Post-quantum RSA

20171123

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Daniel J. Bernstein, University of Illinois at Chicago
- Josh Fried, Massachusetts Institute of Technology
- Nadia Heninger, University of Pennsylvania
- Paul Lou, University of Pennsylvania
- Luke Valenta, University of Pennsylvania

E-mail address (preferred): `authorcontact-pqrsa@box.cr.yp.to`

Telephone (if absolutely necessary): +1-312-996-3422

Postal address (if absolutely necessary): Daniel J. Bernstein, Department of Computer Science, University of Illinois at Chicago, 851 S. Morgan (M/C 152), Room 1120 SEO, Chicago, IL 60607–7053.

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: $\times$. See also printed version of "Statement by Each Submitter".

Document generated with the help of `pqskeleton` version 20171123.

Contents

1 Introduction

Join us as we dance madly on the lip of the volcano!
—John Oliver hypothesizing Apple's view
of the difficulty of securing the iPhone
https://www.youtube.com/watch?v=zsjZ2r9Ygzw

Integer factorization is a security disaster. There is a long list of proposed RSA key sizes that have been shown vulnerable to attack. And yet RSA remains standardized and remarkably popular. Users switch to larger RSA key sizes and believe that they will be safe.

Is it clear that quantum attacks should be handled in a different way? More importantly, is it clear that quantum attacks *will* be handled in a different way?

Post-quantum RSA (pqRSA) uses extremely large RSA keys to stop Shor's algorithm. It uses many relatively small secret primes, and small encryption/verification exponents, so that computations with such large keys are affordable for the legitimate users. It still uses secret primes large enough to resist ECM and quantum versions of ECM.

Perhaps there are better quantum algorithms for factorization, especially when the factors are relatively small. This has not been adequately studied—but how many post-quantum proposals *have* been adequately studied? Many other post-quantum proposals are more efficient than pqRSA—but is efficiency a sign of strength, or a sign of weakness?

It is not difficult to envision many RSA users gradually slouching their way into becoming pqRSA users. The cryptographic community should not ignore this possibility: it should figure out whether the possibility is secure.

2 General algorithm specification (part of 2.B.1)

2.1 Parameter space

This pqRSA submission provides signatures, key encapsulation, and public-key encryption. Each operation has two parameters: K, a power of 2; and B, a positive multiple of 8.

There are actually two different options for public-key encryption:

- Use a generic conversion from the key-encapsulation mechanism into a public-key encryption mechanism. For example, use the KEM to send a 32-byte session key, and then use the session key with AES-256-GCM to encrypt and authenticate the message. NIST has indicated that it will apply this conversion automatically.
- Use the public-key encryption mechanism specified below. This is more complicated but saves space.

2.2 Secret key and public key

Alice's secret key consists of K distinct primes. Each prime is between 2^{B-1} and 2^B, and is congruent to 5 modulo 6. Specifically, Alice accumulates a list of K primes by repeating the following steps:

- Generate a B-bit integer as a $(B/8)$-byte random string interpreted in little-endian form.
- Set bits 0 and $B-1$, obtaining an odd integer p between 2^{B-1} and 2^B.
- If $p \bmod 3 = 2$, p is prime, and p is not in the list, then add p to the list.

If K is a significant fraction of the number of primes congruent to 5 modulo 6 between 2^{B-1} and 2^B, then the "not in the list" condition significantly slows down this procedure. If K is larger than the number of primes then the procedure does not terminate. For the parameter sets in this submission, the "in the list" condition has negligible chance of occurring, and the test can safely be skipped.

There is an extensive literature on primality testing, with various tradeoffs between simplicity, speed, conjectured error probability, and provability. Users are expected to follow NIST standards on this topic.

Our reference implementation simply uses GMP's `mpz_probab_prime_p` function. It is easy to artificially construct non-primes p that have a noticeable chance of passing this test, but a *random* non-prime p becomes increasingly unlikely to pass this test as B grows. All of the parameter sets below have $B \geq 512$, and we conjecture that the chance of Alice's key being invalid is below 2^{-128}.

Alice multiplies these K primes to obtain her public key N, an integer between $2^{K(B-1)}$ and 2^{KB}. The number of bytes needed for Alice's N is called A; i.e., A is the unique integer such that $256^{A-1} \leq N < 256^A$. Note that $K(B-1)/8 < A \leq KB/8$.

The secret key is then encoded as a $3K(B/8)$-byte string, namely the concatenation of the following strings:

- For each p: $B/8$ bytes representing p in little-endian form.
- For each p: $B/8$ bytes representing $(N/p)^{-1} \bmod p$ in little-endian form. (This is used inside various standard methods to compute cube roots.)
- $K(B/8)$ bytes representing N in little-endian form.

The public key is also encoded as $K(B/8)$ bytes representing N in little-endian form.

