CHAPTER

12

RSA

The RSA system is probably the most widely used public-key cryptosystem in the world. It is certainly the best known. It provides both digital signatures and public-key encryption, which makes it a very versatile tool, and it is based on the difficulty of factoring large numbers, a problem that has fascinated many people over the last few millennia and has been studied extensively.

12.1 Introduction

RSA is similar to, yet very different from, Diffie-Hellman (see Chapter 11). Diffie-Hellman (DH for short) is based on a one-way function: assuming p and g are publicly known, you can compute ($g^x \bmod p$) from x, but you cannot compute x given $g^x \bmod p$. RSA is based on a trapdoor one-way function. Given the publicly known information n and e, it is easy to compute $m^e \bmod n$ from m, but not the other way around. However, if you know the factorization of n, then it is easy to do the inverse computation. The factorization of n is the trapdoor information. If you know it, you can invert the function; if you do not know it, you cannot invert the function. This trapdoor functionality allows RSA to be used both for encryption and digital signatures. RSA was invented by Ronald Rivest, Adi Shamir, and Leonard Adleman, and first published in 1978 [105].

Throughout this chapter we will use the values p, q, and n. The values p and q are different large primes, each on the order of a thousand bits long or more. The value n is defined by $n := pq$. (An ordinary product, that is, not modulo something.)

12.2 The Chinese Remainder Theorem

Instead of doing computations modulo a prime p as in the DH system, we will be doing computations modulo the composite number n. To explain what is going on, we will need a little more number theory about computations modulo n. A very useful tool is the *Chinese Remainder Theorem*, or CRT. It is named so because the basic version was first stated by the first-century Chinese mathematician Sun Tzu. (Most of the math you need for DH and RSA dates back thousands of years, so it can't be too difficult, right?)

The numbers modulo n are $0, 1, \ldots, n-1$. These numbers do not form a finite field as they would if n were a prime. Mathematicians still write $\mathbb{Z}_n$ for these numbers and call this a ring, but that is a term we won't need. For each x in $\mathbb{Z}_n$, we can compute the pair $(x \bmod p, x \bmod q)$. The Chinese Remainder Theorem states that you can compute the inverse function: if you know $(x \bmod p, x \bmod q)$, you can reconstruct x.

For ease of notation, we will define $(a, b) := (x \bmod p, x \bmod q)$.

First, we show that reconstruction is possible, then we'll give an algorithm to compute the original x. To be able to compute x given (a, b), we must be sure there is not a second number x' in $\mathbb{Z}_n$ such that $x' \bmod p = a$ and $x' \bmod q = b$. If this were the case, both x' and x would result in the same (a, b) pair, and no algorithm could figure out which of these two numbers was the original input.

Let $d := x - x'$, the difference between the numbers that lead to the same (a, b) pair. We have $(d \bmod p) = (x - x') \bmod p = (x \bmod p) - (x' \bmod p) = a - a = 0$; thus, d is a multiple of p. For much the same reason, d is a multiple of q. This implies that d is a multiple of $\mathrm{lcm}(p, q)$, because lcm is, after all, the *least* common multiple. As p and q are different primes, $\mathrm{lcm}(p, q) = pq = n$, and thus $x - x'$ is a multiple of n. But both x and x' are in the range $0, \ldots, n-1$, so $x - x'$ must be a multiple of n in the range $-n+1, \ldots, n-1$. The only valid solution is $x - x' = 0$, or $x = x'$. This proves that for any given pair (a, b), there is at most one solution for x. All we have to do now is find that solution.

12.2.1 Garner's Formula

The most practical way of computing the solution is *Garner's formula*.

$$x = (((a - b)(q^{-1} \bmod p)) \bmod p) \cdot q + b$$

Here the $(q^{-1} \bmod p)$ term is a constant that depends only on p and q. Remember that we can divide modulo p, and therefore we can compute $(1/q \bmod p)$, which is just a different way of writing $(q^{-1} \bmod p)$.

We don't need to understand Garner's formula. All we need to do is prove that the result x is correct.

First of all, we show that x is in the right range $0, \ldots, n-1$. Obviously $x \geq 0$. The part $t := (((a - b)(q^{-1} \bmod p)) \bmod p)$ must be in the range

$0, \ldots, p-1$ because it is a modulo p result. If $t \leq p-1$, then $tq \leq (p-1)q$ and $x = tq + b \leq (p-1)q + (q-1) = pq - 1 = n - 1$. This shows that x is in the range $0, \ldots, n-1$.

