Post-quantum RSA

Daniel J. Bernstein[1,2], Nadia Heninger[3], Paul Lou[3], and Luke Valenta[3]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[3] Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19103, USA nadiah,plou,lukev@seas.upenn.edu

Abstract. This paper proposes RSA parameters for which (1) key generation, encryption, decryption, signing, and verification are feasible on today's computers while (2) all known attacks are infeasible, even assuming highly scalable quantum computers. As part of the performance analysis, this paper introduces a new algorithm to generate a batch of primes. As part of the attack analysis, this paper introduces a new quantum factorization algorithm that is often much faster than Shor's algorithm and much faster than pre-quantum factorization algorithms. Initial pqRSA implementation results are provided.

Keywords: post-quantum cryptography, RSA scalability, Shor's algorithm, ECM, Grover's algorithm, Make RSA Great Again

1 Introduction

The 1994 publication of Shor's algorithm prompted widespread claims that quantum computers would kill cryptography, or at least public-key cryptography. For example:

Author list in alphabetical order; see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf. This work was supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO) and project number 645421 (ECRYPT-CSA); by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005; by the U.S. National Institute of Standards and Technology under grant 60NANB10D263; by the U.S. National Science Foundation under grants 1314919, 1408734, 1505799, and 1513671; and by a gift from Cisco. P. Lou was supported by the Rachleff Scholars program at the University of Pennsylvania. We are grateful to Cisco for donating much of the hardware used for our experiments. "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: aaf273785255fe95feca9484e74c7833. Date: 2017.04.19.

- [15]: “nobody knows exactly when quantum computing will become a reality, but when and if it does, it will signal the end of traditional cryptography”.
- [37]: “if quantum computers exist one day, Shor’s results will make all current known public-key cryptographic systems useless”.
- [29]: “It is already proven that quantum computers will allow to break public key cryptography.”
- [20]: “When the first quantum factoring devices are built the security of public-key crypstosystems [*sic*] will vanish.”

But these claims go far beyond the actual limits of Shor’s algorithm, and subsequent research into quantum cryptanalysis has done little to close the gap. The conventional wisdom among researchers in post-quantum cryptography is that quantum computers will kill RSA and ECC but will not kill hash-based cryptography, code-based cryptography, lattice-based cryptography, or multivariate-quadratic-equations cryptography.

Contents of this paper. Is it actually true that quantum computers will kill RSA?

The question here is not whether quantum computers will be built, or will be affordable for attackers. This paper assumes that astonishingly scalable quantum computers will be built, making a qubit operation as inexpensive as a bit operation. Under this assumption, Shor’s algorithm easily breaks RSA *as used on the Internet today*. The question is whether RSA parameters can be adjusted so that all known quantum attack algorithms are infeasible while encryption and decryption remain feasible.

The conventional wisdom is that Shor’s algorithm factors an RSA public key n almost as quickly as the legitimate RSA user can decrypt. Decryption uses an exponentiation modulo n; Shor’s algorithm uses a quantum exponentiation modulo n. There are some small overheads in Shor’s algorithm—for example, the exponent is double-length—but these overheads create only a very small gap between the cost of decryption and the cost of factorization. (Shor speculated in [48, Section 3] that faster quantum algorithms for modular exponentiation “could even make breaking RSA on a quantum computer asymptotically faster than encrypting with RSA on a classical computer”; however, no such algorithms have been found.)

The main point of this paper is that standard techniques for speeding up RSA, when pushed to their extremes, create a much larger gap between the legitimate user’s costs and the attacker’s costs. Specifically, for this paper’s version of RSA, the attack cost is essentially *quadratic* in the usage cost.

These extremes require a careful analysis of quantum algorithms for integer factorization. As part of this security analysis, this paper introduces a new quantum factorization algorithm, GEECM, that is often much faster than Shor’s algorithm and all pre-quantum factorization algorithms. See Section 2. GEECM turns out to be one of the main constraints upon parameter selection for post-quantum RSA.

These extremes also require a careful analysis of algorithms for the basic RSA operations. See Section 3. As part of this performance analysis, this paper introduces a new algorithm to generate a large batch of independent uniform random primes more efficiently than any known algorithm to generate such primes one at a time.

Section 4 reports initial implementation results for RSA parameters large enough to push all known quantum attacks above 2^{100} qubit operations. These results include successful completion of the most expensive operation in post-quantum RSA, namely generating a 1-terabyte public key.

Evaluation and comparison. Post-quantum RSA does not qualify as secure under old-fashioned security definitions requiring asymptotic security against polynomial-time adversaries. However, post-quantum RSA does appear to provide a reasonable level of concrete security.

Note that, for theoretical purposes, it is possible that (1) there are no public-key encryption systems secure against polynomial-time quantum adversaries but (2) there *are* public-key encryption systems secure against, e.g., essentially-linear-time quantum adversaries. Post-quantum RSA is a candidate for the second category.

One might think that the quadratic security of post-quantum RSA is no better than the well-known quadratic security of Merkle's original public-key system. However, the well-known quadratic security is against *pre-quantum* attackers, not against *post-quantum* attackers. The analyses by Brassard and Salvail in [**17**], and by Brassard, Høyer, Kalach, Kaplan, Laplante, and Salvail in [**16**], indicate that more complicated variants of Merkle's original public-key system can achieve exponents close to 1.5 against quantum computers, but this is far below the exponent 2 achieved by post-quantum RSA. Concretely, $(2^{100})^{1/1.5}$ is approximately 100000 times larger than $(2^{100})^{1/2}$.

Post-quantum RSA is not what one would call lightweight cryptography: the cost of each new encryption or decryption is on the scale of \$1 of computer time, many orders of magnitude more expensive than pre-quantum RSA. However, if this is the least expensive way to protect high-security information against being recorded by an adversary today and decrypted by future quantum computers, then it should be of interest to some users. One can draw an analogy here with fully homomorphic encryption: something expensive might nevertheless be useful if it is the least expensive way to achieve the user's desired security goal.

Code-based cryptography and lattice-based cryptography have been studied for many years and *appear* to provide secure encryption at far less expense than post-quantum RSA. However, one can reasonably argue that triple encryption with code-based cryptography, lattice-based cryptography, and post-quantum RSA, for users who can afford it, provides a higher level of confidence than only two of the mechanisms. Post-quantum RSA is also quite unusual in allowing post-quantum encryption, signatures, and more advanced cryptographic functionality such as blind signatures to be provided in a familiar way by a single unified mechanism, a multiplicatively homomorphic trapdoor permutation.

Obviously the overall use case for post-quantum RSA relies heavily on the faint possibility of dramatic improvements in attacks against a broad range of alternatives. But the same criticism applies even more strongly to, e.g., the proposals in [16]. More importantly, it is interesting to see that the conventional wisdom is wrong, and that RSA has enough flexibility to survive the advent of quantum computers—beaten, bruised, and limping, perhaps, but not dead.

