rounded vectors mod q.

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

5.3 Decoding for noise generation

In `ntruhps*/compact`, there is a function `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 `ref` code is strikingly less concise, unrolling the conversion of 15 bytes into 4 integers.

```
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 + 2] = (unsigned char) ((t[1] >>  5) | ((t[2] & 0x03) << 6));
    r[11 * i + 3] = (unsigned char) ((t[2] >>  2) & 0xff);
    r[11 * i + 4] = (unsigned char) ((t[2] >> 10) | ((t[3] & 0x7f) << 1));
    r[11 * i + 5] = (unsigned char) ((t[3] >>  7) | ((t[4] & 0x0f) << 4));
    r[11 * i + 6] = (unsigned char) ((t[4] >>  4) | ((t[5] & 0x01) << 7));
    r[11 * i + 7] = (unsigned char) ((t[5] >>  1) & 0xff);
    r[11 * i + 8] = (unsigned char) ((t[5] >>  9) | ((t[6] & 0x3f) << 2));
    r[11 * i + 9] = (unsigned char) ((t[6] >>  6) | ((t[7] & 0x07) << 5));
    r[11 * i + 10] = (unsigned char) ((t[7] >>  3));
  }

  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 + 2] = (unsigned char) (t[1] >>  5) | ((t[2] & 0x03) << 6);
      r[11 * i + 3] = (unsigned char) (t[2] >>  2) & 0xff;
      r[11 * i + 4] = (unsigned char) (t[2] >> 10) | ((t[3] & 0x7f) << 1);
      r[11 * i + 5] = (unsigned char) (t[3] >>  7) | ((t[4] & 0x0f) << 4);
      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);
      r[11 * i + 2] = (unsigned char) (t[1] >>  5) | ((t[2] & 0x03) << 6);
      break;
  }
}
```

Figure 5.2.1: Example of an encoding function from `ntruhps2048*/ref`, packing $n-1$ 11-bit integers into bytes. Compare Figure 5.2.2.

Sorting is used similarly in `sntrup*/compact` (and not in `ntruhrss*` or `kyber*`; see Table 3.2.1), although the usage is split into three subroutines: `urandom32` (8 lines) generates 4 bytes and then decodes those into a 32-bit integer; `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; `Short_random` (6 lines) calls `urandom32` repeatedly and then

```
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);
}
```

Figure 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.

`Short_fromlist`.

There is a function `sample_iid` (5 lines) in `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 `ntruhrss*/compact`, there is also a function `sample_iid_plus` (10 lines) that first calls `sample_iid` and then adjusts the resulting vector to be "positive"; this does not involve further decoding steps.

In `kyber*/compact`, there is a function `cbd2` (13 lines) that generates a secret vector with entries in $\{-2, -1, 0, 1, 2\}$, where each entry is computed as $a + b - c - d$ for 4 bits a, b, c, d. This uses a function `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 `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 `load24_littleendian` (6 lines).

5.4 Encoding and decoding messages

In `kyber*/compact`, there is a function `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 `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. See Appendix B.

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`, `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 `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.

These variable-time rejection-sampling loops are the reason that, as noted in Section 1, `kyber*/ref` does not pass TIMECOP. 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.

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. One expects, and one can prove using absorption details (see [**B1S23**]), that 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`.

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 is not used in `compact` since it would change the functional decomposition of `ref`.

Implementors considering the all-at-once approach are cautioned that it is important to carefully analyze how much output is enough. Skipping this analysis in favor of "test-driven design" could lead to the following type of security failure:

- The buffer size is chosen to have just enough space for the SHAKE128 output used by the available Kyber tests.
- An attacker provides a seed that uses more SHAKE128 output.

- The `enc` implementation reads past the end of the buffer, perhaps reading secret data from other arrays in the program and exposing portions of that data through ciphertexts.

By analogy to Heartbleed [**DKAH+14**] and CacheBleed [**Y1GH16**], one could refer to this failure mode as KyberBleed.

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*`, although NIST has not committed to this at the time of writing.

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 $A = 0$. Decryption computes $aB = aGb + ad = ad - eb$, 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. Decryption computes $C - aB = M + (aG + e)b + c - a(Gb + d) = M + eb + c - ad$, 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 [**B2.19**] and [**N2.21**].