The result should also be correct modulo both p and q.

$$\begin{aligned} x \bmod q &= ((((a-b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod q \\ &= (K \cdot q + b) \bmod q && \text{for some } K \\ &= b \bmod q \\ &= b \end{aligned}$$

The whole thing in front of the multiplication by q is some integer K, but any multiple of q is irrelevant when computing modulo q. Modulo p is a bit more complicated:

$$\begin{aligned} x \bmod p &= ((((a-b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod p \\ &= (((a-b)q^{-1}) \cdot q + b) \bmod p \\ &= ((a-b)(q^{-1}q) + b) \bmod p \\ &= ((a-b) + b) \bmod p \\ &= a \bmod p \\ &= a \end{aligned}$$

In the first line, we simply expand $(x \bmod p)$. In the next line, we eliminate a couple of redundant $\bmod p$ operators. We then change the order of the multiplications, which does not change the result. (You might remember from school that multiplication is associative, so $(ab)c = a(bc)$.) The next step is to observe that $q^{-1}q = 1 \pmod p$, so we can remove this term altogether. The rest is trivial.

This derivation is a bit more complicated than the ones we have seen so far, especially as we use more of the algebraic properties. Don't worry if you can't follow it.

We can conclude that Garner's formula gives a result x that is in the right range and for which $(a, b) = (x \bmod p, x \bmod q)$. As we already know that there can only be one such solution, Garner's formula solves the CRT problem completely.

In real systems, you typically precompute the value $q^{-1} \bmod p$, so Garner's formula requires one subtraction modulo p, one multiplication modulo p, one full multiplication, and an addition.

12.2.2 Generalizations

The CRT also works when n is the product of multiple primes that are all different.[1] Garner's formula can be generalized to these situations, but we won't need that in this book.

[1]There are versions that work when n is divisible by the square or higher power of some primes, but those are even more complicated.

12.2.3 Uses

So what is the CRT good for? If you ever have to do a lot of computations modulo n, then using the CRT saves a lot of time. For a number $0 \leq x < n$, we call the pair $(x \bmod p, x \bmod q)$ the CRT representation of x. If we have x and y in CRT representation, then the CRT representation of $x + y$ is $((x + y) \bmod p, (x + y) \bmod q)$, which is easy to compute from the CRT representations of x and y. The first component $(x + y) \bmod p$ can be computed as $((x \bmod p) + (y \bmod p) \bmod p)$. This is just the sum (modulo p) of the first half of each of the CRT representations. The second component of the result can be computed in a similar manner.

You can compute a multiplication in much the same way. The CRT representation of xy is $(xy \bmod p, xy \bmod q)$, which is easy to compute from the CRT representations. The first part $(xy \bmod p)$ is computed by multiplying $(x \bmod p)$ and $(y \bmod p)$ and taking the result modulo p again. The second part is computed in the same manner modulo q.

Let k be the number of bits of n. Each of the primes p and q is about $k/2$ bits long. One addition modulo n would require one k-bit addition, perhaps followed by a k-bit subtraction if the result exceeded n. In the CRT representation, you have to do two modulo additions on numbers half the size. This is approximately the same amount of work.

For multiplication, the CRT saves a lot of time. Multiplying two k-bit numbers requires far more work than twice multiplying two $k/2$-bit numbers. For most implementations, CRT multiplication is twice as fast as a full multiplication. That is a significant savings.

For exponentiations, the CRT saves even more. Suppose you have to compute $x^s \bmod n$. The exponent s can be up to k bits long. This requires about $3k/2$ multiplications modulo n. Using the CRT representation, each multiplication is less work, but there is also a second savings. We want to compute $(x^s \bmod p, x^s \bmod q)$. When computing modulo p, we can reduce the exponent s modulo $(p - 1)$, and similarly modulo q. So we only have to compute $(x^{s \bmod (p-1)} \bmod p, x^{s \bmod (q-1)} \bmod q)$. Each of the exponents is only $k/2$ bits long and requires only $3k/4$ multiplications. Instead of $3k/2$ multiplications modulo n, we now do $2 \cdot 3k/4 = 3k/2$ multiplications modulo one of the primes. This saves a factor of 3–4 in computing time in a typical implementation.

The only costs of using the CRT are the additional software complexity and the necessary conversions. If you do more than a few multiplications in one computation, the overhead of these conversions is worthwhile. Most textbooks only talk about the CRT as an implementation technique for RSA. We find that the CRT representation makes it much easier to understand the RSA system. This is why we explained the CRT first. We'll soon use it to explain the behavior of the RSA system.

12.2.4 Conclusion

In conclusion: a number x modulo n can be represented as a pair (x mod p, x mod q) when $n = pq$. Conversion between the two representations is fairly straightforward. The CRT representation is useful if you have to do many multiplications modulo a composite number that you know the factorization of. (You cannot use it to speed up your computations if you don't know the factorization of n.)

12.3 Multiplication Modulo *n*

Before we delve into the details of RSA, we must look at how numbers modulo n behave under multiplication. This is somewhat different from the modulo p case we discussed before.