2.3 Signatures

Alice signs a message M as follows:

- Generate a random 32-byte string R.
- Compute $\overline{Y} = H_{A-1}(R, M)$. Here the hash function H_{A-1} is $A-1$ bytes of output of SHAKE256. Recall that A is the number of bytes in N.
- Compute the integer Y represented by $\overline{Y}$ in little-endian form.
- Compute the cube root X of Y modulo N.
- Encode X in little-endian form as a $K(B/8)$-byte string $\overline{X}$.
- The signature is $\overline{X}$ followed by R. The signed message is the signature followed by M.

Bob verifies an alleged signature of a message M' as follows:

- Parse the alleged signature as a $K(B/8)$-byte string $\overline{X'}$ followed by a 32-byte string R'.
- Compute $\overline{Y'} = H_{A-1}(R', M')$.
- Compute the integer X' represented by $\overline{X'}$ in little-endian form. Fail if $X' \geq N$.
- Compute the integer Y' represented by $\overline{Y'}$ in little-endian form.
- Check whether $Y' = (X')^3 \bmod N$.

2.4 KEM

Bob exchanges a secret session key with Alice as follows, given Alice's public key N:

- Generate a string of $K(B/8)$ uniform random bytes.
- Clear the last $K(B/8) - (A-1)$ bytes, obtaining a string $\overline{r}$. Recall that A is the number of bytes in N.
- Compute the session key $H_{32}(\overline{r})$, where H_{32} means 32 bytes of output of SHAKE256.
- Compute the integer r represented by $\overline{r}$ in little-endian form.
- Compute $C = r^3 \bmod N$.
- Encode C in little-endian form as a $K(B/8)$-byte string $\overline{C}$.
- Send $\overline{C}$ as a ciphertext.

Alice decapsulates $\overline{C'}$ as follows:

- Fail if $\overline{C'}$ does not have length $K(B/8)$.
- Compute the integer C' represented by $\overline{C'}$ in little-endian form. Fail if $C' \geq N$.
- Compute the cube root r' of C' modulo N.
- Encode r' in little-endian form as a $K(B/8)$-byte string $\overline{r'}$.
- Compute the session key $H_{32}(\overline{r'})$.

2.5 Public-key encryption

The following encryption mechanism assumes that $K(B-1) \geq 776$. This implies $N \geq 2^{776}$, so $A \geq 98$; recall that A is the number of bytes in N. Define $\alpha = A - 65$; then $\alpha \geq 33$.

Bob sends a secret ℓ-byte message m to Alice as follows, given Alice's public key N:

- If $\ell < \alpha$: Define x_0 as the α-byte string $(m, 1, 0, \ldots, 0)$. There are $\alpha - 1 - \ell$ copies of byte 0.
- If $\ell \geq \alpha$: Define m_0 as the first $\alpha - 33$ bytes of m, and define m_1 as the remaining $\ell - (\alpha - 33)$ bytes of m. Generate a uniform random 32-byte string k, and define x_0 as the α-byte string $(m_0, k, 2)$.
- Generate a uniform random 32-byte string r.
- Compute the α-byte string $x_1 = x_0 \oplus H_\alpha(r, 0)$. Here $\oplus$ means xor; $r, 0$ means r followed by byte 0; and H_α means α bytes of output of SHAKE256.
- Compute the 32-byte string $x_2 = H_{32}(x_0, r, 1)$. Here H_{32} means 32 bytes of output of SHAKE256.
- Compute the 32-byte string $x_3 = r \oplus H_{32}(x_1, x_2, 2)$.
- Compute the $(A-1)$-byte string $\overline{x} = (x_1, x_2, x_3)$.
- Compute the integer x represented by $\overline{x}$ in little-endian form.
- Compute $C = x^3 \bmod N$.
- Encode C in little-endian form as a $K(B/8)$-byte string $\overline{C}$.
- If $\ell < \alpha$: The ciphertext is $\overline{C}$.
- If $\ell \geq \alpha$: The ciphertext is $\overline{C}$ followed by the AES-256-GCM ciphertext for m_1 under key k with nonce 0.

Alice decrypts by reversing the above steps:

- Define $\overline{C'}$ as the first $K(B/8)$ bytes of ciphertext. Fail if the ciphertext has fewer than $K(B/8)$ bytes.
- Define C' as the corresponding integer. Fail if $C' \geq N$.
- Compute the cube root x' of C' modulo N. Fail if $x' \geq 256^{A-1}$.
- Encode x' in little-endian form as an $(A-1)$-byte string $\overline{x}$.
- Parse $\overline{x}$ as (x'_1, x'_2, x'_3) where x'_1, x'_2, x'_3 have $\alpha, 32, 32$ bytes respectively.
- Define r' as the 32-byte string $x'_3 \oplus H_{32}(x'_1, x'_2, 2)$.
- Define x'_0 as the α-byte string $x'_1 \oplus H_\alpha(r', 0)$.
- Compute the 32-byte string $H_{32}(x'_0, r', 1)$. Fail if this string is not x'_2.
- If the ciphertext has exactly $K(B/8)$ bytes: Parse x'_0 as a plaintext m' followed by byte 1 and some number of copies of byte 0. Fail if this parsing fails.
- If the ciphertext has more than $K(B/8)$ bytes: Parse x'_0 as $(m'_0, k', 2)$ where k' has 32 bytes and m'_0 has $\alpha - 33$ bytes. Fail if this parsing fails. Use AES-256-GCM to verify and decrypt the remaining bytes of ciphertext, obtaining m'_1; fail if this fails. Define a plaintext m' as (m'_0, m'_1).

