The RSA Algorithm

Evgeny Milanov

3 June 2009

In 1978, Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman introduced a cryptographic algorithm, which was essentially to replace the less secure National Bureau of Standards (NBS) algorithm. Most importantly, RSA implements a public-key cryptosystem, as well as digital signatures. RSA is motivated by the published works of Diffie and Hellman from several years before, who described the idea of such an algorithm, but never truly developed it.

Introduced at the time when the era of electronic email was expected to soon arise, RSA implemented two important ideas:

1. Public-key encryption. This idea omits the need for a "courier" to deliver keys to recipients over another secure channel before transmitting the originally-intended message. In RSA, encryption keys are public, while the decryption keys are not, so only the person with the correct decryption key can decipher an encrypted message. Everyone has their own encryption and decryption keys. The keys must be made in such a way that the decryption key may not be easily deduced from the public encryption key.

2. Digital signatures. The receiver may need to verify that a transmitted message actually originated from the sender (signature), and didn't just come from there (authentication). This is done using the sender's decryption key, and the signature can later be verified by anyone, using the corresponding public encryption key. Signatures therefore cannot be forged. Also, no signer can later deny having signed the message.

This is not only useful for electronic mail, but for other electronic transactions and transmissions, such as fund transfers. The security of the RSA algorithm has so far been validated, since no known attempts to break it have yet been successful, mostly due to the difficulty of factoring large numbers $n = pq$, where p and q are large prime numbers.

1 Public-key cryptosystems.

Each user has their own encryption and decryption procedures, E and D, with the former in the public file and the latter kept secret. These procedures are related to the keys, which, in RSA specifically, are sets of two special numbers. We of course start out with the message itself, symbolized by M, which is to be "encrypted". There are four procedures that are specific and essential to a public-key cryptosystem:

a) Deciphering an enciphered message gives you the original message, specifically

$$D(E(M)) = M \ . \tag{1}$$

b) Reversing the procedures still returns M:

$$E(D(M)) = M \ . \tag{2}$$

c) E and D are easy to compute.

d) The publicity of E does not compromise the secrecy of D, meaning you cannot easily figure out D from E.

With a given E, we are still not given an efficient way of computing D. If $C = E(M)$ is the ciphertext, then trying to figure out D by trying to satisfy an M in $E(M) = C$ is unreasonably difficult: the number of messages to test would be impractically large.

An E that satisfies (a), (c), and (d) is called a "trap-door one-way function" and is also a "trap-door one-way permutation". It is a trap door because since it's inverse D is easy to compute if certain "trap-door" information is available, but otherwise hard. It is one-way because it is easy to compute in one direction, but hard in the other. It is a permutation because it satisfies (b), meaning every ciphertext is a potential message, and every message is a ciphertext of some other message. Statement (b) is in fact just needed to provide "signatures".

Now we turn to specific keys, and imagine users A and B (Alice and Bob) on a two-user public-key cryptosystem, with their keys: E_A, E_B, D_A, D_B.

2 Privacy.

Encryption, which is now a ubiquitous way of assuring a message is delivered privately, makes it so no intruder can bypass the ciphertext, which is essentially white noise. Without property (d), however, an encryption process is still not public-key, such as the NBS standard. It requires keys to be delivered privately through another secure "courier", which is an extra process that would deem NBS, for example, as slow, inefficient, and possibly expensive. Thus, RSA is a great answer to this problem. The NBS standard could provide useful only if it was a faster algorithm than RSA, where RSA would only be used to securely transmit the keys only. Thus, an efficient computing method of D must be found, so as to make RSA completely stand-alone and reliable. For it to be reliable, it would have to use simple arithmetic, which is easier to compute (a requirement of property (c)) on a general-purpose computer than are bit manipulations, where better hardware is used, where perhaps NBS would be better.

Now, Bob wants to send a private message to Alice. He will retrieve E_A from the public file, encode M, getting $C = E_A(M)$, whereafter Alice decodes it with her own D_A, which only she can do, due to property (d). She could also reply to Bob, using E_B. So all that's needed is both user's agreement to be part of the cryptosystem by placing their encryption data into a public file. No beforehand communication is needed, private or not. Also, due to property (d), no eavesdropper can deduce D from listening in on E.