Future work. There is a line of work suggesting big secrets as a protection against limited-volume side-channel attacks and limited-volume exfiltration by malware. As a recent example, Shamir is quoted in [7] as saying that he wants the file containing the Coca-Cola secret "to be a terabyte, which cannot be [easily] exfiltrated". A terabyte takes only a few hours to transmit over a gigabit-per-second link, but the basic idea of this line of work is that there are sometimes limits on time and/or bandwidth in side channels and exfiltration channels, and that these limits could stop the attacker from extracting the desired secrets. It would be interesting to analyze the extent to which the secrets in post-quantum RSA provide this type of protection. Beware, however, that a positive answer could be undermined by other parts of the system that have not put the same attention into expanding their data.

Our batch prime-generation algorithm suggests that, to help reduce energy consumption and protect the environment, all users of RSA—including users of traditional pre-quantum RSA—should delegate their key-generation computations to NIST or another trusted third party. This speed improvement would also allow users to generate new RSA keys and erase old RSA keys more frequently, limiting the damage of key theft.[4] However, all trusted-third-party protocols raise security questions (see, e.g., [19] and [24]), and there are significant costs to all known techniques to securely distribute or delegate RSA computations. The challenge here is to show that secure multi-user RSA key generation can be carried out more efficiently than one-user-at-a-time RSA key generation.

Another natural direction of followup work is integration of post-quantum RSA into standard Internet protocols such as TLS. This integration is conceptually straightforward but requires tackling many systems-level challenges, such as various limitations on the RSA key sizes allowed in cryptographic libraries.

Acknowledgments. Thanks to Christian Grothoff for pointing out the application to post-quantum blind signatures. Thanks to Joshua Fried for extensive help with the compute cluster. Thanks to Daniel Genkin for pointing out the possibility that post-quantum RSA naturally provides extra side-channel protection. Thanks to anonymous referees for their helpful comments, including asking about [47] and [52].

[4] If the goal is merely to protect past traffic against complete key theft ("forward secrecy") then a user can obtain a speedup by generating many RSA keys in advance, and erasing each key soon after it is first *used*. But erasing each key soon after it has been *generated* is sometimes advertised as helping protect future traffic against limited types of compromise. Furthermore, batching across many users provides larger speedups.

2 Post-quantum factorization

For every modern variant of RSA, including the variants considered in this paper, the best attacks known are factorization algorithms. This section analyzes the post-quantum complexity of integer factorization.

There have been some papers analyzing and improving the complexity of Shor's algorithm; see, e.g., [56]. However, the literature does not seem to contain any broader study of quantum factorization algorithms. There seems to be an implicit assumption that—once large enough quantum computers are available—Shor's algorithm supersedes the entire previous literature on integer factorization, rendering all previous factorization algorithms obsolete, so studying the complexity of factorization in a post-quantum world is tantamount to studying the complexity of Shor's algorithm.

The main point of this section is that post-quantum factorization is actually a much richer subject. It should be obvious that previous algorithms are not always superseded by Shor's algorithm: as a trivial example, an integer divisible by 2 or 3 or 5 is much more efficiently detected by trial division than by Shor's algorithm. Perhaps less obvious is that there are quantum factorization algorithms that are, for many integers, much faster than Shor's algorithm *and* much faster than all known pre-quantum algorithms. These algorithms turn out to be important for post-quantum RSA, as discussed in Section 3.

Overview of pre-quantum integer factorization. There are two important classes of factorization algorithms. The first class consists of algorithms that are particularly fast at finding small primes: e.g., trial division, the rho method [40], the $p-1$ method [39], the $p+1$ method [55], and the elliptic-curve method (ECM) [35].

Each of these algorithms can be rephrased, without serious loss of efficiency, as a **ring algorithm** that composes the ring operations $0, 1, +, -, \cdot$ to produce a large integer divisible by many small primes. By carrying out the same sequence of operations modulo a target integer n and computing the greatest common divisor of the result with n, one sees whether n is divisible by any of the same primes. For example, trial division up through y has essentially the same performance as computing $\gcd\{n, 2 \cdot 3 \cdot 5 \cdot \cdots \cdot y\}$; as another example, m steps of the rho method compute $\gcd\{n, (\rho_2 - \rho_1)(\rho_4 - \rho_2)(\rho_6 - \rho_3) \cdots (\rho_{2m} - \rho_m)\}$ with $\rho_1 = 1$ and $\rho_{i+1} = \rho_i^2 + 10$.

The importance of ring operations is that carrying them out modulo n has the effect of carrying them out modulo every prime p dividing n; i.e., $\mathbf{Z}/n \to \mathbf{Z}/p$ is a ring morphism. To measure the speed and effectiveness of a ring algorithm one sees how many operations are carried out by the algorithm and how many primes p of various sizes divide the output. The size of n is almost irrelevant, except that each ring operation modulo n costs $(\lg n)^{1+o(1)}$ bit operations.

The second class consists of **congruence-combining algorithms**: e.g., the continued-fraction method [33], the quadratic sieve [41], and the number-field sieve (NFS) [34]. These algorithms multiply various congruences modulo n to obtain a congruence of the form $a^2 \equiv b^2 \pmod{n}$, and then hope that $\gcd\{n, a - b\}$

is a nontrivial factor of n. These algorithms are not usefully viewed as ring algorithms (the congruences modulo n are produced in a way that depends on n) and are not particularly fast at finding small primes.

For large n the best congruence-combining algorithm appears to be NFS, which (conjecturally) uses $2^{(\lg n)^{1/3+o(1)}}$ bit operations. For comparison, ECM uses $2^{(\lg y)^{1/2+o(1)}}$ ring operations if ECM parameters are chosen to (conjecturally) find every prime $p \le y$. Evidently ECM uses fewer bit operations than NFS to find sufficiently small primes p; the cutoff is $2^{(\lg n)^{2/3+o(1)}}$.

Shor's algorithm. Shor begins with a circuit to compute the function $x \mapsto (x, 3^x \bmod n)$, where x is an integer having about $2 \lg n$ bits. Exponentiation uses about $2 \lg n$ multiplications modulo n, and the best multiplication methods known use $(\lg n)^{1+o(1)}$ bit operations, so exponentiation uses $(\lg n)^{2+o(1)}$ bit operations.

A standard conversion produces a quantum circuit that uses $(\lg n)^{2+o(1)}$ qubit operations to evaluate the same function on a quantum superposition of inputs. With a small extra overhead (applying a quantum Fourier transform to the output, sampling, et al.) Shor finds the period of this function, i.e., the order of 3 modulo n. This order is a divisor, typically a large divisor, of $\varphi(n) = \#(\mathbf{Z}/n)^*$, and factoring n with this information is a standard exercise. In the rare case that 3 has small order modulo n, one can replace 3 with a random number—preferably a small random number to save time in exponentiation.

There is a tremendous gap between the $(\lg n)^{2+o(1)}$ qubit operations used by Shor and the $2^{(\lg n)^{1/3+o(1)}}$ bit operations used by NFS. Of course, for the moment qubit operations seem impossibly expensive compared to bit operations, but post-quantum cryptography looks ahead to a future where qubit operations are affordable at a large scale. In this future it seems that congruence-combining algorithms will be of little, if any, interest.