Note that, beyond the usual importance of constant-time computations for security, it is particularly important to hide the differences between an $x' \geq 2^{A-2}$ failure and an $x'_2 \neq H_{32}(1, r', x'_0)$ failure.

3 List of parameter sets (part of 2.B.1)

3.1 Parameter set `encrypt/pqrsa15`

PKE with $K = 512$ and $B = 512$.

3.2 Parameter set `encrypt/pqrsa20`

PKE with $K = 16384$ and $B = 512$.

3.3 Parameter set `encrypt/pqrsa25`

PKE with $K = 262144$ and $B = 1024$.

3.4 Parameter set `encrypt/pqrsa30`

PKE with $K = 8388608$ and $B = 1024$.

3.5 Parameter set `kem/pqrsa15`

KEM with $K = 512$ and $B = 512$.

3.6 Parameter set `kem/pqrsa20`

KEM with $K = 16384$ and $B = 512$.

3.7 Parameter set `kem/pqrsa25`

KEM with $K = 262144$ and $B = 1024$.

3.8 Parameter set `kem/pqrsa30`

KEM with $K = 8388608$ and $B = 1024$.

3.9 Parameter set `sign/pqrsa15`

Signatures with $K = 512$ and $B = 512$.

3.10 Parameter set `sign/pqrsa20`

Signatures with $K = 16384$ and $B = 512$.

3.11 Parameter set `sign/pqrsa25`

Signatures with $K = 262144$ and $B = 1024$.

3.12 Parameter set `sign/pqrsa30`

Signatures with $K = 8388608$ and $B = 1024$.

4 Design rationale (part of 2.B.1)

Shoup's "Simple RSA", also known as "RSA-KEM", takes r as a uniform random integer modulo N. We instead take uniform random r from a power-of-2 range, simplifying the generation process, and more specifically a power-of-256 range, further simplifying the generation process. The size of the range is at least $N/256$, so an algorithm to compute our r given $r^E \bmod N$ has probability at least $1/256$ of computing Shoup's r given $r^E \bmod N$.

We reuse the same range for x in public-key encryption, and for Y in signatures.

In the original RSA paper [9], the encryption/verification exponent E was a random number with as many bits as N. Rabin in [8] suggested instead using a small constant E, and said that $E = 2$ is "several hundred times faster." A complication of $E = 2$ is that each square has 2^K different square roots mod N; $E = 3$ is about twice as slow for encryption but avoids this complication. The subsequent literature has focused mainly on $E = 3$ and $E = 65537$.

There are attacks that compute various types of *structured* Eth roots more quickly than factoring. Some of these attacks are specific to very small E, and historically this has led to some preference for $E = 65537$ over $E = 3$. We instead treat the attacks as a reason to never take Eth powers of structured inputs. There is then no known problem taking $E = 3$.

Shoup has also pointed out that the connection between "RSA-OAEP+" and computing Eth roots is very tight for $E = 3$, but becomes looser as E grows. See [11]. This leads to the following conclusions:

- If computing 3rd roots is harder than computing 65537th roots, then breaking RSA-OAEP+ for $E = 3$ is harder than breaking RSA-OAEP+ for $E = 65537$.
- If computing 3rd roots is the same hardness as computing 65537th roots, then breaking RSA-OAEP+ for $E = 3$ is at least as hard as breaking RSA-OAEP+ for $E = 65537$.
- If computing 3rd roots is easier than computing 65537th roots, then breaking RSA-OAEP+ for $E = 3$ could still be harder than breaking RSA-OAEP+ for $E = 65537$.

The public-key encryption system described above is intended to be an example of RSA-OAEP+, although the details need to be checked carefully.

5 Detailed performance analysis (2.B.2)

5.1 Description of platform

The following measurements were collected on one (otherwise idle) core of a computer named `samba`. The CPU on `samba` is an Intel Xeon E3-1220 v5 (Skylake) running at 3 GHz. Turbo Boost is disabled. `samba` has 64GB of RAM and runs Ubuntu 16.04, with `gcc` 5.4.0.

NIST says that the "NIST PQC Reference Platform" is "an Intel x64 running Windows or Linux and supporting the GCC compiler." `samba` is an Intel x64 running Linux and supporting the GCC compiler. Beware, however, that different Intel CPUs have different cycle counts.

5.2 Time

The following measurements are for `kem`. `encrypt` and `sign` have essentially the same performance as `kem`. There is a slight slowdown for the extra hashing in `encrypt`, and a measurable slowdown for long messages.