3 Signatures.

For complete assurance that the message originated form a sender, and was not just sent through him by a third party who may have used the same encryption key (that of the receiver), we need a digital signature to come with the message. This has obvious implications of importance in real-life applications.

Bob wants to send a private message to Alice. To sign the document, we pull a clever little trick, all assuming that the RSA algorithm is quick and reliable, mostly due to property (c). We decrypt a message with Bob's key, allowed by properties (a) and (b), which assert that every message is the ciphertext of another message, and that every ciphertext can be interpreted as a message. Formally,

$$D_B(M) = S \; . \tag{3}$$

Then we encrypt S with Alice's encryption key.

$$E_A(S) = E_A(D_B(M)) \tag{4}$$

This way, we can assure only she can decrypt the document. When she does, she gets the signature by $D_A(E_A(D_B(M)) = S$. She now knows the message came from Bob, since only his decryption key could compute the signature. The message need not be sent separately, since Alice can deduce it from the signature itself by using Bob's publicly available encryption key, formally $E_B(S) = E_B(D_B(M)) = M$. Since S depends on M, and the encrypted transmission Bob sent depends on S, we have a transmission that depends on both the message and the signature, so both can be deduced from the transmitted document.

This brilliantly assures the message could not be modified (if needed to be presented to, say, a "judge"), since a modified M in the form of M' would have to generate a signature $S' = D_B(M')$ as well, which is impossible, since she does not known D_B by property (d).

So not only does Alice possess proof that Bob signed the message and indeed sent it, but she also cannot modify M nor forge a signature for any other message.

Now, say an "intruder" attempted to lie and tell you he was from the public file? This is not a problem in RSA, since "signatures' are used. A signature just needs to assure it came form the public file (PF) itself. Every time a user joins a network, everybody gets a securely sent copy of the most recently updated PF, which is stored on their system, and they never have to look it up. Anyone trying to send a message pretending to be in the public file would not have the appropriate signature, and would be singled out as an "intruder". He would also never receive the PF, since he never joined it.

4 Applications, predictions, hardware implementation.

This has applications to electronic fund transmissions as well. Financial information needs to be secure, and checks can be electronically signed with RSA. Further measures would have to be taken, such as implementing unique check numbers that allow a check with this certain number transmittable/cashable only once.

In fact, such a system can be applied to *any* electronic system that needs to have a cryptosystem implemented. In their 1978 RSA paper, the authors of RSA predicted a secure email world to evolve and for RSA to be used to encrypt a live telephone conversation. Now, these things are indeed a part of more than just daily life because of RSA.

The encryption device must not be the direct buffer between a terminal and the communications channel. Instead, it should be a hardware subroutine that can be executed as needed, since it may need to be encrypted/decrypted with several different sequences of keys, so as to assure more privacy and/or more signatures.

5 The math of the method.

So far, we expect to make E and D easy to compute through simple arithmetic. We must now represent the message numerically, so that we can perform these arithmetic algorithms on it. Now lets represent M by an integer between 0 and $n-1$. If the message is too long, sparse it up and encrypt separately. Let e, d, n be positive integers, with (e, n) as the encryption key, (d, n) the decryption key, $n = pq$.

Now, we encrypt the message by raising it to the eth power modulo n to obtain C, the ciphertext. We then decrypt C by raising it to the dth power modulo n to obtain M again. Formally, we obtain these encryption and decryption algorithms for E and D:

$$\begin{aligned} C &\equiv E(M) &\equiv M^e \pmod{n} \\ M &\equiv D(C) &\equiv C^d \pmod{n}\ . \end{aligned} \tag{5}$$

Note that we are preserving the same information size, since M and C are integers between 0 and $n-1$, and because of the modular congruence. Also note the simplicity of the fact that the encryption/decryption keys are both just pairs of integers, (e, n) and (d, n). These are different for every user, and should generally be subscripted, but we'll consider just the general case here.

