to-do

Disclaimer: This chapter is currently in a transitional state. It covers the mechanics and math of the RSA function (exponentiation mod \(N\)), but not actual applications to digital signatures. Over the next few months I plan to make this a proper chapter about signatures.

RSA was among the first public-key cryptography developed. It was first described in 1978, and is named after its creators, Ron Rivest, Adi Shamir, and Len Adleman.\(^1\) RSA can be used as a building block for public-key encryption and digital signatures. In this chapter we discuss only the application of RSA for digital signatures.

### 13.1 “Dividing” Mod \(n\)

Please review the material from Section 0.2, to make sure your understanding of basic modular arithmetic is fresh. You should be comfortable with the definitions of \(\mathbb{Z}_n\), congruence (\(\equiv\)), the modulus operator (\(\%\)), and how to do addition, multiplication, and subtraction mod \(n\).

Note that we haven’t mentioned division mod \(n\). Does it even make sense to talk about division mod \(n\)?

**Example**

*Consider the following facts which hold mod 15:*

\[
\begin{align*}
2 \cdot 8 &\equiv_{15} 1 \\
4 \cdot 8 &\equiv_{15} 2 \\
6 \cdot 8 &\equiv_{15} 3 \\
8 \cdot 8 &\equiv_{15} 4 \\
10 \cdot 8 &\equiv_{15} 5 \\
12 \cdot 8 &\equiv_{15} 6 \\
14 \cdot 8 &\equiv_{15} 7 \\
\end{align*}
\]

*Now imagine replacing “\(\cdot 8\)” with “\(\div 2\)” in each of these examples:*

\[
\begin{align*}
2 \div 2 &\equiv_{15} 1 \\
4 \div 2 &\equiv_{15} 2 \\
6 \div 2 &\equiv_{15} 3 \\
8 \div 2 &\equiv_{15} 4 \\
10 \div 2 &\equiv_{15} 5 \\
12 \div 2 &\equiv_{15} 6 \\
14 \div 2 &\equiv_{15} 7 \\
\end{align*}
\]

*Everything still makes sense! Somehow, multiplying by 8 mod 15 seems to be the same thing as “dividing by 2” mod 15.*

\(^{1}\text{Clifford Cocks developed an equivalent scheme in 1973, but it was classified since he was working for British intelligence.}\)
The previous examples all used \( x \cdot 8 \) (\((x \div 2)\)) where \( x \) was an even number. What happens when \( x \) is an odd number?

\[
3 \cdot 8 \equiv_{15} 9 \iff "3 \div 2 \equiv_{15} 9" ~ ?
\]

This might seem non-sensical, but if we make the substitutions \( 3 \equiv_{15} -12 \) and \( 9 \equiv_{15} -6 \), then we do indeed get something that makes sense:

\[
-12 \cdot 8 \equiv_{15} -6 \iff -12 \div 2 \equiv_{15} -6
\]

This example shows that there is surely some interesting relationship between the numbers 2, 8, and 15. It seems reasonable to interpret “multiplication by 8” as “division by 2” when working mod 15.

Is there a way we can do something similar for “division by 3” mod 15? Can we find some \( y \) where “multiplication by \( y \) mod 15” has the same behavior as “division by 3 mod 15?” In particular, we would seek a value \( y \) that satisfies \( 3 \cdot y \equiv_{15} 1 \), but you can check for yourself that no such value of \( y \) exists.

Why can we “divide by 2” mod 15 but we apparently cannot “divide by 3” mod 15? We will explore this question in the remainder of this section.

**Multiplicative Inverses**

Instead of talking about “division mod 15”, we usually prefer different terminology. We say that 8 and 2 are **multiplicative inverses** mod 15, since \( 8 \cdot 2 \equiv_{15} 1 \). And instead of using the “\( \div \)” operator, we write \( 2^{-1} \equiv_{15} 8 \), so we can write “division by 2” as “multiplication by \( 2^{-1} \).”