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 4.2.
- Quotient NTRU involves inversions in key generation, both mod q and mod 3; see Section 4.4. These two functions can be merged into one, and a modified KEM can skip the mod-3 inversion entirely (see, e.g., [**H3PS1S+15**, 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 [**A2DPS16**] 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 `Encode` and `Decode` in `sntrup*` (see Section 5.2) are designed for space efficiency of keys and ciphertexts. There are also multiplications by 3 in an `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. See my aforementioned paper [**B2.23**, Section 2] for examples of quantifying cryptographic costs in context.

7.3 Minimizing CPU cycles

The `kyber*` complications in Section 4.3 arise as follows.

NewHope [**A2DPS16**] 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 `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, `kyber*` allows just one choice of n, namely $n = 256$, and uses matrices to support multiple security levels. The following statement appears in [**A5BDK+21a**, 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 [**A1HHP+19**] for an environment where these choices appear to hurt performance; [**N2.21**, Section 6.5] for reasons to believe that these choices will generally hurt hardware performance; and [**A2CCE+21**] and [**C2CHLY23**] 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 [**S3.00**]: 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 [**B2.22b**] 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

Finally, some software complications in `sntrup*` arise as follows.

In 2014, I introduced a "subfield-logarithm" attack [**B2.14**] 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 [**G1.09**] has been broken in quantum polynomial time for modulus $x^n + 1$ when n is a power of 2. See [**N2.21**, Section 1.2] for an overview of attacks and further references, and [**B2.22a**] for an example of ongoing developments.

To simplify security review, and in particular to limit the number-theoretic structure given to the attacker, [**B2.14**] 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 [**H1.18**] 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 [**B2.14**] 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 4.2.

References

[A1HHP+19] Martin R. Albrecht, Christian Hanser, Andrea Höller, Thomas Pöppelmann, Fernando Virdia, Andreas Wallner, *Implementing RLWE-based schemes using an RSA co-processor*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2019** (2019), 169–208. URL: https://eprint.iacr.org/2018/425. Citations in this document: §4.3, §4.3, §4.3, §7.3.

[A2CCE+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben

Niederhagen, Cheng-Jhih Shih, Julian Wälde, Bo-Yin Yang, *Polynomial multiplication in NTRU Prime: comparison of optimization strategies on Cortex-M4*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2021** (2021), 217–238. URL: https://eprint.iacr.org/2020/1216. Citations in this document: §7.3.

[A2DPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, Peter Schwabe, *Post-quantum key exchange—a new hope*, in USENIX 2016 [**H4S16**] (2016), 327–343. URL: https://eprint.iacr.org/2015/1092. Citations in this document: §7.2, §7.3.

[A3BBG+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, Pierre-Yves Strub, *Formally verifying Kyber episode IV: implementation correctness*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2023** (2023), 164–193. URL: https://eprint.iacr.org/2023/215. Citations in this document: §1.1, §3.3.

[A4.23] Argyle Software, *A rust implementation of the Kyber post-quantum KEM* (2023), Accessed 16 December 2023. URL: https://github.com/Argyle-Software/kyber. Citations in this document: §B.1, §B.1.

[A5BDK+21a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, *CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.02)* (2021). URL: https://web.archive.org/web/20211215150153/https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. Citations in this document: §1.2, §3.3, §7.3.

[A5BDK+21b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, *Kyber: Software* (2021); see also newer version [**A5BDK+23**]. URL: https://web.archive.org/web/20210923231123/https://pq-crystals.org/kyber/software.shtml. Citations in this document: §4.3.

[A5BDK+23] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, *Kyber: Software* (2023); see also older version [**A5BDK+23**]. URL: https://web.archive.org/web/20231217003910/https://pq-crystals.org/kyber/software.shtml. Citations in this document: §B.1.

[B1S23] Manuel Barbosa, Peter Schwabe, *Kyber terminates* (2023). URL: https://eprint.iacr.org/2023/708. Citations in this document: §6.2.

[B2.07] Daniel J. Bernstein, *The impact of side-channel attacks on the design of cryptosystems* (2007). URL: https://cr.yp.to/talks.html#2007.05.28. Citations in this document: §B.1.

[B2.14] Daniel J. Bernstein, *A subfield-logarithm attack against ideal lattices* (2014). URL: https://blog.cr.yp.to/20140213-ideal.html. Citations in this document: §7.5, §7.5, §7.5.