On the other hand, Shor's algorithm is not competitive with ring algorithms at finding small primes. Even if a qubit operation is as inexpensive as a bit operation, Shor's $(\lg n)^{2+o(1)}$ qubit operations are as expensive as $(\lg n)^{1+o(1)}$ ring operations. ECM's $2^{(\lg y)^{1/2+o(1)}}$ ring operations are better than this for sufficiently small primes. The cutoff is $2^{(\lg \lg n)^{2+o(1)}}$.

Some wishful thinking. One might think that Shor's algorithm can be tweaked to take advantage of a small prime divisor p of n: the function $x \mapsto 3^x \bmod p$ has small period, and this period should be visible for x having only about $2 \lg p$ bits, rather than the $2 \lg n$ bits used by Shor. This would save a factor of 2 even in the most extreme case $p \approx \sqrt{n}$.

The difficulty is that one is not given the function $x \mapsto 3^x \bmod p$. The function $x \mapsto 3^x \bmod n$ has a small pseudo-period, in the sense that shifting the input produces a related output, but one is also not given this relation.

If there were a fast way to detect pseudo-periods with respect to unknown relations then one could drastically speed up Shor's algorithm by finding the pseudo-period p of the simpler function $x \mapsto x \bmod n$. If x is limited to $2 \lg p < \lg n$ bits then this function is simply the identity function $x \mapsto x$, independent of n, so there would have to be some other way for the algorithm to learn about n. These obstacles seem insurmountable.

A quantum ring algorithm: GEECM. A more productive approach is to take the best pre-quantum algorithms for finding small primes, and to accelerate those algorithms using quantum techniques.

Under standard conjectures, ECM finds primes $p \le y$ using $2^{(\lg y)^{1/2+o(1)}}$ ring operations, as mentioned above; the rho method finds primes $p \le y$ using $y^{1/2+o(1)}$ ring operations; and trial division (in its classic form) finds primes $p \le y$ using $y^{1+o(1)}$ ring operations. Evidently ECM supersedes the rho method and trial division as y grows. The cutoff is generally stated (on the basis of more detailed analyses of the $o(1)$) to be below 2^{30}, and the primes of interest in this paper are much larger, so this paper focuses on ECM.

(There are occasional primes for which the $p-1$ and $p+1$ methods are faster than ECM, but the primes of interest in this paper are randomly generated. Most of the comments in this section generalize to hyperelliptic curves, but genus-$\ge$2-hyperelliptic-curve methods have always been slightly slower than ECM.)

The state-of-the-art variant of ECM is EECM (ECM using Edwards curves), introduced by Bernstein, Birkner, Lange, and Peters in [12]. EECM chooses an Edwards curve $x^2 + y^2 = 1 + dx^2y^2$ over $\mathbf{Q}$, or more generally a twisted Edwards curve, with a known non-torsion point P; EECM also chooses a large integer s and uses the Edwards addition law to compute the sth multiple of P on the curve, and in particular the x-coordinate $x(sP)$, represented as a fraction of integers. The output of the ring algorithm is the numerator of this fraction. Overall the computation takes $(7 + o(1)) \lg s$ multiplications (more than half of which are squarings) and a comparable number of additions and subtractions. For optimized curve choices and further details see [12], [11], [14], [5], and [22].

If s is chosen as $\operatorname{lcm}\{1, 2, \ldots, z\}$ then $\lg s \approx 1.4z$ so this curve computation uses about $10z$ multiplications. If $z \in L^{c+o(1)}$ as $y \to \infty$, where $L = \exp\sqrt{\log y \log\log y}$ and c is a positive real constant, then standard conjectures imply that each prime $p \le y$ is found by this curve with probability $1/L^{1/2c+o(1)}$. Standard conjectures also imply that curves are almost independent, so by trying $L^{1/2c+o(1)}$ curves one finds each prime p with high probability. The total cost of trying all these curves is $L^{c+1/2c+o(1)}$ ring operations. The expression $c+1/2c$ takes its minimum value 1 for $c = 1/\sqrt{2}$; the total cost is then $L^{\sqrt{2}+o(1)}$ ring operations.

This paper introduces GEECM (Grover plus EECM), which uses quantum computers as follows to accelerate the same EECM computation. Recall that Grover's method accelerates searching for roots of functions: if the inputs to a function f are roots of f with probability $1/R$, then classical searching performs (on average) R evaluations of f, while Grover's method performs about $\sqrt{R}$ quantum evaluations of f. Consider, in particular, the function f whose input is an EECM curve choice, and whose output is 0 exactly when the EECM result for that curve choice has a nontrivial factor in common with n. EECM finds a root of f by classical searching; GEECM finds a root of f by Grover's method. If s and z are chosen as above then the inputs to f are roots of f with probability

$1/L^{1/2c+o(1)}$, so GEECM uses just $L^{1/4c+o(1)}$ quantum evaluations of f, for a total of $L^{c+1/4c+o(1)}$ quantum ring operations. The expression $c+1/4c$ takes its minimum value 1 for $c=1/2$; the total cost is then just $L^{1+o(1)}$ ring operations.

To summarize, GEECM reduces the number of ring operations from $L^{\sqrt{2}+o(1)}$ to $L^{1+o(1)}$, where $L=\exp\sqrt{\log y\log\log y}$. For the same number of operations, GEECM increases $\log y$ by a factor $2+o(1)$, almost doubling the number of bits of primes that can be found.

3 RSA scalability

Obviously a post-quantum RSA public key n will need to be quite large to resist the attacks described in Section 2. This section analyzes the scalability of the best algorithms available for RSA key generation, encryption, decryption, signature generation, and signature verification.

Small exponents. The fundamental RSA public-key operation is computing an eth power modulo n. This modular exponentiation uses approximately $\lg e$ squarings modulo n, and, thanks to standard windowing techniques, $o(\lg e)$ extra multiplications modulo n.

In the original RSA paper [**43**], e was a random number with as many bits as n. Rabin in [**42**] suggested instead using a small constant e, and said that $e=2$ is "several hundred times faster." Rabin's speedup factor grows as $\Theta(\lg n)$, making it particularly important for the large sizes of n considered in this paper.

The slower but simpler choice $e=3$ was deployed in a variety of real-world applications. The much slower alternative $e=65537$ subsequently became popular as a means of compensating for poor choices of RSA message-randomization mechanisms, but with proper randomization no attacks against $e=3$ are known that are faster than factorization.

For simplicity this paper also focuses on $e=3$. Computing an eth power modulo n then takes one squaring modulo n and one general multiplication modulo n. Each of these steps takes just $(\lg n)^{1+o(1)}$ bit operations using standard fast-multiplication techniques; see below for further discussion. Notice that $(\lg n)^{1+o(1)}$ is asymptotically far below the $(\lg n)^{2+o(1)}$ cost of Shor's algorithm.