**Definition 13.1** The **multiplicative inverse** of \( x \) mod \( n \) is any number \( x^{-1} \) that satisfies \( x \cdot x^{-1} \equiv_{n} 1 \).

**Example** Continuing to work mod 15, we have:

\[ \text{\checkmark} \quad 4^{-1} \equiv_{15} 4 \text{ since } 4 \cdot 4 = 16 \equiv_{15} 1. \text{ Hence 4 is its own multiplicative inverse! You can also understand this as:} \]

\[
4^{-1} = (2^2)^{-1} = (2^{-1})^2 \equiv_{15} 8^2 = 64 \equiv_{15} 4
\]

\[ \text{\checkmark} \quad 7^{-1} \equiv_{15} 13 \text{ since } 7 \cdot 13 = 91 \equiv_{15} 1. \]

Which numbers have a multiplicative inverse mod \( n \)? The answer is quite simple:

**Definition 13.2** The **multiplicative group** \( \mathbb{Z}_n^* \) **modulo** \( n \) is defined as:

\[
\mathbb{Z}_n^* = \{ x \in \mathbb{Z}_n \mid x \text{ has a multiplicative inverse mod } n \}
\]

\(^2\)“Group” is a term from abstract algebra. It refers to a collection of items that have an operator, where applying the operator to any two items from the group results in another item also in the group. A group must also have some sense of inverses under that operation. In this case, the operation is multiplication, and our definition restricts items to those that have inverses.
For example, we have seen that $\mathbb{Z}_n^*$ contains the numbers 2, 4, and 7 (and perhaps others), but it doesn’t contain the number 3.

**Theorem 13.3**  
$x$ has a multiplicative inverse mod $n$ if and only if $\gcd(x, n) = 1$. That is, $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid \gcd(x, n) = 1\}$.

We prove the theorem using another fact from abstract algebra which is often useful:

**Theorem 13.4**  
(Bezout’s Theorem)  
For all integers $x$ and $y$, there exist integers $a$ and $b$ such that $ax + by = \gcd(x, y)$. In fact, $\gcd(x, y)$ is the smallest positive integer that can be written as an integral linear combination of $x$ and $y$.

We won’t prove Bezout’s theorem, but we will show how it is used to prove Theorem 13.3:

**Proof (of Theorem 13.3)**  
We must show both directions of the if-and-only-if.

$(\Leftarrow)$ Suppose $\gcd(x, n) = 1$. We will show that $x$ has a multiplicative inverse mod $n$. From Bezout’s theorem, there exist integers $a, b$ satisfying $ax + bn = 1$. By reducing both sides of this equation modulo $n$, we have

$$1 = ax + bn \equiv_n ax + b \cdot 0 = ax.$$  

Thus the integer $a$ that falls out of Bezout’s theorem is the multiplicative inverse of $x$ modulo $n$.

$(\Rightarrow)$ Suppose $x$ has a multiplicative inverse mod $n$, so $xx^{-1} \equiv_n 1$. We need to show that $\gcd(x, n) = 1$. From the definition of $\equiv_n$, we know that $n$ divides $xx^{-1} - 1$, so we can write $xx^{-1} - 1 = kn$ (as an expression over the integers) for some integer $k$. Rearranging, we have that $xx^{-1} - kn = 1$. That is to say, we have a way to write 1 as an integral linear combination of $x$ and $n$. From Bezout’s theorem, this must mean that $\gcd(x, n) = 1$. □

**Example**  
$\mathbb{Z}_{15} = \{0, 1, \ldots, 14\}$, and to obtain $\mathbb{Z}_{15}^*$ we exclude any of the numbers that share a common factor with 15. In other words, we exclude the multiples of 3 and multiples of 5. The remaining numbers are $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

Since 11 is a prime, 0 is the only number in $\mathbb{Z}_{11}$ that shares a common factor with 11. All the rest satisfy $\gcd(x, 11) = 1$. Hence, $\mathbb{Z}_{11}^* = \{1, 2, \ldots, 10\}$.