Now comes the question of creating the encryption key itself. First, choosing two "random" large primes p and q, we multiply and produce $n = pq$. Although n is public, it will not reveal p and q since it is essentially impossible to factor them form n, and therefore will assure that d is practically impossible to derive from e.

Now we want to obtain the appropriate e and d. We pick d to be a random large integer, which must be coprime to $(p-1)\cdot(q-1)$, meaning the following equation has to be satisfied:

$$\gcd(d, (p-1)\cdot(q-1)) = 1\ . \tag{6}$$

"gcd" means greatest common divisor.

The reason we want d to be coprime to $(p-1)\cdot(q-1)$ is peculiar. I will not show the "direct motivation" behind it; rather, it will become clear why that statement is important when I show towards the end of this section that it guarantees (1) and (2).

We will want to compute e from d, p, and q, where e is the multiplicative inverse of d. That means we need to satisfy

$$e \cdot d = 1 \pmod{\phi(n)} \; . \tag{7}$$

Here, we introduce the Euler totient function $\phi(n)$, whose output is the number of positive integers less than n which are coprime to n. For primes p, this clearly becomes $\phi(p) = p - 1$. For n, we obtain, by elementary properties of the totient function, that

$$\begin{aligned} \phi(n) &= \phi(p) \cdot \phi(q) \\ &= (p-1) \cdot (q-1) \\ &= n - (p+q) + 1 \; . \end{aligned} \tag{8}$$

From this equation, we can substitute $\phi(n)$ into equation (7) and obtain

$$e \cdot d \equiv 1 \pmod{\phi(n)}$$

which is equivalent to

$$e \cdot d = k \cdot \phi(n) + 1$$

for some integer k.

By the laws of modular arithmetic, the multiplicative inverse of a modulo m exists if and only if a and m are coprime. Indeed, since d and $\phi(n)$ are coprime, d has a multiplicative inverse e in the ring of integers modulo $\phi(n)$.

So far, we can safely assured the following:

$$\begin{aligned} D(E(M)) \equiv (E(M))^d \equiv (M^e)^d \pmod{n} = M^{e \cdot d} \pmod{n} \\ E(D(M)) \equiv (D(M))^e \equiv (M^d)^e \pmod{n} = M^{e \cdot d} \pmod{n} \end{aligned}$$

Also, since $e \cdot d = k \cdot \phi(n) + 1$, we can substitute into the above equations and obtain

$$M^{e \cdot d} \equiv M^{k \cdot \phi(n) + 1} \pmod{n} \; .$$

Clearly, we want that to equal M. To prove this, will need an important identity due to Euler and Fermat: for any integer M coprime to n, we have

$$M^{\phi(n)} \equiv 1 \pmod{n} \; . \tag{9}$$

Since we previously specified that $0 \leq M < n$, we know that M would *not* be coprime to n if and only if M was either p or q, of the integers in that interval. Therefore, the chances of M happening to be p or

q are on the same order of magnitude as $2/n$. This means that M is almost definitely relatively prime to n, therefore equation (9) holds and, using it, we evaluate:

$$M^{e \cdot d} \equiv M^{k \cdot \phi(n)+1} \equiv (M^{\phi(n)})^k M \equiv 1^k M \pmod{n} = M \ .$$

It turns out this works for all M, and in fact we see that (1) and (2) hold for all $M, 0 \leq M < n$. Therefore E and D are inverse permutations.

6 Algorithms.

6.1 Efficient encryption and decryption operations.

The authors of RSA claim that "computing $M^e \pmod{n}$ requires at most $2 \cdot \log_2(e)$ multiplications and $2 \cdot \log_2(e)$ divisions" if we use their procedure below. It is important for us to know the amount of steps it would take a computer to encrypt the message so we can see if a method is fast and efficient, or not. We now "exponentiate by repeated squaring and multiplication":

```
Step 1. Let $e_k e_{k-1} ... e_1 e_0$ be the binary representation of $e$.
Step 2. Set the variable $C$ to 1.
Step 3. Repeat steps 3a and 3b for $i = k, k-1, ..., 0$:
    Step 3a. Set $C$ to the remainder of $C^2$ when divided by $n$.
    Step 3b. If $e_i = 1$ then set $C$ to the remainder of $C \cdot M$ when divided by $n$.
Step 4. Halt. Now $C$ is the encrypted form of $M$.
```

There are more efficient procedures out there, but this one is good too. Also, since decryption follows the same identical procedure as encryption, we can implement the whole operation on a few integrated chips.