[B2.19] Daniel J. Bernstein, *Comparing proofs of security for lattice-based encryption* (2019), Second PQC Standardization Conference. URL: https://cr.yp.to/papers.html#latticeproofs. Citations in this document: §7.1.

[B2.21] Daniel J. Bernstein, saferewrite (2021). URL: https://pqsrc.cr.yp.to/downloads.html. Citations in this document: §B.1.

[B2.22a] Daniel J. Bernstein, *Fast norm computation in smooth-degree Abelian number fields*, Research in Number Theory **9** (2023), article 82, Algorithmic Number Theory Symposium (ANTS) 2022. URL: https://cr.yp.to/papers.html#abeliannorms. Citations in this document: §7.5.

[B2.22b] Daniel J. Bernstein, *A one-time single-bit fault leaks all previous NTRU-HRSS session keys to a chosen-ciphertext attack*, in Indocrypt 2022 [**I3S22**] (2022), 617–643. URL: https://cr.yp.to/papers.html#ntrw. Citations in this document: §7.4.

[B2.23] Daniel J. Bernstein, *Cryptographic competitions*, Journal of Cryptology **37** (2024), article 7. URL: https://cr.yp.to/papers.html#competitions. Citations in this document: §1, §7.2.

[B2B22] Daniel J. Bernstein, Billy Bob Brumley, *Timing attacks: Programming FAQ* (2022). URL: https://timing.attacks.cr.yp.to/programming.html. Citations in this document: §B.1.

[B2BCC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, Bo-Yin Yang, *NTRU Prime: round 3* (2020). URL: https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf. Citations in this document: §4.2.

[B2C23] Daniel J. Bernstein, Tung Chou, *libmceliece* (2023), Accessed 1 December 2023. URL: https://lib.mceliece.org. Citations in this document: §3.2.

[B2L24] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2024), Accessed 1 December 2023 and 28 March 2024. URL: https://bench.cr.yp.to. Citations in this document: §1.1.

[B2SV15] Daniel J. Bernstein, Peter Schwabe, Gilles Van Assche, *Tweetable FIPS 202* (2015). URL: https://keccak.team/2015/tweetfips202.html. Citations in this document: §3.2.

[B2Y19] Daniel J. Bernstein, Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2019.3** (2019), 340–398. URL: https://gcd.cr.yp.to/papers.html. Citations in this document: §4.4.

[B3DHP+23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michael Peeters, Gilles Van Assche, Ronny Van Keer, *eXtended Keccak Code Package* (2023), Accessed 1 December 2023. URL: https://github.com/XKCP/XKCP. Citations in this document: §1.1.

[B4SW21] Jenny Blessing, Michael A. Specter, Daniel J. Weitzner, *You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries* (2021). URL: https://arxiv.org/abs/2107.04940. Citations in this document: §1.1.

[B5.98] Joe P. Buhler (editor), *Algorithmic number theory, third international symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, proceedings*, Lecture Notes in Computer Science, 1423, Springer, 1998. ISBN 3-540-64657-4. See [H3PS2.1998].

[C1DHH+19] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, *NTRU: algorithm specifications and supporting documentation* (2019). URL: `https://ntru.org/f/ntru-20190330.pdf`. Citations in this document: §3.3.

[C2CHLY23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, Chi-Ting Liu, Bo-Yin Yang, *Algorithmic views of vectorized polynomial multipliers for NTRU and NTRU Prime* (2023). URL: `https://eprint.iacr.org/2023/541`. Citations in this document: §7.3.

[DKAH+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael D. Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, Vern Paxson, *The matter of Heartbleed*, in IMC 2014 [**W3AT14**] (2014), 475–488. DOI: `10.1145/2663716.2663755`. Citations in this document: §6.3.

[EWPWG23] Wesley Evans, Bas Westerbaan, Christopher Patton, Peter Wu, Vânia Gonçalves, *Post-quantum cryptography goes GA* (2023). URL: `https://blog.cloudflare.com/post-quantum-cryptography-ga/`. Citations in this document: §1.3.

[F23] Steven K. Fisher, *Pure Java implementation of the Kyber (version 3) post-quantum IND-CCA2 KEM* (2023), Accessed 16 December 2023. URL: `https://github.com/fisherstevenk/kyberJCE`. Citations in this document: §B.1, §B.1.