For any prime p, we know that for all $0 < x < p$ the equation $x^{p-1} = 1 \pmod{p}$ holds. This is not true modulo a composite number n. For RSA to work, we need to find an exponent t such that $x^t = 1 \bmod n$ for (almost) all x. Most textbooks just give the answer, which does not help you understand why the answer is true. It is actually relatively easy to find the correct answer by using the CRT.

We want a t such that, for almost all x, $x^t = 1 \pmod{n}$. This last equation implies that $x^t = 1 \pmod{p}$ and $x^t = 1 \pmod{q}$. As both p and q are prime, this only holds if $p - 1$ is a divisor of t, and $q - 1$ is a divisor of t. The smallest t that has this property is therefore $\text{lcm}(p-1, q-1) = (p-1)(q-1)/\gcd(p-1, q-1)$. For the rest of this chapter, we will use the convention that $t = \text{lcm}(p-1, q-1)$.

The letters p, q, and n are used by everybody, although some use capital letters. Most books don't use our t, but instead use the Euler totient function $\phi(n)$. For an n of the form $n = pq$, the Euler totient function can be computed as $\phi(n) = (p-1)(q-1)$, which is a multiple of our t. It is certainly true that $x^{\phi(n)} = 1$, and that using $\phi(n)$ instead of t gives correct answers, but using t is more precise.

We've skipped over one small issue in our discussion: $x^t \bmod p$ cannot be equal to 1 if $x \bmod p = 0$. So the equation $x^t \bmod n = 1$ cannot hold for *all* values x. There are not many numbers that suffer from this deficiency; there are q numbers with $x \bmod p = 0$ and p numbers with $x \bmod q = 0$, so the total number of values that have this problem is $p + q$. Or $p + q - 1$, to be more precise, because we counted the value 0 twice. This is an insignificant fraction of the total number of values $n = pq$. Even better, the actual property used by RSA is that $x^{t+1} = x \pmod{n}$, and this still holds even for these special numbers. Again, this is easy to see when using the CRT representation. If $x = 0$

(mod p), then $x^{t+1} = 0 = x \pmod{p}$, and similarly modulo q. The fundamental property $x^{t+1} = x \pmod{n}$ is preserved, and holds for all numbers in $\mathbb{Z}_n$.

12.4 RSA Defined

We can now define the RSA system. Start by randomly choosing two different large primes p and q, and compute $n = pq$. The primes p and q should be of (almost) equal size, and the modulus n ends up being twice as long as p and q are.

We use two different exponents, traditionally called e and d. The requirement for e and d is that $ed = 1 \pmod{t}$ where $t := \text{lcm}(p-1, q-1)$ as before. Recall that many texts write $ed = 1 \pmod{\phi(n)}$. We choose the public exponent e to be some small odd value and use the EXTENDEDGCD function from Section 10.3.5 to compute d as the inverse of e modulo t. This ensures that $ed = 1 \pmod{t}$.

To encrypt a message m, the sender computes the ciphertext $c := m^e \pmod{n}$. To decrypt a ciphertext c, the receiver computes $c^d \pmod{n}$. This is equal to $(m^e)^d = m^{ed} = m^{kt+1} = (m^t)^k \cdot m = (1)^k \cdot m = m \pmod{n}$, where k is some value that exists. So the receiver can decrypt the ciphertext m^e to get the plaintext m.

The pair (n, e) forms the public key. These are typically distributed to many different parties. The values (p, q, t, d) are the private key and are kept secret by the person who generated the RSA key.

For convenience, we often write $c^{1/e}$ mod n instead of c^d mod n. The exponents of a modulo n computation are all taken modulo t, because $x^t = 1 \pmod{n}$, so multiples of t in the exponent do not affect the result. And we computed d as the inverse of e modulo t, so writing d as $1/e$ is natural. The notation $c^{1/e}$ is often easier to follow, especially when multiple RSA keys are in use. That is why we also talk about taking the e'th root of a number. Just remember that computations of any roots modulo n require knowledge of the private key.

12.4.1 Digital Signatures with RSA

So far, we've only talked about encrypting messages with RSA. One of the great advantages of RSA is that it can be used for both encrypting messages and signing messages. These two operations use the same computations. To sign a message m, the owner of the private key computes $s := m^{1/e}$ mod n. The pair (m, s) is now a signed message. To verify the signature, anyone who knows the public key can verify that $s^e = m \pmod{n}$.

As with encryption, the security of the signature is based on the fact that the e'th root of m can only be computed by someone who knows the private key.

12.4.2 Public Exponents

The procedure described so far has one problem. If e has a common factor with $t = \text{lcm}(p-1, q-1)$, there is no solution for d. So we have to choose p, q, and e such that this situation does not occur. This is more of a nuisance than a problem, but it has to be dealt with.