Measurements were collected by the program shown in Figure 1, compiled with `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv`. Various measurements were also checked (with no obvious discrepancies) against results of `./do-part` from `supercop-20170904`, with the compiler list reduced to just `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv`, with reduced values of `LOOPS` and `TIMINGS`, and with timeouts (`killafter`) extended.

`pqrsa15`: About 3.5 billion cycles (3483292516) for key generation; 17 million cycles (17492210 17410534 17382984 17361047 17358893) for encapsulation; and 122 million cycles (122127482 122462936 122079316 122135561 122018320) for decapsulation.

Compared to the 32768-byte key size, these are around 110000 cycles per byte, 530 cycles per byte, and 3700 cycles per byte respectively. These figures are useful in understanding how well pqRSA scales to larger key sizes.

`pqrsa20`: About 120 billion cycles (119047642299) for key generation; 1.1 billion cycles (1071561548 1077606577 1076427117 1076293353 1076391885) for encapsulation; and 6.1 billion cycles (6123286512 6116529230 6119549109 6118574953 6118670863) for decapsulation.

Compared to the 1048576-byte key size, these are around 110000 cycles per byte, 1000 cycles per byte, and 5800 cycles per byte respectively.

`pqrsa25`: About 18 trillion cycles (18177137014865) for key generation; 46 billion cycles (46248174238 46222427697 46191747583 46242537978 46191225425) for encapsulation; and 520 billion cycles (519581069107 519569878218 519608774814 519612644254 519558611912) for decapsulation.

Compared to the 33554432-byte key size, these are around 540000 cycles per byte, 1400 cycles per byte, and 15000 cycles per byte respectively. Note that primes are bigger here, 1024 bits instead of 512 bits.

`pqrsa30`: About 590 quadrillion cycles (586593568853135) for key generation; 1.8 quadrillion cycles (1821539719905 1822281625179 1822120731196 1819092856762 1821360421361) for encapsulation; and 22 quadrillion cycles (22294539298463 22296758019988 22296538833629 22300640944170 22296717126117) for decapsulation.

Compared to the 1073741824-byte key size, these are around 550000 cycles per byte, 1700 cycles per byte, and 21000 cycles per byte respectively.

5.3 Space

Sizes are straightforwardly calculated from parameters (and confirmed in various experiments). Specifically, keys are 2^{15} bytes for `pqrsa15`, 2^{20} bytes for `pqrsa20`, 2^{25} bytes for `pqrsa25`, and 2^{30} bytes for `pqrsa30`. KEM ciphertexts have the same size as keys. PKE ciphertexts have the same size as keys if the transmitted messages are short enough. Signatures are 32 bytes longer.

5.4 How parameters affect performance

Encryption and signature verification involve a small number of modular multiplications. Decryption and signature generation are slower, and key generation is even slower, but if B is chosen sensibly then these slowdowns are by factors logarithmic in the number of bits in the modulus.

For comparison, Shor's algorithm involves a quantum modular exponentiation, which is a large number of modular multiplications. See Section 8. The gap in costs grows with the number of bits in the modulus. The growth is essentially linear, giving the legitimate user a rapidly increasing advantage over the attacker as the user's computer power increases.

5.5 Optimizations

Compared to the KEM, the PKE is more complicated but allows compression of encrypted messages. If messages are close to the key size, or longer, then almost the entire traffic is used for message contents.

Similarly, a slightly more complicated scheme for public-key signatures allows pqRSA signed messages to be compressed. This submission skips this option for simplicity.

Checking primality of many independent uniform random integers is faster than checking primality of each integer separately. This speedup is not in the reference software that we are submitting, but it is already implemented in our experimental software.

See [3] for further discussion of various speedup techniques.

6 Expected strength (2.B.4) in general

6.1 Security definitions

The KEM and PKE are designed for IND-CCA2 security. The signature system is designed for EUF-CMA security. See Section 7 for quantitative estimates of the security of specific parameter sets.

6.2 Rationale

See Section 8 for an analysis of known attacks. This analysis also presents the rationale for these security estimates.

7 Expected strength (2.B.4) for each parameter set

7.1 Parameter set `encrypt/pqrsa15`

Scaled-down version provided as a target for cryptanalysis.

7.2 Parameter set `encrypt/pqrsa20`

Scaled-down version provided as a target for cryptanalysis.

7.3 Parameter set `encrypt/pqrsa25`

Scaled-down version provided as a target for cryptanalysis.

7.4 Parameter set `encrypt/pqrsa30`

Category 2, assuming depth limit 2^{64}.

7.5 Parameter set `kem/pqrsa15`

Scaled-down version provided as a target for cryptanalysis.

7.6 Parameter set `kem/pqrsa20`

Scaled-down version provided as a target for cryptanalysis.

7.7 Parameter set `kem/pqrsa25`

Scaled-down version provided as a target for cryptanalysis.

7.8 Parameter set `kem/pqrsa30`

Category 2, assuming depth limit 2^{64}.

7.9 Parameter set `sign/pqrsa15`

Scaled-down version provided as a target for cryptanalysis.

7.10 Parameter set `sign/pqrsa20`

Scaled-down version provided as a target for cryptanalysis.