[G1.09] Craig Gentry, *Fully homomorphic encryption using ideal lattices*, in STOC 2009 [**M09**] (2009), 169–178. DOI: `10.1145/1536414.1536440`. Citations in this document: §7.5.

[G2P16] Benedikt Gierlichs, Axel Y. Poschmann (editors), *Cryptographic hardware and embedded systems—CHES 2016—18th international conference, Santa Barbara, CA, USA, August 17–19, 2016, proceedings*, Lecture Notes in Computer Science, 9813, Springer, 2016. ISBN 978-3-662-53139-6. See [Y1GH16].

[G3.15] Wouter de Groot, *A performance study of X25519 on Cortex-M3 and M4* (2015). URL: `https://research.tue.nl/en/studentTheses/a-performance-study-of-x25519-on-cortex-m3-and-m4`. Citations in this document: §B.1.

[H1.18] Mike Hamburg, *OFFICIAL COMMENT: Round5 = Round2 + Hila5* (2018), Email dated 24 Aug 2018 11:30:24 -0700. URL: `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/YsGkKEJTt5c/m/V0eivEroAAAJ`. Citations in this document: §7.5.

[H2.19] Robert Hannigan, *Organising a government for cyber: the creation of the UK's National Cyber Security Centre* (2019). URL: `https://web.archive.org/web/20200113194346/https://rusi.org/sites/default/files/20190227_hannigan_final_web.pdf`. Citations in this document: §1.

[H3PS1S+15] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, Zhenfei Zhang, *Choosing parameters for NTRUEncrypt* (2015). URL: https://eprint.iacr.org/2015/708. Citations in this document: §7.1.

[H3PS2.1998] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: a ring-based public key cryptosystem*, in ANTS 1998 [**B5.98**] (1998), 267–288, See also [**H3PS2.2016**]. URL: https://ntru.org/f/hps98.pdf. Citations in this document: §4.

[H3PS2.2016] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: a new high speed public key cryptosystem* (2016), Circulated at Crypto 1996, put online in 2016; see also [**H3PS2.1998**]. URL: https://ntru.org/f/hps96.pdf.

[H4S16] Thorsten Holz, Stefan Savage (editors), *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10–12, 2016*, USENIX Assocation, 2016. See [A2DPS16].

[I1.16] Intellectual Property Office of the United Kingdom, *Trade mark: National Cyber Security Centre* (2016). URL: https://trademarks.ipo.gov.uk/ipo-tmcase/page/Results/1/UK00003155044. Citations in this document: §1.

[I2.22] ISE Crypto PQC working group, *Securing tomorrow today: Why Google now protects its internal communications from quantum threats* (2022). URL: https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms. Citations in this document: §1.3.

[I3S22] Takanori Isobe, Santanu Sarkar (editors), *Progress in cryptology—INDOCRYPT 2022—23rd international conference on cryptology in India, Kolkata, India, December 11–14, 2022, proceedings*, 13774, Springer, 2022. ISBN 978-3-031-22911-4. See [B2.22b].

[J21] Jan Jancar, *The state of tooling for verifying constant-timeness of cryptographic implementations* (2021). URL: https://neuromancer.sk/article/26. Citations in this document: §B.1.

[K23] Nadim Kobeissi, *Go implementation of the Kyber (version 3) post-quantum IND-CCA2 KEM* (2023), Accessed 16 December 2023. URL: https://github.com/SymbolicSoft/kyber-k2so. Citations in this document: §B.1, §B.1.

[L1.13] Adam Langley, *Lucky Thirteen attack on TLS CBC* (2013). URL: https://www.imperialviolet.org/2013/02/04/luckythirteen.html. Citations in this document: §B.1.

[L2.23] Jack Lloyd, *Cryptography toolkit* (2023), Accessed 16 December 2023. URL: https://github.com/randombit/botan. Citations in this document: §B.1, §B.1.

[M09] Michael Mitzenmacher (editor), *Proceedings of the 41st annual ACM symposium on theory of computing, STOC 2009, Bethesda, MD, USA, May 31–June 2, 2009*, ACM, 2009. ISBN 978-1-60558-506-2. See [G1.09].