According to the authors of RSA, "the encryption time per block increases no faster than the cube of the number of digits in n.

6.2 Finding large prime numbers.

Finding n is the first step to the entire process. The number n will be revealed in the encryption and decryption keys, but the numbers p and q, whose product make up n, will not be explicitly shown. They are essentially impossible to derive from n, in fact, especially if we pick, say, 100-digit primes p and q, which would make a 200-digit n. These figures were at least sufficient in 1978. However, today we must use far larger numbers. The scale of these numbers is mentioned in the last section of this article.

Each user needs to privately choose his own two large prime numbers p and q. To do this, we need to generate, say, random odd 100-digit numbers until a prime is found. We will have to test each number, and according to the prime number theorem, there will be about $(\ln 10^{100})/2 = 115$ number to test.

To test a large b for primality, we can use an algorithm due to Solovay and Strassen. First, we pick a random number a from a uniform distribution on $1, ..., b-1$ and test whether

$$\gcd(a, b) = 1 \ \text{ and } \ J(a, b) = a^{(b-1) \div 2} \pmod{b} \ , \tag{10}$$

where $J(a,b)$ is the Jacobi symbol, which can also be represented as

$$\left(\frac{a}{b}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1}\left(\frac{a}{p_2}\right)^{\alpha_1} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k},$$

where $b = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ is the primal factorization of b, and the Legendre symbol is defined for all integers a and all odd primes p by

$$\begin{aligned} \left(\frac{a}{p}\right) &= \begin{cases} 0 \text{ if } a \equiv 0 \pmod p \\ +1 \text{ if } a \not\equiv 0 \pmod p \text{ and for some integer } x,\ a \equiv x^2 \pmod p \\ -1 \text{ if there is no such } a \equiv 0 \pmod p \end{cases} \\ \left(\frac{a}{1}\right) &\equiv 1\,. \end{aligned}$$

The Jacobi symbol is only defined when a is an integer and b is a positive odd integer. Also, $J(a,b)$ is 0 if $\gcd(a,b) \neq 1$ and ± 1 if $\gcd(a,b) = 1$. Equation (10) is always true if b is prime, otherwise (if b is composite), (10) will have a chance of being false of over 50%. If (10) is true 100 times for randomly chosen a's, then b is almost certainly prime, with a chance of being composite of 1 in 2^{100}. If accidentally a composite were used for p or q in the process, the recipient would see "junk" and realize the decryption wasn't done correctly. I now present an efficient program for computing $J(a,b)$ that the authors of RSA recommended in their original article:

```
$J(a,b)$ = **if** $a = 1$ **then** 1 **else**
           **if** is even **then** $J(a \div 2, b) \cdot (-1)^{(b^2-1)\div 8}$
           **else** $J(b \pmod a, a) \cdot (-1)^{(a-1)\cdot(b-1)\div 4}$
```

To protect against sophisticated factoring algorithms, p and q should differ in length by a few digits, $\gcd(p-1, q-1)$ should be small, and both $(p-1)$ and $(q-1)$ should contain large prime factors. To assure the latter, we generate a large random prime number u and take the first prime in the sequence $i \cdot u + 1,\ i = 2, 4, 6, \ldots$. This process would be very fast on a computer. When the authors of RSA published their article, on a high-speed computer, testing a 100-digit number for primality would take several seconds, while finding the next prime would take around a minute and a half. Imagine how quickly an average PC can do this today! Perhaps this is why the RSA algorithm has survived so long.