7.11 Parameter set `sign/pqrsa25`

Scaled-down version provided as a target for cryptanalysis.

7.12 Parameter set `sign/pqrsa30`

Category 2, assuming depth limit 2^{64}.

8 Analysis of known attacks (2.B.5)

8.1 Factorization

2048-bit RSA keys are widely standardized and deployed. This key size is typically estimated to provide more than 2^{100} security against the number-field sieve, and even higher security against other known non-quantum methods for integer factorization. However, (1) precise

estimates vary; (2) it is not clear whether the security level is maintained against multi-user attacks; and, most importantly, (3) post-quantum cryptography also considers *quantum* algorithms. In particular, Shor's quantum algorithm is believed to pose a serious threat to 2048-bit RSA keys.

pqRSA uses much larger RSA keys, slowing down Shor's algorithm so as to reach any desired security level. The number-field sieve scales much more poorly than Shor's algorithm, so we focus on the performance of Shor's algorithm.

Naive analysis. The main bottleneck in Shor's algorithm is computing an n-bit quantity $a^e \bmod N$. Here N is the public key; n is the number of bits in N; a is an integer, which can safely be taken to be small; and e is a superposition of $2n$-bit integers.

Shor computes $a^e \bmod N$ as follows: compute a, $a^2 \bmod N$, $a^4 \bmod N$, $a^8 \bmod N$, etc.; multiply these consecutively into a superposition of products, conditioned upon the bits of e. For reversibility, each multiplication is followed by a corresponding multiplication by $1/a \bmod N$, $1/a^2 \bmod N$, $1/a^4 \bmod N$, $1/a^8 \bmod N$, etc. Overall there are about $8n$ multiplications here, half of which are in superposition.

Each step, multiplying two n-bit integers modulo N, becomes increasingly expensive as n grows. Häner–Roetteler–Svore [7] report approximately $32n^2 \lg n$ Toffoli gates (not counting CNOT gates) for a reversible n-bit modular multiplication, and thus $64n^3 \lg n$ Toffoli gates overall for Shor's algorithm, using a total of $2n+2$ qubits. For $n = 2^{33}$ this is approximately 2^{110} Toffoli gates using approximately 2^{34} qubits.

If each Toffoli gate has comparable cost to 2^{36} non-quantum gates then the total cost is comparable to 2^{146} non-quantum gates, i.e., the cost of finding a SHA3-256 collision, the definition of NIST's "Category 2".

Higher security: communication costs. The naive analysis above counts only the cost of *computation* and not the cost of *communication*. This is important because the algorithm is constantly communicating data across large distances.

Communication of a qubit—or merely a bit—costs energy proportional to the distance communicated. Concretely, Intel states that at 22nm the energy cost of simply *moving* 8 bytes of data is 11.20 pJ "per 5 mm" moved, and that this is "more difficult to scale down" than computation cost; see [6, page 9]. Replacing wires with a different technology might save a constant factor but does not eliminate the scaling difficulties: e.g., lasers spread out linearly over distance. Even with quantum teleportation, there is a cost of the initial setup, namely pushing EPR pairs from one place to another; the cost per bit transmitted again increases linearly with the distance.

Storing $2n$ qubits, or merely $2n$ bits, involves distances at least $n^{1/2+o(1)}$ on any realistic two-dimensional architecture. Architectures are two-dimensional because this allows energy to arrive (and depart) through the third dimension:

- Billions of transistors are spread across a two-dimensional chip, with only a few layers in the third dimension, because otherwise nobody knows how to get the energy in and out.
- At a larger scale, nodes in a computer cluster are spread across two dimensions, with only a few layers in the third dimension, because otherwise nobody knows how to get the energy in and out.

There is a massive literature on real two-dimensional computations, and the costs of accessing n bits of memory are consistently at least $n^{1/2}$ (times the feature sizes etc., which are independent of n). The occasional papers on three-dimensional computations (e.g., [10]) do not seriously address the energy issues, and we have not found literature reporting experiments that can be plausibly interpreted as beating $n^{1/2}$. Similarly, every serious quantum-computing proposal is limited to two dimensions.

Higher security: limits on latency. The naive analysis also makes the absurd assumption that attackers can wait for roughly 2^{100} **serial** qubit operations.

NIST sensibly suggests that submissions consider attacks "restricted to a fixed running time, or circuit depth." NIST observes that 2^{40} sequential qubit operations is "the approximate number of gates that presently envisioned quantum computing architectures are expected to serially perform in a year", and that 2^{64} sequential bit operations is "the approximate number of gates that current classical computing architectures can perform serially in a decade".

Integer multiplication might seem trivial to parallelize with enough hardware: all n bits of one input can be multiplied by all n bits of the other input in parallel; adding the resulting n^2 bits also allows massive parallelism; final carries can also be done in parallel, or skipped with redundant representations of integers. However, this parallelism severely increases communication costs. Specifically, handling n^2 bits in parallel means that distances increase to n, losing another factor $n^{1/2}$ in communication costs beyond the factor $n^{1/2}$ discussed above.