[N1.23] National Cyber Security Centre, *Next steps in preparing for post-quantum cryptography* (2023). URL: https://web.archive.org/web/20231104161635/https://www.ncsc.gov.uk/whitepaper/next-steps-preparing-for-post-quantum-cryptography. Citations in this document: §1, §1, §1, §1, §1, §1, §1.3.

[N2.21] NTRU Prime Risk-Management Team, *Risks of lattice KEMs* (2021). URL: https://ntruprime.cr.yp.to/warnings.html. Citations in this document: §7.1, §7.3, §7.5.

[O22] OpenBSD Project, *OpenSSH 9.0 release notes* (2022). URL: https://www.openssh.com/txt/release-9.0. Citations in this document: §1.3.

[O23] OpenBSD Project, *Portable OpenSSH: sntrup761.c* (2023). URL: https://github.com/openssh/openssh-portable/blob/master/sntrup761.c. Citations in this document: §1.1.

[P1.18] Thomas Pornin, *The problem* (2018). URL: https://www.bearssl.org/ctmul.html. Citations in this document: §B.1.

[P2.00] Bart Preneel (editor), *Advances in cryptology—EUROCRYPT 2000, international conference on the theory and application of cryptographic techniques, Bruges, Belgium, May 14–18, 2000, proceeding*, Lecture Notes in Computer Science, 1807, Springer, 2000. ISBN 3-540-67517-5. See [S3.00].

[R1HTZ13] Danijel Radjenovic, Marjan Hericko, Richard Torkar, Ales Zivkovic, *Software fault prediction metrics: A systematic literature review*, Information and Software Technology **55** (2013), 1397–1418. URL: https://torkar.github.io/pdfs/000d837774c3e143487d18be46c4a28e.pdf. Citations in this document: §1.1.

[R2CS18] John Renner, Sunjay Cauligi, Deian Stefan, *Constant-time WebAssembly* (2018), PriSC 2018: Principles of Secure Compilation. URL: https://cseweb.ucsd.edu/~dstefan/pubs/renner:2018:ct-wasm.pdf. Citations in this document: §B.1.

[S1GG23] Robin Leander Schröder, Stefan Gast, Qian Guo, *HQC implementation out of date/vulnerable* (2023). URL: https://github.com/PQClean/PQClean/issues/482. Citations in this document: §B.1.

[S2.23] Peter Schwabe, *Re: variable-time Kyber ref software* (2023). URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/hWqFJCucuj4/m/ZNLQo2hOAAAJ. Citations in this document: §B.1.

[S3.00] Victor Shoup, *Using hash functions as a hedge against chosen ciphertext attack*, in Eurocrypt 2000 [**P2.00**] (2000), 275–288. Citations in this document: §7.4.

[S4.21] Haroon Siddique, *GCHQ's mass data interception violated right to privacy, court rules* (2021). URL: https://www.theguardian.com/uk-news/2021/may/25/gchqs-mass-data-sharing-violated-right-to-privacy-court-rules. Citations in this document: §1.

[T22] Anton Tutoveanu, *JavaScript implementation of CRYSTALS-KYBER (version 3) post-quantum key exchange algorithm* (2022), Accessed 16 December 2023. URL: https://github.com/antontutoveanu/crystals-kyber-javascript. Citations in this document: §B.1, §B.1, §B.1.

[W1.23] Bas Westerbaan, *"Not negligible, but (at least on the server-side) it's fast enough that we deployed a non AVX2 implementation of Kyber, because we're more worried about implementation mistakes."* (2023), Tweet. URL: https://archive.today/JW9d9. Citations in this document: §1.1.

[W2.17] Whois, *ncsc.gov.uk* (2017). URL: https://web.archive.org/web/20170302143201/https://www.whois.com/whois/ncsc.gov.uk. Citations in this document: §1.

[W3AT14] Carey Williamson, Aditya Akella, Nina Taft (editors), *Proceedings of the 2014 Internet measurement conference, IMC 2014, Vancouver, BC, Canada, November 5–7, 2014*, ACM, 2014. ISBN 978-1-4503-3213-2. URL: http://dl.acm.org/citation.cfm?id=2663716. See [DKAH+14].

[Y1GH16] Yuval Yarom, Daniel Genkin, Nadia Heninger, *CacheBleed: a timing attack on OpenSSL constant time RSA*, in CHES 2016 [**G2P16**] (2016), 346–367. URL: https://eprint.iacr.org/2016/224. Citations in this document: §6.3, §B.1.