The relationship between multiplicative inverses and GCD goes even farther than Theorem 13.3. Recall that we can compute $\gcd(x, n)$ efficiently using Euclid’s algorithm. There is a relatively simple modification to Euclid’s algorithm that also computes the corresponding Bezout coefficients with little extra work. In other words, given $x$ and $n$, it is possible to efficiently compute integers $a, b,$ and $d$ such that

$$ax + bn = d = \gcd(x, n)$$

In the case where $\gcd(x, n) = d = 1$, the integer $a$ is a multiplicative inverse of $x$ mod $n$. The “extended Euclidean algorithm” for GCD is given below:
CHAPTER 13. RSA & DIGITAL SIGNATURES

Example

Below is a table showing the computation of \( \text{EXTGCD}(35, 144) \). Note that the columns \( x, y \) are computed from the top down (as recursive calls to \( \text{EXTGCD} \) are made), while the columns \( d, a, b \) are computed from bottom up (as recursive calls return). Also note that in each row, we indeed have \( d = ax + by \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( \lfloor \frac{x}{y} \rfloor )</th>
<th>( x \mod y )</th>
<th>( d )</th>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>144</td>
<td>0</td>
<td>35</td>
<td>1</td>
<td>-37</td>
<td>9</td>
</tr>
<tr>
<td>144</td>
<td>35</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>-37</td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>-1</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The final result demonstrates that \( 35^{-1} \equiv_{144} -37 \equiv_{144} 107 \).

The Totient Function

Euler’s totient function is defined as \( \phi(n) \equiv |\mathbb{Z}_n^*| \), in other words, the number of elements of \( \mathbb{Z}_n \) which are relatively prime to \( n \).

As an example, if \( n \) is a prime, then \( \mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\} \) because every integer in \( \mathbb{Z}_n \) apart from zero is relatively prime to \( n \). Therefore, \( \phi(n) = n - 1 \) in this case.

We will frequently work \( n \) that is the product of two distinct primes \( n = pq \). In that case, \( \phi(n) = (p - 1)(q - 1) \). To see why, let’s count how many elements in \( \mathbb{Z}_{pq}^* \) share a common divisor with \( pq \) (i.e., are not in \( \mathbb{Z}_{pq}^* \)).

- The multiples of \( p \) share a common divisor with \( pq \). These include 0, \( p, 2p, 3p, \ldots, (q - 1)p \). There are \( q \) elements in this list.

- The multiples of \( q \) share a common divisor with \( pq \). These include 0, \( q, 2q, 3q, \ldots, (p - 1)q \). There are \( p \) elements in this list.

We have clearly double-counted element 0 in these lists. But no other element is double counted. Any item that occurs in both lists would be a common multiple of both \( p \) and \( q \), but since \( p \) and \( q \) are relatively prime, their least common multiple is \( pq \), which is larger than any item in these lists.

We count \( p + q - 1 \) elements of \( \mathbb{Z}_{pq}^* \) which share a common divisor with \( pq \). The rest belong to \( \mathbb{Z}_{pq}^* \), and there are \( pq - (p + q - 1) = (p - 1)(q - 1) \) of them. Hence \( \phi(pq) = (p - 1)(q - 1) \).
General formulas for $\phi(n)$ exist, but they typically rely on knowing the prime factorization of $n$. We will see more connections between the difficulty of computing $\phi(n)$ and the difficulty of factoring $n$ later in this part of the course.

The reason we consider $\phi(n)$ at all is this fundamental theorem from abstract algebra:

**Theorem 13.5 (Euler’s Theorem)**

If $x \in \mathbb{Z}_n^*$ then $x^{\phi(n)} \equiv n \pmod{1}$.