Furthermore, the higher-level loop in Shor's algorithm is quite difficult to parallelize. One can use many more qubits to parallelize the multiplications by a, $a^2 \bmod N$, $a^4 \bmod N$, $a^8 \bmod N$, etc., but this does nothing to parallelize the initial computation of a, $a^2 \bmod N$, $a^4 \bmod N$, $a^8 \bmod N$, etc.

Knowing the factors of N allows parallel exponentiation, as pointed out by von zur Gathen [13] and much later Zeugmann [14], but the problem facing the attacker is to figure out those factors in the first place. Adleman and Kompella [1] suggested a parallel modular-exponentiation algorithm that is essentially a sieving-type discrete-logarithm algorithm run in parallel, but this involves an intolerable amount of computation once n is moderately large. Bernstein and Sorenson [4] slightly reduced the latency of modular exponentiation using a polynomial number of processors in a simplified model of computation, but this incurs huge costs in memory consumption (and in the total number of operations), presumably increasing

communication latency in realistic models of computation.

For comparison, NIST appears to estimate that checking a guess for an AES-128 key takes about 2^{15} bit operations. These operations allow considerable parallelization, so a key-search core carrying out a sequence of 2^{48} key guesses will fit comfortably within the 2^{64} latency limit. A cluster of 2^{80} such cores will find the AES-128 key within the same latency limit. Distributing the target to 2^{80} cores in the first place is a nontrivial communication problem but will still fit within the same latency limit. Similar comments apply to NIST's "Category 2", finding a SHA-256 collision: parallel collision search [12] involves negligible communication costs even with massive parallelization.

The same reasonable latency limit does not appear to allow a search for an AES-256 key with noticeable success probability: for high success probability one needs more than 2^{200} cores, but there is not enough time to communicate the target to so many cores. Does NIST's "Category 5" implicitly assume a higher latency limit? Or does it implicitly rely upon unrealistic assumptions about communication costs? The **lack of clear definitions of NIST's model of computation** makes it difficult to evaluate whether pqRSA fits Categories 3–5 with gigabyte keys. The security estimates above have thus been limited to Category 2.

Lower security: improved algorithms. Attackers will take every possible opportunity to save logarithmic factors and constant factors: for example, there are various techniques to use somewhat shorter exponents in Shor's algorithm. More importantly, the fastest known n-bit multiplication methods take only $n(\log n)^{1+o(1)}$ bit operations.

On the other hand, all integer-multiplication methods on realistic architectures are constrained by the Brent–Kung area-time theorem [5], which states that a chip containing A parallel cores cannot compute n-bit integer multiplication in time below $\Theta(n/\sqrt{A})$. Asymptotically, all known factorization algorithms cost energy at least $n^{2.5+o(1)}$ and have latency at least $n^{1.5+o(1)}$.

Concretely, can an attacker break a gigabyte key within a reasonable latency limit (say a year), while at the same time having the energy costs of non-quantum computation, non-quantum communication, quantum computation, and quantum communication all below the energy cost of finding collisions in SHA3-256? The literature does not demonstrate this; even if it is possible, it is not a risk for the foreseeable future; and users can use larger keys to eliminate the risk. We have successfully generated a 1-terabyte pqRSA key ($n = 2^{43}$ with $K = 2^{31}$ and $B = 2^{12}$), demonstrating feasibility of pqRSA for parameters that leave a substantial security margin.

8.2 Factorization when factors are small

There are various factorization algorithms, such as trial division and the elliptic-curve method (ECM), that are faster than the number-field sieve at finding *small* factors. These methods are even faster than Shor's method when factors are sufficiently small.

ECM finds a prime divisor p of N using $L^{\sqrt{2}+o(1)}$ multiplications modulo N, where $L = \exp(\sqrt{\log p \log\log p})$. Optimizations summarized in [2, Table 10.1] indicate that the $\sqrt{2}+o(1)$ is as low as $0.9\sqrt{2}$ for p around 50 bits, but not noticeably lower; as p increases, the $\sqrt{2}+o(1)$ is forced to converge to $\sqrt{2}$. If $p \approx 2^{512}$ then $L^{0.9\sqrt{2}} \approx 2^{84}$; if $p \approx 2^{1024}$ then $L^{0.9\sqrt{2}} \approx 2^{125}$.

These multiplications are divided into $L^{1/\sqrt{2}+o(1)}$ scalar multiplications. Each scalar multiplication is a series of $L^{1/\sqrt{2}+o(1)}$ multiplications modulo N, similar to the modular exponentiation inside Shor's algorithm. One can adjust ECM parameters to reduce latency, but this increases the total number of operations: if each scalar multiplication is limited to $L^{c+o(1)}$ multiplications modulo N then the attacker needs $L^{1/2c+o(1)}$ scalar multiplications. See [2, Figure 10.1] for a visualization of this effect.