Many primes. The fundamental RSA secret-key operation is computing an eth root modulo n. For $e=3$ one chooses n as a product of distinct primes congruent to 2 modulo 3; then the inverse of $x\mapsto x^3 \bmod n$ is $x\mapsto x^d \bmod n$, where $d=(1+2\prod_{p|n}(p-1))/3$. Unfortunately, d is not a small exponent—it has approximately $\lg n$ bits.

A classic speedup in the computation of $x^d \bmod n$ is to compute $x^d \bmod p$ and $x^d \bmod q$, where p and q are the prime divisors of n, and to combine them into $x^d \bmod n$ by a suitably explicit form of the Chinese remainder theorem. Fermat's identity $x^p \bmod p = x \bmod p$ further implies that $x^d \bmod p = x^{d \bmod (p-1)} \bmod p$ (since $d \bmod (p-1)\ge 1$) and similarly $x^d \bmod q = x^{d \bmod (q-1)} \bmod q$. The exponents $d \bmod (p-1)$ and $d \bmod (q-1)$ have only half as many bits as n; the exponentiation $x^d \bmod n$ is thus replaced by two exponentiations with half-size exponents and half-size moduli.

If n is a product of more primes, say $k \geq 3$ primes, then the same speedup becomes even more effective, using k exponentiations with $(1/k)$-size exponents and $(1/k)$-size moduli. Prime generation also becomes much easier since the primes are smaller. Of course, if primes are too small then the attacker can find them using the ring algorithms discussed in the previous section—specifically EECM before quantum computers, and GEECM after quantum computers.

What matters for this paper is how multi-prime RSA scales to much larger moduli n. Before quantum computers the top threats are EECM and NFS, and balancing these threats implies that each prime p has $(\lg n)^{2/3+o(1)}$ bits (see above), i.e., that $k \in (\lg n)^{1/3+o(1)}$. After quantum computers the top threats are GEECM and Shor's algorithm, and balancing these threats implies that each prime p has just $(\lg \lg n)^{2+o(1)}$ bits, i.e., that $k \in (\lg n)/(\lg \lg n)^{2+o(1)}$. RSA key generation, decryption, and signature generation then take $(\lg n)^{1+o(1)}$ bit operations; see below for further discussion.

Key generation. To recap: A k-prime exponent-3 RSA public key n is a product of k distinct primes p congruent to 2 modulo 3. In particular, a post-quantum RSA public key n is a product of k distinct primes p congruent to 2 modulo 3, where each prime p has $(\lg \lg n)^{2+o(1)}$ bits.

Standard prime-generation techniques use $(\lg p)^{3+o(1)}$ bit operations. See, e.g., [**6**, Section 3] and [**38**, Section 4.5]. The point is that one must try about $\log p$ random numbers before finding a prime, and checking primality has similar cost to a single exponentiation modulo p.

A standard speedup is to check whether p is divisible by any primes up through some limit, say y. The chance of a random integer surviving this divisibility test is approximately $1/\log y$, reducing the original pool of $\log p$ random numbers to $(\log p)/\log y$ random numbers and saving an overall factor of $\log y$ *if* the trial division is not a bottleneck. The conventional view is that keeping the cost of trial division under control requires y to be chosen as a polynomial in $\lg p$, saving a factor of only $\Theta(\lg \lg p)$ and thus still requiring $(\lg p)^{3+o(1)}$ bit operations.

A nonstandard speedup is to replace trial division (or sieving) by batch trial division [**8**] or batch smoothness detection [**9**]. The algorithm of [**9**] reads a finite sequence S of positive integers and a finite set P of primes, and finds "the largest P-smooth divisor of each integer in S" using just $b(\lg b)^{2+o(1)}$ bit operations, where b is the total number of bits in P and S. In particular, if P is the set of primes up through y, and S is a sequence of $\Theta(y/\lg p)$ integers each having $\Theta(\lg p)$ bits, then b is $\Theta(y)$ and this algorithm uses just $y(\lg y)^{2+o(1)}$ bit operations, i.e., $(\lg p)(\lg y)^{2+o(1)}$ bit operations for each element of S. Larger sequences S can trivially be split into sequences of size $\Theta(y/\lg p)$, producing the same performance per element of S.

To do even better, assume that the original size of S is at least 2^{2^α}, and apply batch smoothness detection successively for $y = 2^{2^0}$, $y = 2^{2^1}$, $y = 2^{2^2}$, and so on through $y = 2^{2^\alpha}$. Each step weeds out about half of the remaining elements of S as composites; the next step costs about four times as much per element but is applied to only half as many elements. The total cost is just $(\lg p)(2^\alpha)^{1+o(1)}$ bit operations for each of the original elements of S. Each of the original elements has probability about $1/2^\alpha$ of surviving this process and incurring an exponentiation, which costs $(\lg p)^{2+o(1)}$ bit operations. Choosing $2^\alpha \in (\lg p)^{0.5+o(1)}$ balances these costs as $(\lg p)^{1.5+o(1)}$ for each of the original elements of S, i.e., $(\lg p)^{2.5+o(1)}$ for each prime generated.

In the context of post-quantum RSA the assumption about the original size of S is satisfied: one has to generate $(\lg n)^{1+o(1)}$ primes, so the original size of S is $(\lg n)^{1+o(1)}$, which is at least 2^{2^α} for $2^\alpha \in (1+o(1)) \lg\lg n$; this choice of α satisfies $2^\alpha \in (\lg p)^{0.5+o(1)}$ since $\lg p \in (\lg\lg n)^{2+o(1)}$. The primes are also balanced, in the sense that $(\lg n)/k \in (\lg p)^{1+o(1)}$ for each p, so generating k primes in this way uses $k(\lg p)^{2.5+o(1)} = (\lg n)(\lg p)^{1.5+o(1)} = (\lg n)(\lg\lg n)^{3+o(1)}$ bit operations.

Computing n by multiplying these primes uses only $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations using standard fast-arithmetic techniques; see, e.g., [10, Section 12]. At this level of detail it does not matter whether one uses the classic Schönhage–Strassen multiplication algorithm [46], Fürer's multiplication algorithm [21], or the Harvey–van der Hoeven–Lecerf multiplication algorithm [27].

The total number of bit operations for key generation is essentially linear in $\lg n$. For comparison, the usual picture is that prime generation is vastly more expensive than any of the other steps in RSA.

One can try to further accelerate key generation using Takagi's idea [52] of choosing n as $p^{k-1}q$. We point out two reasons that this is worrisome. The first reason is lattice attacks [13]. The second reason is that any nth power modulo n has small order, namely some divisor of $(p-1)(q-1)$; Shor's algorithm finds the order at relatively high speed once the nth power is computed.

Encryption and decryption. There are many different RSA encryption mechanisms in the literature. The oldest mechanisms use RSA to directly encrypt a user's message; this requires careful padding and scrambling of the message. Newer mechanisms generate a secret key (for example, an AES key), use the secret key to encrypt and authenticate the user's message, and use RSA to encrypt the secret key; this allows simpler padding, since the secret key is already randomized. The newest mechanisms such as Shoup's "RSA-KEM" [51] simply use RSA to encrypt $\lg n$ bits of random data, hash the random data to obtain a secret key, and use the secret key to encrypt and authenticate the user's message; this does not require any padding. For simplicity this paper takes the last approach.