Choosing a short public exponent makes RSA more efficient, as fewer computations are needed to raise a number to the power e. We therefore try to choose a small value for e. In this book, we will choose a fixed value for e, and choose p and q to satisfy the conditions above.

You have to be careful that the encryption functions and digital signature functions don't interact in undesirable ways. You don't want it to be possible for an attacker to decrypt a message c by convincing the owner of the private key to sign c. After all, signing the ''message'' c is the same operation as decrypting the ciphertext c. The encoding functions presented later in this book will prevent this. These encodings are remotely akin to block cipher modes of operation; you should not use the basic RSA operation directly. But even then, we still don't want to use the same RSA operation for both functions. We could use different RSA keys for encryption and authentication, but that would increase complexity and double the amount of key material.

Another approach, which we use here, is to use two different public exponents on the same n. We will use $e = 3$ for signatures and $e = 5$ for encryption. This decouples the systems because cube roots and fifth roots modulo n are independent of each other. Knowing one does not help the attacker to compute the other [46].

Choosing fixed values for e simplifies the system and also gives predictable performance. It does impose a restriction on the primes that you can use, as both $p-1$ and $q-1$ cannot be multiples of 3 or 5. It is easy to check for this when you generate the primes in the first place.

The rationale for using 3 and 5 is simple. These are the smallest suitable values.[2] We choose the smaller public exponent for signatures, because signatures are often verified multiple times, whereas any piece of data is only encrypted once. It therefore makes more sense to let the signature verification be the more efficient operation.

Other common values used for e are 17 and 65537. We prefer the smaller values, as they are more efficient. There are some minor potential problems with the small public exponents, but we will eliminate them with our encoding functions further on.

It would also be nice to have a small value for d, but we have to disappoint you here. Although it is possible to find a pair (e, d) with a small d, using a

[2]You could in principle use $e = 2$, but that would introduce a lot of extra complexities.

small d is insecure [127]. So don't play any games by choosing a convenient value for d.

12.4.3 The Private Key

It is extremely difficult for the attacker to find any of the values of the private key p, q, t, or d if she knows only the public key (n, e). As long as n is large enough, there is no known algorithm that will do this in an acceptable time. The best solution we know of is to factor n into p and q, and then compute t and d from that. This is why you often hear about factoring being so important for cryptography.

We've been talking about the private key consisting of the values p, q, t, and d. It turns out that knowledge of any one of these values is sufficient to compute all the other three. This is quite instructive to see.

We assume that the attacker knows the public key (n, e), as that is typically public information. If she knows p or q, things are easy. Given p she can compute $q = n/p$, and then she can compute t and d just as we did above.

What if the attacker knows (n, e, t)? First of all, $t = (p-1)(q-1)/\gcd(p-1, q-1)$, but as $(p-1)(q-1)$ is very close to n, it is easy to find $\gcd(p-1, q-1)$ as it is the closest integer to n/t. (The value $\gcd(p-1, q-1)$ is never very large because it is very unlikely that two random numbers share a large factor.) This allows the attacker to compute $(p-1)(q-1)$. She can also compute $n - (p-1)(q-1) + 1 = pq - (pq - p - q + 1) + 1 = p + q$. So now she has both $n = pq$ and $s := p + q$. She can now derive the following equations:

$$\begin{aligned} s &= p + q \\ s &= p + n/p \\ ps &= p^2 + n \\ 0 &= p^2 - ps + n \end{aligned}$$

The last is just a quadratic equation in p that she can solve with high-school math. Of course, once the attacker has p, she can compute all the other private key values as well.

Something similar happens if the attacker knows d. In all our systems, e will be very small. As $d < t$, the number $ed - 1$ is only a small factor times t. The attacker can just guess this factor, compute t, and then try to find p and q as above. If she fails, she just tries the other possibilities. (There are faster techniques, but this one is easy to understand.)

In short, knowing any one of the values p, q, t, or d lets the attacker compute all the others. It is therefore reasonable to assume that the owner of the private key has all four values. Implementations only need to store one of these values, but often store several of the values they need to perform the RSA decryption operation. This is implementation dependent, and is not relevant from a cryptographic point of view.

If Alice wants to decrypt or sign a message, she obviously must know d. As knowing d is equivalent to knowing p and q, we can safely assume that she knows the factors of n and can therefore use the CRT representation for her computations. This is nice, because raising a number to the power d is the most expensive operation in RSA, and using the CRT representation saves a factor of 3–4 work.

12.4.4 The Size of *n*

The modulus n should be the same size as the modulus p that you would use in the DH case. See Section 11.7 for the detailed discussion. To reiterate: the absolute minimum size for n is 2048 bits or so if you want to protect your data for 20 years. This minimum will slowly increase as computers get faster. If you can afford it in your application, let n be 4096 bits long, or as close to this size as you can manage. Furthermore, make sure that your software supports values of n up to 8192 bits long. You never know what the future will bring, and it could be a lifesaver if you can switch to using larger keys without replacing software or hardware.