A series of 2^{34} multiplications modulo a 2^{33}-bit public key N is challenging to fit within acceptable latency limits; see the analysis of Shor's algorithm in Section 8.1. If each scalar multiplication in ECM is limited to 2^{34} multiplications then the number of parallel scalar multiplications must be more, presumably much more, than

- 2^{50}, involving more than 2^{83} bits, for $p \approx 2^{512}$; and
- 2^{91}, involving more than 2^{124} bits, for $p \approx 2^{1024}$.

Communicating the target to all of these parallel machines involves further latency problems, as in Section 8.1.

It seems likely that the total number of operations would already reach Category 2 for $p \approx 2^{512}$. We leave a security margin here, without much cost in performance, by instead taking $p \approx 2^{1024}$. The total number of operations is then beyond Category 2, and it is not clear that it is physically possible to achieve the required parallelism within the latency limit.

If latency were not an issue then Grover's method would be applicable for sufficiently large inputs: "GEECM" in [3] carries out a series of just $L^{1/4c+o(1)}$ scalar multiplications, each being a series of $L^{c+o(1)}$ multiplications modulo N. However, as NIST has commented, it is "quite likely" that Grover's method "will provide no advantage to an adversary wishing to perform a cryptanalytic attack that can be completed in a matter of years, or even decades." Furthermore, since the underlying function inside ECM is already at the limit of latency, the only way to apply Grover's method is to further reduce c, further increasing the total number of scalar multiplications.

It is important to study whether there are better quantum algorithms to find small factors, but the current situation is that $p \approx 2^{1024}$ has a comfortable security margin beyond all known algorithms.

8.3 Attacks without factorization

The fastest algorithm known to compute the cube root of a "random-looking" integer modulo N, given the integer and N, is to factor N into primes.

There are much faster algorithms when the root is created with some structure, but we avoid such structure. We clear the top byte of the root, but, as noted above, this cannot improve the success probability of any root-finding algorithm by a factor larger than 256.

An attack against the KEM that works for all functions H can be converted into a root-finding algorithm with similar speed and similar success probability. Of course, this does not eliminate the possibility of an attack that works better for a particular choice of H.

Similar comments apply to the signature system. Similar comments should apply to the PKE system, which is based on Shoup's "OAEP+", although it is difficult to find a full proof in the literature. All of these conversions should be checked carefully.

9 Advantages and limitations (2.B.6)

pqRSA is, as one of our PQCrypto 2017 referees put it, "not cheap". The cost of pqRSA is particularly noticeable in scenarios requiring forward secrecy. However, pqRSA has many compensating advantages in simplicity, flexibility, and familiarity.

Unlike most post-quantum proposals, pqRSA provides encryption and signatures in a single cryptographic primitive. pqRSA also provides more advanced cryptographic functions such as blind signatures.

pqRSA provides much higher pre-quantum security levels than most post-quantum proposals. pqRSA also provides much higher pre-quantum *confidence* than most post-quantum proposals. Any fast algorithm to completely factor a pqRSA key can be converted into a fast algorithm to factor traditional RSA keys: for example, the attacker takes a 2048-bit product of two 1024-bit primes, multiplies by many more 1024-bit primes to obtain a `pqrsa30` key, and factors the result.

Finally, pqRSA benefits from the community's extensive experience with RSA, allowing uncommon levels of reuse of existing software and standards.

References

[1] Leonard M. Adleman and Kireeti Kompella. Using smoothness to achieve parallelism (abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 528–538. ACM, 1988.

[2] Daniel J. Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters. ECM using Edwards curves. *Mathematics of Computation*, 82:1139–1179, 2013.

[3] Daniel J. Bernstein, Nadia Heninger, Paul Lou, and Luke Valenta. Post-quantum RSA. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-quantum cryptography—8th inter-*

national workshop, PQCrypto 2017, Utrecht, the Netherlands, June 26–28, 2017, proceedings, volume 10346 of *Lecture Notes in Computer Science*, pages 311–329, 2017.

[4] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Math. Comput.*, 76(257):443–454, 2007.

[5] Richard P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, 1981.

[6] Dave Dunning. Fabrics—why we love them and why we hate them (talk slides), 2015. http://www.openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Tuesday/tuesday_10.pdf.

[7] Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n + 2$ qubits with Toffoli based modular multiplication. *Quantum Information and Computation*, 17, 2017. https://arxiv.org/abs/1611.07995.

[8] Michael O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, January 1979.

[9] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[10] Arnold L. Rosenberg. Three-dimensional VLSI: A case study. *J. ACM*, 30(3):397–416, 1983.

[11] Victor Shoup. OAEP reconsidered. *J. Cryptology*, 15(4):223–249, 2002. https://eprint.iacr.org/2000/060.

[12] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[13] Joachim von zur Gathen. Computing powers in parallel. *SIAM J. Comput.*, 16(5):930–945, 1987.

[14] Thomas Zeugmann. Highly parallel computations modulo a number having only small prime factors. *Inf. Comput.*, 96(1):95–114, 1992.