Generating large amounts of truly random data is expensive. Fortunately, truly random data can be simulated by pseudorandom data produced by a stream cipher from a much smaller key. (Even better, slight deficiencies in the randomness of the cipher key do not compromise security.) The literature contains several scalable ciphers that produce a $\Theta(b)$-bit block of output from a $\Theta(b)$-bit key, with a conjectured 2^b security level, using $b^{2+o(1)}$ bit operations (and even fewer for some ciphers), i.e., $b^{1+o(1)}$ bit operations for each output bit. In the context of post-quantum RSA one has $b \in \Theta(\lg\lg n)$ so generating $\lg n$

pseudorandom bits costs $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations. The same ciphers can also be converted into hash functions with only a constant-factor loss in efficiency, so hashing the bits also costs $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations.

Multiplication also takes $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations. Squaring, reduction modulo n, multiplication, and another reduction modulo n together take $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations. The overall cost of RSA encryption is therefore $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations plus the cost of encrypting and authenticating the user's message under the resulting secret key.

Decryption is more complicated but not much slower; it works as follows. First reduce the ciphertext modulo all of the prime divisors of n. This takes $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations using a remainder tree or a scaled remainder tree; see, e.g., [**10**, Section 18]. Then compute a cube root modulo each prime. A cube root modulo p takes $(\lg p)^{2+o(1)}$ bit operations, so all of the cube roots together take $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations. Then reconstruct the cube root modulo n. This takes $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations using fast interpolation techniques; see, e.g., [**10**, Section 23]. Finally hash the cube root. The overall cost of RSA decryption is $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations, plus the cost of verifying and decrypting the user's message under the resulting secret key.

Shamir in [**47**] proposed decrypting modulo just one prime, and choosing plaintexts to be smaller than primes. However, this requires exponents to be much larger for security, and in the context of post-quantum RSA this slows down encryption by vastly more than it speeds up decryption. A more interesting variant, which we do not explore further, is to use a significant fraction of the primes to decrypt a plaintext having $(\lg n)/(\lg\lg n)^{0.5+o(1)}$ bits; this should reduce the total cost of encryption and decryption to $(\lg n)(\lg\lg n)^{1.5+o(1)}$ bit operations with a properly chosen exponent.

Signature generation and verification. Standard padding schemes for RSA signatures involve the same operations discussed above, such as hashing to a short string and using a stream cipher to expand the short string to a long string.

The final speeds are, unsurprisingly, $(\lg n)(\lg\lg n)^{2+o(1)}$ bit operations to generate a signature and $(\lg n)(\lg\lg n)^{1+o(1)}$ bit operations to verify a signature, plus the cost of hashing the user's message.

4 Concrete parameters and initial implementation

Summarizing what we've learned so far: Shor's algorithm takes $(\lg n)^{2+o(1)}$ qubit operations to factor n. If the prime divisors of n are too small then GEECM becomes a larger threat than Shor's algorithm; protecting against GEECM requires each prime to have $(\lg\lg n)^{2+o(1)}$ bits. Section 3 showed that, under this constraint, all of the RSA operations can be carried out using $(\lg n)(\lg\lg n)^{O(1)}$ bit operations; the $O(1)$ is $3 + o(1)$ for key generation, $2 + o(1)$ for decryption and signature generation, and $1 + o(1)$ for encryption and signature verification.

These asymptotics do not imply anything about any particular size of n. This section looks at performance in more detail, and in particular reports successful generation of a 1-terabyte post-quantum RSA key built from 4096-bit primes.

Prime sizes and key sizes. Before looking at performance, we explain why these sizes (1-terabyte key, 4096-bit primes) provide ample security.

A 1-terabyte key n has 2^{43} bits, so Shor's algorithm uses 2^{44} multiplications modulo n. We have not found literature analyzing the cost of circuits for optimized FFT-based multiplication at this scale, so we extrapolate as follows.

The recent speed records from Harvey–van der Hoeven–Lecerf [28] for multiplication of degree-2^{21} polynomials over a particularly favorable finite field, $\mathbf{F}_{2^{60}}$, use 640 milliseconds on a 3.4GHz CPU core. More than half of the cycles are performing 128-bit vector xor, and more than 10% of the cycles are performing 64×64-bit polynomial multiplications, according to [28, Section 3.3], for a total of approximately 2^{40} bit operations to multiply 2^{27}-bit inputs.

Imagine that the same 2^{13} ratio scales directly from 2^{27}-bit inputs to 2^{43}-bit inputs; that integer multiplication uses as few bit operations as binary-polynomial multiplication; that reduction modulo n does not cost anything; and that there are no overheads for switching from bit operations to reversible qubit operations inside a realistic quantum-computer architecture. (For comparison, the ratio in [56] is more than 2^{20} for 2^{20}-bit inputs.) Each multiplication modulo n inside Shor's algorithm then uses 2^{56} qubit operations, and overall Shor's algorithm consumes an astonishing 2^{100} qubit operations.

We caution the reader that this is only a preliminary estimate. A thorough analysis would have to account for several overheads mentioned above; for the number of Shor iterations required; for known techniques to reduce the number of iterations; for techniques to use slightly fewer multiplications per iteration; and for the latest improvements in integer-multiplication algorithms.

As for prime sizes: Standard pre-quantum cost analyses conclude that 4096-bit RSA keys provide roughly 2^{140} security against all available algorithms. ECM is well known to be inferior to NFS at such sizes; evidently it uses even more than 2^{140} bit operations to find 2048-bit primes. ECM would be even slower against a much larger modulus, simply because arithmetic is slower. However, the speedup from ECM to GEECM reduces the post-quantum security level of 2048-bit primes. Rather than engaging in a detailed analysis of this loss, we move up to 4096-bit primes, obviously putting GEECM far out of reach.

Implementation. We now discuss our implementation of post-quantum RSA. Our main result is successful generation of a 1-terabyte exponent-3 RSA key consisting of 4096-bit primes. We also have preliminary results for encryption and decryption, although so far only for smaller sizes.

Our computations were performed on a heterogeneous cluster. We give a description of the machines in Appendix A. The memory-intensive portions of our computations were carried out a single machine running Ubuntu with 24 cores at 3.40 GHz (4 Intel Xeon E7-8893 v2 processors), 3 terabytes of DRAM, and 4.9 terabytes of swap memory built from enterprise SSDs. We will refer to this machine as `lattice0` below. We measured memory consumption and overall

		Encryption	*Decryption*		
Key Size	**Bytes**		**Rem. tree**	**Cube root**	**CRT tree**
1MB	2^{20}	0.3	0.2	4.8	25.0
10MB	$2^{23.3}$	5	6	18	262
100MB	$2^{26.6}$	77	261	177	2851
1GB	2^{30}	654	812	1765	33586
4GB	2^{32}	3123	2318	8931	101309
8GB	2^{33}	6689	7214	17266	212215
16GB	2^{34}	18183	20420	34376	476798
32GB	2^{35}	29464	62729	62567	N/A
128GB	2^{37}	150975	N/A	N/A	N/A
256GB	2^{38}	362015	N/A	N/A	N/A

Table 4.1. Encryption and decryption times—We measure wall clock time in seconds on `lattice0` for encryption and the three stages of decryption: reducing the ciphertext modulo each prime factor, computing a cube root modulo each prime, and reconstructing the plaintext modulo the product.

runtime for bignum multiplications using GNU's Multiple Precision (GMP) Library [26]. We encountered a number of software limits and bugs, which we detail in Appendix A.