We could also find large primes by taking a number whose factorization we know, add 1 to it, and test for primality. If we get a number we think is prime, we *could* potentially prove that it is prime by using the factorization of $(p-1)$.

6.3 Finding d.

This is very easy. We want to find a number d coprime to $\phi(n)$; any prime number greater than $\max(p,q)$ is fine. Since the set of primes $\mathbb{P}$ is large, it assures that a cryptanalyst cannot find d by a direct search. In fact, any method of finding d that picks d out of a big set would do.

6.4 Finding e form d and $\phi(n)$.

Here, we use a variation of Euclid's algorithm for computing the greatest common divisor of $\phi(n)$ and d. First, we compute a series $x_0, x_1, x_2, ...,$ where $x_0 \equiv \phi(n), x_1 = d, ..., x_{i+1} \equiv x_{i-1} \pmod{x_i}$, until an $x_k = 0$ is found. Then $\gcd(x_0, x_1) = x_{k-1}$. Now we find numbers a_i and b_i such that $x_i = a_i \cdot x_0 + b_i \cdot x_1$. If $x_{k-1} = 1$ then b_{k-1} is the multiplicative inverse of $x_1 \pmod{n}$, and is precisely e. Since $k < 2\log_2(n)$, this can be computed quickly. Since the difficulty in computing complicated modular arithmetic in part contributes to the difficulty in cracking RSA, we need to use this to our advantage. Therefore, if $e < \log_2(n)$, we find another e that's not too small so that the encrypted message undergoes reduction in modulo n ("wrap-around").

7 An example.

Let $p = 37, q = 43, n = p \cdot q = 1591, d = 71$. It follows that $\phi(1591) = 36 \cdot 42 = 1512$ and now we compute e:

$$\begin{array}{lll}
x_0 = 1512, & a_0 = 1, & b_0 = 0, \\
x_1 = 71, & a_1 = 0, & b_1 = 1, \\
x_2 = 21, & a_2 = 1, & b_2 = -21, \\
x_3 = 8, & a_3 = -3, & b_3 = 64, \\
x_4 = 5, & a_4 = 7, & b_4 = -149, \\
x_5 = 3, & a_5 = -10, & b_5 = 213, \\
x_6 = 2, & a_6 = 17, & b_6 = -362, \\
x_7 = 1, & a_7 = -27, & b_7 = 575 \ .
\end{array}$$

Thus $e = 575$ is the multiplicative inverse in modulo 1512 of $d = 71$. Using this and the rest of our values, we now begin to encrypt. We use a fairly standard numerical representation of the English alphabet: blank = 00, A = 01, B = 02, ... , Z = 26. Each block of the message has to be less than $n = 1591$. In this case, we will break the message into two-letter blocks, because luckily no block will exceed 159. The bigger p and q are, the more information we can encrypt per block. We choose a short message for ease:

```
NO ODD
```

is encoded:

```
1415 0015 0404
```

575 is 1000111111 in binary. We now use the algorithm in section 6.1. We simplify it using modular arithmetic. We replace steps 3a and 3b by the following conditions: if $e_k = 1$ then we square C and multiply by M, otherwise we just square C. After step 4, we put C into modulo n.

Taking, for example, the first block and setting $M = 1415$ and encrypting we get:

$$M^{575} = (((((((((1^2 \cdot M)^2)^2)^2)^2 \cdot M)^2 \cdot M)^2 \cdot M)^2 \cdot M)^2 \cdot M)^2 \cdot M = 824 \pmod{1591} \ .$$

The whole message becomes:

```
0824 1253 0267
```

Similarly, we can decrypt the message to check it deciphers properly: $824^{71} \equiv 1415 \pmod{1591}$, etc.

8 How secure is RSA?

The RSA algorithm is indeed among the strongest, but can it withstand anything? Certainly nothing can withstand the test of time. In fact, no encryption technique is even perfectly secure from an attack by a realistic cryptanalyst. Methods such as brute-force are simple but lengthy and may crack a message, but not likely an entire encryption scheme. We must also consider a probabilistic approach, meaning there's always a chance some one may get the "one key out of a million". So far, we don't know how to prove whether an encryption scheme is unbreakable. If we cannot prove it, we will at least see if someone can break the code. This is how the NBS standard and RSA were essentially certified. Despite years of attempts, no one has been known to crack either algorithm. Such a resistance to attack makes RSA secure in practice.