The two primes p and q should be of equal size. For a k-bit modulus n, you can just generate two random $k/2$-bit primes and multiply them. You might end up with a $k-1$-bit modulus n, but that doesn't matter much.

12.4.5 Generating RSA Keys

To pull everything together, we present two routines that generate RSA keys with the desired properties. The first one is a modification of the GENERATE-LARGEPRIME function of Section 10.4. The only functional change is that we require that the prime satisfies $p \bmod 3 \neq 1$ and $p \bmod 5 \neq 1$ to ensure that we can use the public exponents 3 and 5. Of course, if you want to use a different fixed value for e, you have to modify this routine accordingly.

function GENERATERSAPRIME
input: k Size of the desired prime, in number of bits.
output: p A random prime in the interval $2^{k-1}, \ldots, 2^k - 1$ subject to $p \bmod 3 \neq 1 \wedge p \bmod 5 \neq 1$.

Check for a sensible range.
assert $1024 \leq k \leq 4096$
Compute maximum number of attempts.
$r \leftarrow 100k$
repeat
  $r \leftarrow r - 1$
  assert $r > 0$

Choose n as a random k-bit number.
$n \in_{\mathcal{R}} 2^{k-1}, \ldots, 2^k - 1$
Keep on trying until we find a prime.
until $n \bmod 3 \neq 1 \wedge n \bmod 5 \neq 1 \wedge \textsc{isPrime}(n)$
return n

Instead of specifying a full range in which the prime should fall, we only specify the size of the prime. This is a less-flexible definition, but somewhat simpler, and it is sufficient for RSA. The extra requirements are in the loop condition. A clever implementation will not even call $\textsc{isPrime}(n)$ if n is not suitable modulo 3 or 5, as isPrime can take a significant amount of computations.

So why do we still include the loop counter with the error condition? Surely, now that the range is large enough, we will always find a suitable prime? We'd hope so, but stranger things have happened. We are not worried about getting a range with no primes in it—we're worried about a broken PRNG that always returns the same composite result. This is, unfortunately, a common failure mode of random number generators, and this simple check makes generateRSAPrime safe from misbehaving PRNGs. Another possible failure mode is a broken isPrime function that always claims that the number is composite. Of course, we have more serious problems to worry about if any of these functions is misbehaving.

The next function generates all the key parameters.

function generateRSAKey
input: k Size of the modulus, in number of bits.
output: p, q Factors of the modulus.
n Modulus of about k bits.
d_3 Signing exponent.
d_5 Decryption exponent.

Check for a sensible range.
assert $2048 \leq k \leq 8192$
Generate the primes.
$p \leftarrow \textsc{generateRSAPrime}(\lfloor k/2 \rfloor)$
$q \leftarrow \textsc{generateRSAPrime}(\lfloor k/2 \rfloor)$
A little test just in case our PRNG *is bad . . . .*
assert $p \neq q$
Compute t as $\mathrm{lcm}(p-1, q-1)$.
$t \leftarrow (p-1)(q-1)/\mathrm{GCD}(p-1, q-1)$
Compute the secret exponents using the modular inversion feature of the extended GCD algorithm.
$g, (u, v) \leftarrow \textsc{extendedGCD}(3, t)$

Check that the GCD is correct, or we don't get an inverse at all.
assert $g = 1$
Reduce u modulo t, as u could be negative and d_3 shouldn't be.
$d_3 \leftarrow u \bmod t$
And now for d_5.
$g, (u, v) \leftarrow \text{EXTENDEDGCD}(5, t)$
assert $g = 1$
$d_5 \leftarrow u \bmod t$
return p, q, pq, d_3, d_5

Note that we've used the fixed choices for the public exponents, and that we generate a key that can be used both for signing ($e = 3$) and for encryption ($e = 5$).

12.5 Pitfalls Using RSA

Using RSA as presented so far is very dangerous. The problem is the mathematical structure. For example, if Alice digitally signs two messages m_1 and m_2, then Bob can compute Alice's signature on $m_3 := m_1 m_2 \bmod n$. After all, Alice has computed $m_1^{1/e}$ and $m_2^{1/e}$ and Bob can multiply the two results to get $(m_1 m_2)^{1/e}$.

Another problem arises if Bob encrypts a very small message m with Alice's public key. If $e = 5$ and $m < \sqrt[5]{n}$, then $m^e = m^5 < n$, so no modular reduction ever takes place. The attacker Eve can recover m by simply taking the fifth root of m^5, which is easy to do because there are no modulo reductions involved. A typical situation in which this could go wrong is if Bob tries to send an AES key to Alice. If she just takes the 256-bit value as an integer, then the encrypted key is less than $2^{256 \cdot 5} = 2^{1280}$, which is much smaller than our n. There is never a modulo reduction, and Eve can compute the key by simply computing the fifth root of the encrypted key value.