Prime generation. Generating a 1-terabyte exponent-3 RSA key requires 2^{31} 4096-bit primes that are congruent to 2 mod 3. To efficiently generate such a large number of primes, our implementation first applies the batched smoothness detection technique discussed in Section 3 to an input collection of random 4096-bit numbers. We then use the Fermat congruence primality test to produce our final set of primes. While we do not *prove* that each number in the final output is prime, this test is sufficient to guarantee with high confidence that all of the 4096-bit numbers in the final output are prime. See [31] for quantitative upper bounds on the error probability.

We found that first filtering for random numbers congruent to 5 mod 6, and then applying batch sieving with the successive bounds $y = 2^{10}$ and $y = 2^{20}$ worked well in practice. Our heterogeneous cluster was able to generate primes at a rate of 750–1585 primes per core-hour. Generating all 2^{31} primes took approximately 1,975,000 core-hours. In calendar time, prime generation completed in four months running on spare compute capacity of a 1,400-core cluster.

Product tree. After we successfully generated 2^{31} 4096-bit primes, we used a product tree to compute the 1-terabyte public RSA key. We distributed individual multiplications across our heterogeneous cluster to reduce the wall-clock time. We first multiplied batches of 8 million primes and wrote their products out to disk. Each subsequent single-threaded multiplication job read two integers from disk and wrote their product back to disk. Running times varied due to different CPU types and non-pqRSA related jobs sharing cache space. Once the integers reached 256GB in size, we finished computing the product on lattice0. The aggregate wall-clock time used by individual multiply jobs was about 1,239,626 seconds, and the elapsed time for the terabyte key generation was about four days. The final multiplication of two 512 GB integers took 176,223 seconds in wall-clock time, using 3.166TB of RAM and 2.5 TB of swap storage.

Encryption. We implemented RSA encryption using RSA-KEM, as described in Section 3. With the exponent $e = 3$, we found that a simple square-and-reduce using GMP's mpz_mult and mpz_mod was almost twice as fast as using the modular exponentiation function mpz_powm. Each operation was single-threaded. We were able to complete RSA encryption for modulus sizes up to 2 terabits, as shown in Table 4.1. For the 2Tb (256GB) encryption, the longest multiplication took 13 hours, modular reduction took 40 hours, and in total encryption took a little over 100 hours.

Decryption. We implemented RSA decryption as described in Section 3. Table 4.1 gives wall-clock timings for the three computational steps in decryption, each parallelized across 48 threads. Precomputing the entire product and remainder tree for a terabyte-sized key and storing it to disk would have taken 32TB of disk space, so instead we recomputed portions of the trees on the fly. The reported timings for the remainder tree step in Table 4.1 include the time it takes to recompute both the product and remainder tree with a batch size of 8 million primes. Using a batch size of 8 million primes was roughly twice as fast as using a batch size of 2 million primes. We obtained experimental results for decryption of messages for key sizes of up to 16GB.

References

[1] — (no editor), *Second international conference on quantum, nano, and micro technologies, ICQNM 2008, February 10–15, 2008, Sainte Luce, Martinique, French Caribbean*, IEEE Computer Society, 2008. See [17].

[2] — (no editor), *kernel BUG at mm/huge_memory.c:1798!* (2012). URL: http://linux-kernel.2935.n7.nabble.com/kernel-BUG-at-mm-huge-memory-c-1798-td574029.html. Citations in this document: §A.

[3] — (no editor), *Proceedings of the 23rd USENIX security symposium, August 20–22, 2014, San Diego, CA, USA*, USENIX, 2014. See [19].

[4] Michel Abdalla, Paulo S. L. M. Barreto (editors), *Progress in cryptology—LATINCRYPT 2010, first international conference on cryptology and information security in Latin America, Puebla, Mexico, August 8–11, 2010, proceedings*, Lecture Notes in Computer Science, 6212, Springer, 2010. See [11].

[5] Razvan Barbulescu, Joppe W. Bos, Cyril Bouvier, Thorsten Kleinjung, Peter L. Montgomery, *Finding ECM-friendly curves through a study of Galois properties* (2013), 63–86, *ANTS-X: proceedings of the tenth Algorithmic Number Theory Symposium*, 2013. URL: http://msp.org/obs/2013/1/p04.xhtml. Citations in this document: §2.

[6] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, Carl Pomerance, *The generation of random numbers that are probably prime*, Journal of Cryptology **1** (1988), 53–64. URL: https://math.dartmouth.edu/~carlp/probprime.pdf. Citations in this document: §3.

[7] Mihir Bellare, Daniel Kane, Phillip Rogaway, *Big-Key symmetric encryption: resisting key exfiltration*, in [**44**] (2016), 373–402. URL: https://eprint.iacr.org/2016/541.pdf. Citations in this document: §1.

[8] Daniel J. Bernstein, *How to find small factors of integers* (2002). URL: https://cr.yp.to/papers.html#sf. Citations in this document: §3.

[9] Daniel J. Bernstein, *How to find smooth parts of integers* (2004). URL: https://cr.yp.to/papers.html#smoothparts. Citations in this document: §3, §3.

[10] Daniel J. Bernstein, *Fast multiplication and its applications*, in [**18**] (2008), 325–384. URL: https://cr.yp.to/papers.html#multapps. Citations in this document: §3, §3, §3.

[11] Daniel J. Bernstein, Peter Birkner, Tanja Lange, *Starfish on strike*, in Latincrypt 2010 [**4**] (2010), 61–80. URL: https://eprint.iacr.org/2010/367. Citations in this document: §2.

[12] Daniel J. Bernstein, Peter Birkner, Tanja Lange, Christiane Peters, *ECM using Edwards curves* (2008). URL: https://eprint.iacr.org/2008/016. Citations in this document: §2, §2.

[13] Dan Boneh, Glenn Durfee, Nick Howgrave-Graham, *Factoring $N = p^r q$ for large r*, in [**54**] (1999), 326–337. URL: http://crypto.stanford.edu/~dabo/abstracts/prq.html. Citations in this document: §3.

[14] Joppe W. Bos, Thorsten Kleinjung, *ECM at work*, in Asiacrypt 2012 [**53**] (2012), 467–484. URL: https://eprint.iacr.org/2012/089. Citations in this document: §2.

[15] Sergai Boukhonine, *Cryptography: a security tool of the information age* (1998). URL: https://pdfs.semanticscholar.org/3932/8253d692f791b37c425e776f6cee0b8c3e56.pdf. Citations in this document: §1.