[Y2.23] Terry Yin, *A simple code complexity analyser* (2023). URL: https://github.com/terryyin/lizard. Citations in this document: §3.1.

A Supplements

The `supplements.tar.gz` software attached to this PDF includes scripts to rebuild the tables in this paper and to carry out a key-recovery demo exploiting timing variability in Kyber software (see Appendix B). This appendix explains how to run the software.

Prerequisites: Debian with packages `git`, `time`, `python3`, `build-essential` installed; the `lizard` tool (for example, installed in `$HOME` with `python3 -m pip install lizard` if `python3-pip` is installed); for the key-recovery demo, a Raspberry Pi 2 running Raspbian.

To unpack the supplements and switch into the resulting `supplements` directory:

```
gunzip < supplements.tar.gz | tar -xf -
cd supplements
```

In the `supplements` directory, to rebuild `measure-compact.tex` (Table 3.1.1) from the `crypto_kem/*/compact/*` files:

```
sh measure-compact.sh
```

To rebuild `measure-ref.tex` (Table 3.1.2) from the `crypto_kem/*/ref/*` files:

```
sh measure-ref.sh
```

To rebuild `measure-sub.tex` (Table 3.2.2) from the `crypto_{hash,sort}/*/compact/*` files:

```
sh measure-sub.sh
```

To rebuild `merge.tex` (Table 3.3.1) from the `crypto_kem/*/compact/*` files:

```
sh merge.sh
```

To run three experiments with the key-recovery demo:

```
sh demo.sh; sh demo.sh; sh demo.sh
```

A typical experiment takes a few hours and is not guaranteed to succeed (it gives up if it has not found the key from timings of $7 \cdot 2^{18}$ decapsulations), but the demo was observed succeeding ten times in ten experiments. Success prints `yes, eve succeeded`.

B Recovering Kyber secret keys from timings of the reference code

A spot-check carried out as part of this paper's analysis found a timing variation in `kyber*/ref`; see Appendix B.1. I promptly announced this as "possibly exploitable", and promptly investigated, producing a key-recovery demo two weeks later; see Appendix B.2. An accompanying web site `https://kyberslash.cr.yp.to` shows—with caveats regarding the data not necessarily being reliable—18 Kyber library implementations that had the problematic code at the beginning of December 2023 (and 7 that did not); 13 of those 18 have been patched by the time of this writing, in 8 cases with security announcements.

The line counts and other code-complexity measurements reported in this paper are after patches to remove this particular source of timing variability. This should not be interpreted as vouching for the remaining code.

B.1 Discovery of the timing variability

Section 4.1 covers various examples of `ref` and `compact` code avoiding C's built-in division and mod operators. The usual rationale for avoiding those operators 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. See, e.g., [**B2.07**, "What's safe?"], [**L1.13**, "DIV instruction"], [**B2B22**, "division instructions"], and [**S1GG23**, "divide instructions"].
- Compilers might instead convert divisions by constants into multiplication instructions, but one can't rely on this happening. For example, testing various recent versions of `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 `kyber*/ref` computation of `v` displayed in Section 4.1.

But is it actually true that lattice KEM software uses these operators only for public data? Scanning for "`/KYBER_Q`" in `kyber*/ref/*` found some divisions where the numerator was a run-time variable, not just a cryptosystem parameter. In at least one case, this variable was obviously derived from secrets: `indcpa_dec` combines the secret key with a ciphertext and then calls `poly_tomsg` (see Section 5.4), which, when the work for this paper began, had a line

```
t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
```

dividing secret results by q.

Checking (not comprehensively) Kyber implementations listed in [**A5BDK+23**] showed that the same division had been copied into [**L2.23**, "kyber.cpp"], [**A4.23**, "poly.rs"], [**F23**, "Poly.java"], [**K23**, "poly.go"], and [**T22**, "kyber512.js", "kyber768.js", and "kyber1024.js"], although in the case of [**T22**] the choice of JavaScript raises larger questions about constant-time behavior; see generally [**R2CS18**].

There are many tools available to check for timing variations. See [**J21**] for a survey. Some tools, such as `saferewrite` from [**B2.21**], 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 [**L2.23**], [**A4.23**], [**F23**], [**K23**], [**T22**], and SUPERCOP.