One of the reasons we have explained the theory behind RSA in such detail is to teach you some of the mathematical structure that we encounter. This very same structure invites many types of attack. We've mentioned some simple ones in the previous paragraph. There are far more advanced attacks, based on techniques for solving polynomial equations modulo n. All of them come down to a single thing: it is very bad to have any kind of structure in the numbers that RSA operates on.

The solution is to use a function that destroys any available structure. Sometimes this is called a padding function, but this is a misnomer. The word padding is normally used for adding additional bytes to get a result of the right

length. People have used various forms of padding for RSA encryption and signatures, and quite a few times this has resulted in attacks on their designs. What you need is a function that removes as much structure as possible. We'll call this the encoding function.

There are standards for this, most notably PKCS #1 v2.1 [110]. As usual, this is not a single standard. There are two RSA encryption schemes and two RSA signature schemes, each of which can take a variety of hash functions. This is not necessarily bad, but from a pedagogical perspective we don't like the extra complexity. We'll therefore present some simpler methods, even though they might not have all the features of some of the PKCS methods. And, as we mentioned before in the case of AES, there are many advantages to using a standardized algorithm in practice. For example, for encryption you might use RSA-OAEP [9], and for signatures you might use RSA-PSS [8].

The PKCS #1 v2.1 standard also demonstrates a common problem in technical documentation: it mixes specification with implementation. The RSA decryption function is specified twice; once using the equation $m = c^d \bmod n$ and once using the CRT equations. These two computations have the same result: one is merely an optimized implementation of the other. Such implementation descriptions should not be part of the standard, as they do not produce different behavior. They should be discussed separately. We don't want to criticize this PKCS standard in particular; it is a very widespread problem that you find throughout the computer industry.

12.6 Encryption

Encrypting a message is the canonical application of RSA, yet it is almost never used in practice. The reason is simple: the size of the message that can be encrypted using RSA is limited by the size of n. In real systems, you cannot even use all the bits, because the encoding function has an overhead. This limited message size is too impractical for most applications, and because the RSA operation is quite expensive in computational terms, you don't want to split a message into smaller blocks and encrypt each of them with a separate RSA operation.

The solution used almost everywhere is to choose a random secret key K, and encrypt K with the RSA keys. The actual message m is then encrypted with key K using a block cipher or stream cipher. So instead of sending something like $E_{\text{RSA}}(m)$, you send $E_{\text{RSA}}(K), E_K(m)$. The size of the message is no longer limited, and only a single RSA operation is required, even for large messages. You have to transmit a small amount of extra data, but this is usually a minor price to pay for the advantages you get.

We will use an even simpler method of encryption. Instead of choosing a K and encrypting K, we choose a random $r \in \mathbb{Z}_n$ and define the bulk encryption key as $K := h(r)$ for some hash function h. Encrypting r is done by simply raising

it to the fifth power modulo n. (Remember, we use $e = 5$ for encryption.) This solution is simple and secure. As r is chosen randomly, there is no structure in r that can be used to attack the RSA portion of the encryption. The hash function in turn ensures that no structure between different r's propagates to structure in the K's, except for the obvious requirement that equal inputs must yield equal outputs.

For simplicity of implementation, we choose our r's in the range $0, \ldots, 2^k - 1$, where k is the largest number such that $2^k < n$. It is easier to generate a random k-bit number than to generate a random number in $\mathbb{Z}_n$, and this small deviation from the uniform distribution is harmless in this situation.

Here is a more formal definition:

function ENCRYPTRANDOMKEYWITHRSA
input: (n, e) RSA public key, in our case $e = 5$.
output: K Symmetric key that was encrypted.
c RSA ciphertext.

Compute k.
$k \leftarrow \lfloor \log_2 n \rfloor$
Choose a random r such that $0 \leq r < 2^k - 1$.
$r \in_{\mathcal{R}} \{ 0, \ldots, 2^k - 1 \}$
$K \leftarrow \mathrm{SHA}_d\text{-}256(r)$
$c \leftarrow r^e \bmod n$
return (K, c)

The receiver computes $K = h(c^{1/e} \bmod n)$ and gets the same key K.

function DECRYPTRANDOMKEYWITHRSA
input: (n, d) RSA private key with $e = 5$.
c Ciphertext.
output: K Symmetric key that was encrypted.

assert $0 \leq c < n$
This is trivial.
$K \leftarrow \mathrm{SHA}_d\text{-}256(c^{1/e} \bmod n)$
return K

We previously dealt extensively with how to compute $c^{1/e}$ given the private key, so we won't discuss that here again. Just don't forget to use the CRT for a factor of 3–4 speed-up.