[16] Gilles Brassard, Peter Høyer, Kassem Kalach, Marc Kaplan, Sophie Laplante, Louis Salvail, *Merkle puzzles in a quantum world*, in Crypto 2011 [**45**] (2011), 391–410. URL: https://arxiv.org/abs/1108.2316. Citations in this document: §1, §1.

[17] Gilles Brassard, Louis Salvail, *Quantum Merkle puzzles*, in ICQNM 2008 [**1**] (2008), 76–79. Citations in this document: §1.

[18] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, New York, 2008. See [10].

[19] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, *On the practical exploitability of Dual EC in TLS implementations*, in USENIX Security 2014 [**3**] (2014). URL: https://projectbullrun.org/dual-ec/index.html. Citations in this document: §1.

[20] Artur Ekert, *Quantum cryptoanalysis—introduction* (2010). URL: http://www.qi.damtp.cam.ac.uk/node/69. Citations in this document: §1.

[21] Martin Fürer, *Faster integer multiplication*, in [**30**] (2007), 57–66. URL: https://www.cse.psu.edu/~furer/. Citations in this document: §3.

[22] Alexandre Gélin, Thorsten Kleinjung, Arjen K. Lenstra, *Parametrizations for families of ECM-friendly curves* (2016). URL: https://eprint.iacr.org/2016/1092. Citations in this document: §2.

[23] Shafi Goldwasser (editor), *35th annual IEEE symposium on the foundations of computer science. Proceedings of the IEEE symposium held in Santa Fe, NM, November 20–22, 1994*, IEEE, 1994. ISBN 0-8186-6580-7. MR 98h:68008. See [48].

[24] Dan Goodin, *Symantec employees fired for issuing rogue HTTPS certificate for Google* (2015). URL: https://arstechnica.com/security/2015/09/symantec-employees-fired-for-issuing-rogue-https-certificate-for-google/. Citations in this document: §1.

[25] Torbjörn Granlund, *gmp integer size limitation* (2012). URL: https://gmplib.org/list-archives/gmp-discuss/2012-April/005020.html. Citations in this document: §A.

[26] Torbjörn Granlund, the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library* (2015). URL: https://gmplib.org/. Citations in this document: §4.

[27] David Harvey, Joris van der Hoeven, Grégoire Lecerf, *Even faster integer multiplication*, Journal of Complexity **36** (2016), 1–30. URL: https://arxiv.org/abs/1407.3360. Citations in this document: §3.

[28] David Harvey, Joris van der Hoeven, Grégoire Lecerf, *Fast polynomial multiplication over* $\mathbf{F}_{2^{60}}$, proceedings of ISSAC 2016, to appear (2016). URL: https://hal.archives-ouvertes.fr/hal-01265278. Citations in this document: §4, §4.

[29] id Quantique, *Future-proof data confidentiality with quantum cryptography* (2005). URL: https://classic-web.archive.org/web/20070728200504/http://www.idquantique.com/products/files/vectis-future.pdf. Citations in this document: §1.

[30] David S. Johnson, Uriel Feige (editors), *Proceedings of the 39th annual ACM symposium on theory of computing, San Diego, California, USA, June 11–13, 2007*, Association for Computing Machinery, New York, 2007. ISBN 978-1-59593-631-8. See [21].

[31] Su Hee Kim, Carl Pomerance, *The probability that a random probable prime is composite*, Mathematics of Computation **53** (1989), 721–741. URL: https://math.dartmouth.edu/~carlp/PDF/paper72.pdf. Citations in this document: §4.

[32] Hugo Krawczyk (editor), *Advances in cryptology—CRYPTO '98, 18th annual international cryptology conference, Santa Barbara, California, USA, August 23–27, 1998, proceedings*, Lecture Notes in Computer Science, 1462, Springer, 1998. ISBN 3-540-64892-5. MR 99i:94059. See [52].

[33] Derrick H. Lehmer, R. E. Powers, *On factoring large numbers*, Bulletin of the American Mathematical Society **37** (1931), 770–776. Citations in this document: §2.

[34] Arjen K. Lenstra, Hendrik W. Lenstra, Jr. (editors), *The development of the number field sieve*, Lecture Notes in Mathematics, 1554, Springer-Verlag, Berlin, 1993. ISBN 3-540-57013-6. MR 96m:11116. Citations in this document: §2.

[35] Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673. MR 89g:11125. Citations in this document: §2.

[36] Hendrik W. Lenstra, Jr., R. Tijdeman (editors), *Computational methods in number theory I*, Mathematical Centre Tracts, 154, Mathematisch Centrum, Amsterdam, 1982. ISBN 90-6196-248-X. MR 84c:10002. See [41].

[37] Franck Leprévost, *The end of public key cryptography or does God play dices?*, PricewaterhouseCoopers Cryptographic Centre of Excellence Quaterly Journal (1999). URL: http://tinyurl.com/jdkkxc3. Citations in this document: §1.

[38] Ueli M. Maurer, *Fast generation of prime numbers and secure public-key cryptographic parameters*, Journal of Cryptology **8** (1995), 123–155. URL: http://link.springer.com/article/10.1007/BF00202269. Citations in this document: §3.

[39] John M. Pollard, *Theorems on factorization and primality testing*, Proceedings of the Cambridge Philosophical Society **76** (1974), 521–528. MR 50 #6992. Citations in this document: §2.

[40] John M. Pollard, *A Monte Carlo method for factorization*, BIT **15** (1975), 331–334. MR 52 #13611. Citations in this document: §2.

[41] Carl Pomerance, *Analysis and comparison of some integer factoring algorithms*, in [**36**] (1982), 89–139. MR 84i:10005. Citations in this document: §2.

[42] Michael O. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, Technical Report 212, MIT Laboratory for Computer Science, 1979. URL: https://archive.org/details/bitsavers_mitlcstrMI_457188. Citations in this document: §3.

[43] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM **21** (1978), 120–126. ISSN 0001-0782. URL: https://people.csail.mit.edu/rivest/Rsapaper.pdf. Citations in this document: §3.

[44] Matthew Robshaw, Jonathan Katz (editors), *Advances in cryptology—CRYPTO 2016—36th annual international cryptology conference, Santa Barbara, CA, USA, August 14–18, 2016, proceedings, part I*, Lecture Notes in Computer Science, 9814, Springer, 2016. ISBN 978-3-662-53017-7. See [7].

[45] Phillip Rogaway (editor), *Advances in cryptology—CRYPTO 2011, 31st annual cryptology conference, Santa Barbara, CA, USA, August 14–18, 2011, proceedings*, Lecture Notes in Computer Science, 6841, Springer, 2011. See [16].

[46] Arnold Schönhage, Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), 281–292. ISSN 0010-485X. MR 45:1431. URL: http://link.springer.com/article/10.1007/BF02242355. Citations in this document: §3.

[47] Adi Shamir, *RSA for paranoids*, CryptoBytes **1** (1995). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.5763&rep=rep1&type=pdf. Citations in this document: §1, §3.

[48] Peter W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, in [**23**] (1994), 124–134; see also newer version [**49**]. MR 1489242. Citations in this document: §1.