In response to the 15 December 2023 announcement, the maintainer of the official Kyber software asked [**S2.23**] 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 $5(q-1)/2 = 8320$). The answer is yes. For example, on AMD Zen 2, division slows down by a cycle when the numerator reaches 8192. As another example, on the SiFive U74 RISC-V CPU, division speeds up by 62 cycles when the numerator reaches 2048, then slows down by a cycle when the numerator reaches 4096, then slows down by another cycle when the numerator reaches 8192.

CacheBleed [**Y1GH16**] is an example of a timing attack exploiting single-cycle variations. The only safe presumption was that this division in Kyber software was also exploitable. This presumption turned out to be correct: this paper's key-recovery demo (see Appendix B.2) was posted on 30 December 2023.

Prasanna Ravi and Matthias Kannwischer pointed out in the meantime that there were further secret numerators beyond the "at least one case" mentioned above. All of those have now been patched in `kyber*/ref`.

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., [**G3.15**] and [**P1.18**]. Eliminating variable-time multiplications is outside the scope of this paper.

B.2 The demo

This demo demonstrates exploitability of variations in decapsulation timing in the end-of-November-2023 official `kyber512` reference code running under Raspbian (`gcc` 8.3.0) on a Raspberry Pi 2. The demo is provided as an attachment to the paper; see Appendix A.

Three experiments were carried out with the original version of the demo before it was posted in December 2023. Two of the experiments succeeded in recovering the full `kyber512` secret key. The current version of the demo differs only in the mechanism used by the attacker to check the computer's clock. Ten experiments were carried out with this version of the demo, and each of the ten succeeded within a few hours (2:13:53, 2:04:45, 3:32:45, 1:59:11, 3:23:30, 2:06:31, 2:34:05, 2:04:46, 3:16:21, 2:23:04). This does not mean that the demo always succeeds.

Internally, the demo works as follows. No claims of optimality are made for the details here: the process of optimizing the demo stopped when the demo was fast enough to run.

Recall that decryption in `kyber*` takes a ciphertext of the form (B, C) and decodes $C - aB$ to recover the message M, where a is the secret key. This decoding process applies the line

```
t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
```

to each coefficient of $C - aB$.

On this platform, `gcc -Os` converts each division into a call to a division subroutine `divsi3`, since `gcc` is compiling for an ABI that doesn't guarantee division instructions. Checking the cost of the `divsi3` subroutine for divisions of n by 3329, for each n in the range of interest, shows that there is a jump by 20 cycles when the numerator n reaches 3329, a further jump by 2 cycles when n reaches 4096, and a further jump by 1 cycle when n reaches 8192.

Each input coefficient t in $C-aB$ turns into a division of $2t+1664$ by 3329. Consequently, there is a big jump in division cost when t reaches 833, and there are smaller jumps when t reaches 1216 and 3264.

Recall that C is a 256-coefficient polynomial (with some constraints on the coefficients because the ciphertext format enforces rounding), and B is a vector of such polynomials. Consider what decryption does when, e.g., $C = 2081 + 2081x + \cdots + 2081x^{254} + 208x^{255}$ and $B = (72x^{100}, 0)$. Note that the final coefficient of C is 208, not 2081.

The coefficient of x^{255} in $C-aB$ has the form $208-72a_{0,155}$, where $a_{0,155}$ is one entry in the Kyber secret key. Recall that each entry is between -3 and 3. If $a_{0,155}$ happens to be 3, then $208 - 72a_{0,155} \bmod 3329$ is 3321, producing a slow division. Otherwise $208 - 72a_{0,155}$ is between 64 and 424, producing a fast division. Other coefficients in $C - aB$ are between 1865 and 2297, producing slow divisions with no obvious dependence upon secrets. One expects the total decapsulation time to be correlated (diagonally) with the time for this division, distinguishing $a_{0,155} = 3$ from the other possibilities for $a_{0,155}$.

Replacing 72 with -72 similarly distinguishes -3 from $-2, -1, 0, 1, 2, 3$; replacing 72 with 107 distinguishes $2, 3$ from $-3, -2, -1, 0, 1$; etc. The demo uses standard techniques to filter out noise in timings, and tries enough ciphertexts that ten out of ten experiments successfully recovered the Kyber secret key.