In section 8, we will see why breaking RSA is at least as hard as factoring n. Factoring large numbers is not provably hard, but no algorithms exists today to factor a 200-digit number in a reasonable amount of time. Fermat and Legendre have both contributed to this field by developing factoring algorithms, though factoring is still an age-old math problem. This is precisely what has partially "certified" RSA as secure.

To show that RSA is secure, we will consider how a cryptanalyst may try to obtain the decryption key from the public encryption key, and not how an intruder may attempt to "steal" the decryption key. This should be taken care of as one would protect their money, through physical security methods. The authors of RSA provide an example: the encryption device (which could be, say, a set of integrated chips within a computer), would be separate from the rest of the system. It would generate encryption and decryption keys, but would not print out the decryption key, even for its owner. It would, in fact, erase the decryption key if it sensed an attempted intrusion.

8.1 Factoring n.

Since knowing the factors of n would give away $\phi(n)$ and therefore d, a cryptanalyst would break the code if he factored n. However, factoring numbers has practically proven to be far harder than determining primality or compositeness. Nonetheless, many factoring algorithms are around. The authors of RSA referenced Knuth and Pollard as good sources for such algorithms. They also present an unpublished algorithm due to Richard Schroeppel, which factors n in approximately

$$\exp\sqrt{\ln n \cdot \ln\ln n} = n^{\sqrt{\ln\ln n \div \ln n}} = (\ln n)^{\sqrt{\ln n \div \ln\ln n}}$$

steps. The following table is the one the authors of RSA presented in 1978. They assume an operation in the Schroeppel factoring algorithm takes one microsecond to compute, and present the following data for various lengths of n:

Digits	**Number of operations**	**Time**
50	1.4×10^{10}	3.9 hours
75	9.0×10^{12}	104 days
100	2.3×10^{15}	74 years
200	1.2×10^{23}	3.8×10^{9} years
300	1.5×10^{29}	4.9×10^{15} years
500	1.3×10^{39}	4.2×10^{25} years

The authors of RSA recommend that n be about 200 digits long. However, the length of n may be varied, based on the importance of speed of encryption versus security. RSA, in effect, allows the user (administrator, etc.) to choose a key-length, and thus a level of security, a flexibility not found in many encryption schemes before 1978 (such as the NBS method).

8.2 Computing $\phi(n)$ without factoring n.

This method would break the system because if one could compute $\phi(n)$, then he could compute d as the multiplicative inverse of the publicly revealed e modulo $\phi(n)$, using the method developed in section 6.4. This method is at least as difficult as factoring the publicly-revealed n since it would allow a cryptanalyst to easily factor n from $\phi(n)$. Since this approach to factoring n has not turned out practical, this method of computing $\phi(n)$ is also impractical, since it is more difficult, by the previous sentence.

Indeed, factoring n as such is difficult because performing certain operations involving p, q, n, and even $\phi(n)$, all of immense sizes, would take too long. To factor n using $\phi(n)$, we first obtain $(p+q)$ from n and $\phi(n) = n - (p+q) + 1$, then compute $(p-q)$ from this easily verifiable equation:

$$(p-q)^2 = (p+q)^2 - 4n \ .$$

Finally, we obtain q from half the difference of $(p-q)$ and $(p+q)$, and we get p from either $p = n \div q$ or from half the sum of $(p-q)$ and $(p+q)$.

If n was prime, $\phi(n) = n-1$ would be easy to compute, thus n must be composite, as originally stated.

8.3 Determining d without factoring n or computing $\phi(n)$.

We use the same logic in the previous section. We have an approach to factoring n from a for-some-reason known d, described in the next paragraph, which has not proven practical. We argue that computing d is no easier than factoring n, since knowing d allows n to be factored easily.