[49] Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer* (1995); see also older version [**48**]; see also newer version [**50**]. URL: https://arxiv.org/abs/quant-ph/9508027v2.

[50] Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), 1484–1509; see also older version [**49**]. MR 98i:11108.

[51] Victor Shoup, *A proposal for an ISO standard for public key encryption (version 2.1)* (2001). URL: http://www.shoup.net/papers. Citations in this document: §3.

[52] Tsuyoshi Takagi, *Fast RSA-type cryptosystem modulo $p^k q$*, in [**32**] (1998), 318–326. URL: http://imi.kyushu-u.ac.jp/~takagi/takagi/publications/cr98.ps. Citations in this document: §1, §3.

[53] Xiaoyun Wang, Kazue Sako (editors), *Advances in cryptology—ASIACRYPT 2012, 18th international conference on the theory and application of cryptology*

Level	Time (s)	Level	Time (s)	Level	Time (s)	Level	Time (s)
1	4417.1	9	750.3	17	2121.7	25	4482.4
2	4039.3	10	1035.7	18	2188.4	26	5548.5
3	312.9	11	918.1	19	2392.1	27	9019.0
4	2709.8	12	1078.5	20	2463.8	28	16453.6
5	446.5	13	1180.3	21	2485.0	29	32835.6
6	1003.4	14	1291.4	22	2533.5	30	69089.7
7	647.7	15	1402.2	23	2632.7	31	123100.4
8	998.7	16	1503.6	24	3078.2		

Table A.1. Time per product-tree level in key generation—We record the time for each product-tree level in a 1-terabyte key generation using `lattice0`. Level 1 takes 1,953,125,000 4096-bit numbers as input, and produces 976,562,500 8192-bit numbers as output. Level 31 takes two 500GB numbers and multiplies them to create the final 1TB output.

and information security, Beijing, China, December 2–6, 2012, proceedings, Lecture Notes in Computer Science, 7658, Springer, 2012. ISBN 978-3-642-34960-7. See [14].

[54] Michael Wiener (editor), *Advances in cryptology—CRYPTO '99, 19th annual international cryptology conference, Santa Barbara, California, USA, August 15–19, 1999, proceedings*, Lecture Notes in Computer Science, 1666, Springer, 1999. ISBN 3-540-66347-9. MR 2000h:94003. See [13].

[55] Hugh C. Williams, *A $p+1$ method of factoring*, Mathematics of Computation **39** (1982), 225–234. MR 83h:10016. Citations in this document: §2.

[56] Christof Zalka, *Fast versions of Shor's quantum factoring algorithm* (1998). URL: `https://arxiv.org/abs/quant-ph/9806084`. Citations in this document: §2, §4.

[57] Paul Zimmermann, *About memory-usage of mpz_mul* (2016). URL: `https://gmplib.org/list-archives/gmp-discuss/2016-June/006009.html`. Citations in this document: §A.

A Appendix: Implementation barriers and details

Extending GMP's integer capacity. The GMP library uses hard-coded 32-bit integers to represent sizes in multiple locations in the library. Without any modifications, GMP supports 2^{37}-bit integers on 64-bit machines [**25**]. To represent large values, we extended GMP's capacity from 32-bit integers to 64-bit integers by changing the data typing in GMP's integer structure, `mpz`. Namely, we changed `mpz_size` and `mpz_alloc` from `int` types to `int64_t` types. To accommodate increased memory usage, we increased the bound for GMP's memory allocation for the `mpz` struct in `realloc.c` to `LLONG_MAX`. The final modifications we made were to create binary-format I/O functions for 64-bit `mpz`s, namely in `mpz_inp_out.c` and `mpz_out_raw.c`.

Impact of swapping. We initially evaluated the performance of our product-tree implementation by generating a "dummy key", a terabyte product of random 4096-bit integers. During this product computation, we counted instructions

Name	CPU type	Physical cores	RAM	Count
`lattice0`	3.40GHz Intel Xeon E7-8893 v2	quad 6-core	3TB	1
`raminator`	2.60GHz Intel Xeon E7-4860 v2	quad 12-core	1TB	1
`siv-1-[1-8]`	2.50GHz Intel Xeon E5-2680 v3	dual 12-core	512GB	8
`lattice[1-6]`	2.30GHz Intel Xeon E5-2699 v3	dual 18-core	256GB	6
`siv-[2-3]-[1-8]`	2.20GHz Intel Xeon E5-2699 v4	dual 22-core	512GB	16
`utah[1-4]`	2.20GHz Intel Xeon E5-2699 v4	dual 22-core	512GB	4

Table A.2. Heterogeneous compute cluster—The experiments in this paper were carried out on a heterogeneous cluster.

per CPU cycle (IPCs) with the command `perf stat -e instructions,cycles -a sleep 1` to measure the lost performance caused by swapping. When no swapping occurred, the machine had about 2 instructions per cycle, but upon swapping, the instructions per cycles dropped as low as 0.37 instructions per cycle and held around 0.5 to 1.2 instructions per cycle.

GMP memory consumption. GMP's memory consumption is another concern. High RAM and swap usage at higher levels in the product tree are attributed to GMP's FFT implementation. According to GMP's developers, their FFT implementation consumes about $8n$ bytes of temporary memory space for an $n{\cdot}n$ product where n is the byte size of the factors [57]. This massive consumption of memory also triggered a known race condition in the Linux kernel [2]. The bug was found in the `huge_memory.c` code. There are numerous bug reports for variants of the same bug on various mainline Linux systems throughout the past six years. Disabling transparent huge pages avoided the `transparent_hugepage` code in the kernel.

Measurements for 1-terabyte key product tree. In Table A.1, we show the wall-clock time for each level of computing a 1-terabyte product tree. Levels far down in the product tree are easily parallelized. We carried out the entire computation on `lattice0` using 48 threads. The computation used a peak of 3.16TB of RAM and 2.22TB of swap memory, and completed in 356,709 seconds, or approximately 4 days, in wall-clock time.

Heterogeneous cluster description. See Table A.2.

B Credits for multi-prime RSA

The idea of using RSA with more than two primes is most commonly credited to Collins, Hopkins, Langford, and Sabin, who received patent 5848159 in 1998 for "RSA with several primes":

> The invention, allowing 4 primes each about 150 digits long to obtain a 600 digit n, instead of two primes about 350 [*sic*] digits long, results in a marked improvement in computer performance. For, not only are

> primes that are 150 digits in size easier to find and verify than ones on the order of 350 digits, but by applying techniques the inventors derive from the Chinese Remainder Theorem (CRT), public key cryptography calculations for encryption and decryption are completed much faster—even if performed serially on a single processor system.

However, the same idea had already appeared in the original RSA patent in 1983:

> In alternative embodiments, the present invention may use a modulus n which is a product of three or more primes (not necessarily distinct). Decoding may be performed modulo each of the prime factors of n and the results combined using "Chinese remaindering" or any equivalent method to obtain the result modulo n.

In any event, both of these patents have now expired, so they will not interfere with the deployment of post-quantum RSA.