A Statements

These statements "must be mailed to Dustin Moody, Information Technology Laboratory, Attention: Post-Quantum Cryptographic Algorithm Submissions, 100 Bureau Drive – Stop 8930, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, or can be given to NIST at the first PQC Standardization Conference (see Section 5.C)."

First blank in submitter statement: full name. Second blank: full postal address. Third, fourth, and fifth blanks: name of cryptosystem. Sixth and seventh blanks: describe and enumerate or state "none" if applicable.

First blank in patent statement: full name. Second blank: full postal address. Third blank: enumerate. Fourth blank: name of cryptosystem.

First blank in implementor statement: full name. Second blank: full postal address. Third blank: full name of the owner.

A.1 Statement by Each Submitter

I, _____, of _____, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that (check one):

- *I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ OR (check one or both of the following):*
  - *to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as _____ may be covered by the following U.S. and/or foreign patents:*

    _____
  - *I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations:*

    _____

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

A.2 Statement by Patent (and Patent Application) Owner(s)

If there are any patents (or patent applications) identified by the submitter, including those held by the submitter, the following statement must be signed by each and every owner, or each owner's authorized representative, of each patent and patent application identified.

I, _____, of _____, am the owner or authorized representative of the owner (print full name, if different than the signer) of the following patent(s) and/or patent application(s): _____

and do hereby commit and agree to grant to any interested party on a worldwide basis, if the cryptosystem known as _____ is selected for standardization, in consideration of its evaluation and selection by NIST, a non-exclusive license for the purpose of implementing the standard (check one):

- *without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination, OR*
- *under reasonable terms and conditions that are demonstrably free of any unfair discrimination.*

I further do hereby commit and agree to license such party on the same basis with respect to any other patent application or patent hereafter granted to me, or owned or controlled by me, that is or may be necessary for the purpose of implementing the standard.

I further do hereby commit and agree that I will include, in any documents transferring ownership of each patent and patent application, provisions to ensure that the commitments and assurances made by me are binding on the transferee and any future transferee.

I further do hereby commit and agree that these commitments and assurances are intended by me to be binding on successors-in-interest of each patent and patent application, regardless of whether such provisions are included in the relevant transfer documents.

I further do hereby grant to the U.S. Government, during the public review and the evaluation process, and during the lifetime of the standard, a nonexclusive, nontransferrable, irrevocable, paid-up worldwide license solely for the purpose of modifying my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability) for incorporation into the standard.

Signed:

Title:

Date:

Place:

A.3 Statement by Reference/Optimized Implementations' Owner(s)

The following must also be included:

I, _____, _____, am the owner or authorized representative of the owner _____ of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

```
#include <stdio.h>
#include <stdlib.h>
#include "cpucycles.h"
#include "crypto_kem.h"

#define TIMINGS 5

long long t[TIMINGS + 1];

void printtimings(const char *s)
{
  long long i;
  long long j;
  long long above;
  long long below;

  printf("%lld %s",(long long) crypto_kem_PUBLICKEYBYTES,s);
  for (i = 0;i < TIMINGS;++i) t[i] = t[i + 1] - t[i];
  for (j = 0;j < TIMINGS;++j) {
    above = below = 0;
    for (i = 0;i < TIMINGS;++i) if (t[i] < t[j]) ++below;
    for (i = 0;i < TIMINGS;++i) if (t[i] > t[j]) ++above;
    if (below <= TIMINGS/2 && above <= TIMINGS/2) {
      printf(" %lld (%lf)",t[j],t[j] / (1.0 * crypto_kem_PUBLICKEYBYTES));
      break;
    }
  }

  for (i = 0;i < TIMINGS;++i) printf(" %lld",t[i]);
  printf("\n");
}
```

Figure 1: A benchmarking program with limited double-checking. See Figure 2 for second half.

```
int main()
{
  unsigned char *sk = malloc(crypto_kem_SECRETKEYBYTES);
  unsigned char *pk = malloc(crypto_kem_PUBLICKEYBYTES);
  unsigned char *c = malloc(crypto_kem_CIPHERTEXTBYTES);
  unsigned char *k = malloc(crypto_kem_BYTES);
  long long i;

  if (!sk) abort();
  if (!pk) abort();
  if (!c) abort();
  if (!k) abort();

  t[0] = cpucycles();
  crypto_kem_keypair(pk,sk);
  t[1] = cpucycles();
  t[1] -= t[0];
  printf("%lld keypair %lld (%lf)\n",(long long) crypto_kem_PUBLICKEYBYTES
    ,t[1],t[1] / (1.0 * crypto_kem_PUBLICKEYBYTES));

  for (i = 0;i < TIMINGS + 1;++i) {
    t[i] = cpucycles();
    crypto_kem_enc(c,k,pk);
  }
  printtimings("enc");

  for (i = 0;i < TIMINGS + 1;++i) {
    t[i] = cpucycles();
    crypto_kem_dec(k,c,sk);
  }
  printtimings("dec");

  return 0;
}
```

Figure 2: A benchmarking program with limited double-checking. See Figure 1 for first half.