Knowing d allows us to calculate $e \cdot d - 1$, which is a multiple of $\phi(n)$. Miller [6] shows that n can be factored using any such multiple, thus a cryptanalyst cannot determine d easier than he can factor n.

Should one try to find a d' which is equivalent to the secret d, and if such values d' were common, then a brute-force attack could crack the code. However, all such values d' differ by the least common multiple (lcm) of $(p-1)$ and $(q-1)$, and here is why. Since, by arithmetic modulo $\phi(n)$ we have $e \cdot d = 1 \pmod{\phi(n)}$, then indeed $e \cdot kd' = 1 \pmod{k \cdot \phi(n)}$ for an integer k, and setting $k = 1$, we obtain the original equation. Seeing that since $\phi(n) = (p-1) \cdot (q-1)$ and $k \cdot (p-1) \cdot (q-1) = \text{lcm}((p-1),(q-1))$ when $k = 1$, then indeed $\phi(n) = \text{lcm}((p-1),(q-1))$. Thus, finding such a d' is as hard as factoring n.

8.4 Other ways of computing d.

The authors of RSA argued that computing e-th roots modulo n without factoring n is practically impossible computationally. Thus, since the only other realistic option is to discover a way to break the code without factoring n, such a discovery would yield an efficient factoring algorithm. However, such a discovery, combined with ways of breaking the code by factoring n, would have to all be as hard as factoring n. The authors of RSA do not provide a proof for this conjecture. However, it seems that that were in fact correct, since there has been no known record of anyone breaking RSA in the 31 years since it has been published.

9 Avoiding "reblocking" for encryption of a signed message.

Reblocking means having to break a signed message up into smaller blocks, since the signature n may be greater than the encryption n (both are different, since they are from different keys from two or more distinct users). The authors of RSA, however, have provided a way to avoid reblocking a message: choose a threshold value h (say $h = 10^{202} - 33$) for the public-key cryptosystem, and assure that "every user maintains two public (e, n) pairs, one for enciphering and one for signature verification, where every signature n is less than h, and every enciphering n is greater than h". Thus, message blocking only depends on the transmitter's signature n.

10 Conclusions.

RSA is a strong encryption algorithm that has stood a partial test of time. RSA implements a public-key cryptosystem that allows secure communications and "digital signatures", and its security rests in part on the difficulty of factoring large numbers. The authors urged anyone to attempt to break their code, whether by factorization techniques or otherwise, and nobody to date seems to have succeeded. This has in effect certified RSA, and will continue to assure its security for as long as it stands the test of time against such break-ins.

At the time, the RSA encryption function seemed to be the only known candidate for a trap-door one-way permutation, but now, others certainly exist, such as those described in [3] and [7].

The average size of n must increase with time as more efficient factoring algorithms are made and as computers are getting faster. In 1978, the authors of RSA suggested 200-digit long values for n. "As of 2008, the largest (known) number factored by a general-purpose factoring algorithm was [200 digits (663 bits)] long" [10]. Currently, RSA keys are typically between 1024 and 2048 bits long, which experts predict may be breakable in the near future. So far, no one sees 4096-bit keys to be broken anytime soon. Today, an n no longer than 300 bits can be factored on a PC in several hours, thus keys are typically 4-7 times longer today.

RSA is slower than certain other symmetric cryptosystems. RSA is, in fact, commonly used to securely transmit the keys for another less secure, but faster algorithm. Several issues in fact exist that could potentially damage RSA's security, such as timing attacks and problems with key distribution. I will not go into detail about these issues here. They are described succinctly in [10]. In fact, these issues have solutions; the only downside is that any device implementing RSA would have to have much more hardware and software to counter certain types of attacks or attempts at eavesdropping.

A very major threat to RSA would be a solution to the Riemann hypothesis. Thus a solution has neither been proven to exist nor to not exist. Development on the Riemann hypothesis is currently relatively stagnant. However, if a solution were found, prime numbers would be too easy to find, and RSA would fall apart. Undoubtedly, much more sophisticated algorithms than RSA will continue to be developed as mathematicians discover more in the fields of number theory and cryptanalysis.