Here is a good way to look at the security. Let's assume that Bob encrypts a key K for Alice, and Eve wants to know more about this key. Bob's message depends only on some random data and on Alice's public key. So at worst this message could leak data to Eve about K, but it cannot leak any data about any other secret, such as Alice's private key. The key K is computed using a hash function, and we can pretend that the hash function is a random mapping. (If we cannot treat the hash function as a random mapping, it doesn't satisfy

our security requirement for hash functions.) The only way to get information about the output of a hash function is to know most of the input. That means having information about r. But if RSA is secure—and we have to assume that since we have chosen to use it—then it is impossible to get any significant amount of information about a randomly chosen r given just ($r^e \bmod n$). This leaves the attacker with a lot of uncertainty about r, and consequently, no knowledge about K.

Suppose the key K is later revealed to Eve, maybe due to a failure of another component of the system. Does this reveal anything about Alice's private key? No. K is the output of a hash function, and it is impossible for Eve to derive any information about the inputs to the hash function. So even if Eve chose c in some special way, the K she acquires does not reveal anything about r. Alice's private key was only used to compute r, so Eve cannot learn anything about Alice's private key either.

This is one of the advantages of having a hash function in the DECRYPTRANDOMKEYWITHRSA function. Suppose it just returned $c^{1/e} \bmod n$. This routine could then be used to play all kinds of games. Suppose some other part of the system had a weakness and Eve learned the least significant bit of the output. Eve could then send specially chosen values $c_1, c_2, c_3, \ldots$ to Alice and get the least significant bits of $c_1^{1/e}, c_2^{1/e}, c_3^{1/e}, \ldots$. These answers have all kinds of algebraic properties, and it is quite conceivable that Eve could learn something useful from a situation like this. The hash function h in DECRYPTRANDOMKEYWITHRSA destroys all mathematical structure. Learning one bit from the output K gives Eve almost no information about $c^{1/e}$. Even the full result K divulges very little useful information; the hash function is not invertible. Adding the hash function here makes the RSA routines more secure against failures in the rest of the system.

This is also the reason why DECRYPTRANDOMKEYWITHRSA does *not* check that the r we compute from c falls in the range $0, \ldots, 2^k - 1$. If we checked this condition, we would have to handle the error that could result. As error handling always leads to different behavior, it is quite probable that Eve could detect whether this error occurred. This would provide Eve with a function that reveals information: Eve could choose any value c and learn whether $c^{1/e} \bmod n < 2^k$. Eve cannot compute this property without Alice's help, and we don't want to help Eve if we can avoid it. By not checking the condition, we at most generate a nonsense output, and that is something that can happen in any case, as c might have been corrupted without resulting in an invalid r value.[3]

An aside: there is a big difference between revealing a random pair $(c, c^{1/e})$, and computing $c^{1/e}$ for a c chosen by someone else. Anybody can produce pairs

[3]Placing more restrictions on r does not stop the problem of nonsensical outputs. Eve can always use Alice's public key and a modified ENCRYPTRANDOMKEYWITHRSA function to send Alice encryptions of nonsensical keys.

of the form $(c, c^{1/e})$. All you do is choose a random r, compute the pair (r^e, r), and then set $c := r^e$. There is nothing secret about pairs like that. But if Alice is kind enough to compute $c^{1/e}$ for a c she receives from Eve, Eve can choose c values with some special properties—something she can't do for the $(c, c^{1/e})$ pairs she generates herself. Don't provide this extra service for your attacker.

12.7 Signatures

For signatures, we have to do a bit more work. The problem is that the message m we want to sign can have a lot of structure to it, and we do not want any structure in the number we compute the RSA root on. We have to destroy the structure.

The first step is to hash the message. So instead of a variable-length message m, we deal with a fixed-size value $h(m)$ where h is a hash function. If we use SHA_d-256, we get a 256-bit result. But n is much bigger than that, so we cannot use $h(m)$ directly.

The simple solution is to use a pseudorandom mapping to expand $h(m)$ to a random number s in the range $0, \ldots, n-1$. The signature on m is then computed as $s^{1/e} \pmod{n}$. Mapping $h(m)$ to a modulo n value is a bit of work (see the discussion in Section 9.7). In this particular situation, we can safely simplify our problem by mapping $h(m)$ to a random element in the range $0, \ldots, 2^k - 1$, where k is the largest number such that $2^k < n$. Numbers in the range $0, \ldots, 2^k - 1$ are easy to generate because we only need to generate k random bits. In this particular situation, this is a safe solution, but don't use it just anywhere. There are many situations in cryptography where this will break your entire system.

We will use the generator from our Fortuna PRNG from Chapter 9. Many systems use the hash function h to build a special little random number generator for this purpose, but we've already defined a good one. Besides, you need the PRNG to choose the primes to generate the RSA keys, so you have the PRNG in the software already.

This results in three functions—one to map the message to s, one to sign the message, and one to verify the signature.

function MsgToRSANumber
input: n RSA public key, modulus.
m Message to be converted to a value modulo n.
output: s A number modulo n.

Create a new PRNG *generator.*
$\mathcal{G} \leftarrow$ InitializeGenerator()
Seed it with the hash of the message.

ReSeed($\mathcal{G}$, SHA$_d$-256(m))
Compute k.
$k \leftarrow \lfloor \log_2 n \rfloor$
$x \leftarrow$ PseudoRandomData($\mathcal{G}$, $\lceil k/8 \rceil$)
As usual, we treat the byte-string x as an integer using the LSByte first convention. The modulo reduction can be implemented with a simple AND *on the last byte of x.*
$s \leftarrow x \bmod 2^k$
return s

function SignWithRSA
input: (n, d) RSA private key with $e = 3$.
m Message to be signed.
output: σ Signature on m.
$s \leftarrow$ MsgToRSANumber(n, m)
$\sigma \leftarrow s^{1/e} \bmod n$
return σ

The letter σ, or sigma, is often used for signatures because it is the Greek equivalent of our letter s. By now you should know how to compute $s^{1/e} \bmod n$, given the private key.

function VerifyRSASignature
input: (n, e) RSA public key with $e = 3$.
m Message that is supposed to be signed.
σ Signature on the message.
$s \leftarrow$ MsgToRSANumber(n, m)
assert $s = \sigma^e \bmod n$

Of course, in a real application there will be some action to take if the signature verification fails. We've just written an assertion here to indicate that normal operations should not proceed. A signature failure should be taken like any other failure in a cryptographic protocol: as a clear signal that you are under active attack. Don't send any replies unless you absolutely have to, and destroy all the material you are working on. The more information you send out, the more information you give the attacker.

The security argument for our RSA signatures is similar to that of the RSA encryptions. If you ask Alice to sign a bunch of messages $m_1, m_2, \ldots, m_i$, then you are getting pairs of the form $(s, s^{1/e})$, but the s values are effectively random. As long as the hash function is secure, you can only affect $h(m)$ by trial and error. The random generator is again a random mapping. Anyone can create pairs of the form $(s, s^{1/e})$ for random s values, so this provides no new information that helps the attacker forge a signature. However, for any

particular message m, only someone who knows the private key can compute the corresponding $(s, s^{1/e})$ pair, because s must be computed from $h(m)$, then $s^{1/e}$ must be computed from s. This requires the private key. Therefore, anyone who verifies the signature knows that Alice must have signed it.

This brings us to the end of our treatment of RSA, and to the end of the math-heavy part of this book. We will be using DH and RSA for our key negotiation protocol and the PKI, but that only uses the math we have already explained. No new mathematics will be introduced.

12.8 Exercises

Exercise 12.1 Let $p = 89$, $q = 107$, $n = pq$, $a = 3$, and $b = 5$. Find x in $\mathbb{Z}_n$ such that $a = x \pmod{p}$ and $b = x \pmod{q}$.

Exercise 12.2 Let $p = 89$, $q = 107$, $n = pq$, $x = 1796$, and $y = 8931$. Compute $x + y \pmod{n}$ directly. Compute $x + y \pmod{n}$ using CRT representations.

Exercise 12.3 Let $p = 89$, $q = 107$, $n = pq$, $x = 1796$, and $y = 8931$. Compute $xy \pmod{n}$ directly. Compute $xy \pmod{n}$ using CRT representations.

Exercise 12.4 Let $p = 83$, $q = 101$, $n = pq$, and $e = 3$. Is (n, e) a valid RSA public key? If so, compute the corresponding private RSA key d. If not, why not?

Exercise 12.5 Let $p = 79$, $q = 89$, $n = pq$, and $e = 3$. Is (n, e) a valid RSA public key? If so, compute the corresponding private RSA key d. If not, why not?

Exercise 12.6 To speed up decryption, Alice has chosen to set her private key $d = 3$ and computes e as the inverse of d modulo t. Is this a good design decision?

Exercise 12.7 Does a 256-bit RSA key (a key with a 256-bit modulus) provide strength similar to that of a 256-bit AES key?

Exercise 12.8 Let $p = 71$, $q = 89$, $n = pq$, and $e = 3$. First find d. Then compute the signature on $m_1 = 5416$, $m_2 = 2397$, and $m_3 = m_1 m_2 \pmod{n}$ using the basic RSA operation. Show that the third signature is equivalent to the product of the first two signatures.