Using the formula for $\phi(n)$, we can see that $\phi(15) = \phi(3 \cdot 5) = (3 - 1)(5 - 1) = 8$. Euler’s theorem says that raising any element of $\mathbb{Z}_{15}^*$ to the 8 power results in 1:

- $2^8 = 256 = 255 + 1 \equiv 1_{15}$
- $4^8 = 65536 = 65535 + 1 \equiv 1_{15}$

Similarly, $1^8, 7^8, 8^8, 11^8, 13^8, 14^8$ all are congruent to 1 mod 15.

**13.2 The RSA Function**

The RSA function is defined as follows:

- Let $p$ and $q$ be distinct primes (later we will say more about how they are chosen), and let $N = pq$. $N$ is called the **RSA modulus**.
- Let $e$ and $d$ be integers such that $ed \equiv \phi(N) \pmod{1}$. That is, $e$ and $d$ are multiplicative inverses mod $\phi(N)$ — not mod $N$! $e$ is called the **encryption exponent**, and $d$ is called the **decryption exponent**. These names are historical, but not entirely precise since RSA by itself does not achieve CPA security.
- The RSA function is: $m \mapsto m^e \pmod{N}$, where $m \in \mathbb{Z}_N$.
- The inverse RSA function is: $c \mapsto c^d \pmod{N}$, where $c \in \mathbb{Z}_N$.

Essentially, the RSA function (and its inverse) is a simple modular exponentiation. The most confusing thing to remember about RSA is that $e$ and $d$ “live” in $\mathbb{Z}_{\phi(N)}^*$, while $m$ and $c$ “live” in $\mathbb{Z}_N$.

Let’s make sure the function we called the “inverse RSA function” is actually an inverse of the RSA function. The RSA function raises its input to the $e$ power, and the inverse RSA function raises its input to the $d$ power. So it suffices to show that raising to the $ed$ power has no effect modulo $N$.

Since $ed \equiv \phi(N) \pmod{1}$, we can write $ed = t\phi(N) + 1$ for some integer $t$. Then:

$$(m^e)^d = m^{ed} = m^{t\phi(N)+1} = (m^{\phi(N)})^t m \equiv_N 1^t m = m$$

Note that we have used the fact that $m^{\phi(N)} \equiv_N 1$ from Euler’s theorem.
How [Not] to Exponentiate Huge Numbers

When you see an expression like \(m^e \% N\), you might be tempted to implement it with the following algorithm:

```plaintext
NaiveExponentiate(m, e, N):
result = 1
for i = 1 to e: // compute m^e
    result = result \times m
return result \% N
```

While this algorithm would indeed give the correct answer, it is a really bad way of doing it. To see why, you must keep in mind that the RSA function involves numbers that are thousands of bits long. Suppose each of \(m\), \(e\), and \(N\) are around a thousand bits each (so the magnitude of these numbers is close to \(2^{1000}\)). There are two seriously problematic things about the NaiveExponentiate algorithm:

1. If \(e\) is approximately as large as \(2^{1000}\), then the algorithm will spend approximately \(2^{1000}\) iterations in the for-loop!

2. This algorithm computes \(m^e\) as an integer first, and then reduces that integer mod \(N\). Even if there was a better way to compute the integer \(m^e\) (there is), if \(m\) is 1000 bits long, then \(m^e\) will be roughly \(e \cdot 1000\) bits long! It would take about \(2^{1000} \cdot 1000\) bits just to write it down.

As you can see, there is neither enough time nor storage capacity in the universe to use this algorithm. So how can we actually compute values like \(m^e \% N\) on huge numbers?

1. Suppose you were given an integer \(m\) and were asked to compute \(m^{17}\). You can compute it as:

\[
m^{17} = \underbrace{m \cdot m \cdot m \cdots m}_{16 \text{ multiplications}}.
\]

But a more clever way is to observe that:

\[
m^{17} = m^{16} \cdot m = ((m^2)^2)^2 \cdot m.
\]

This expression can be evaluated with only 5 multiplications (squaring is just multiplying a number by itself).

More generally, you can compute an expression like \(m^e\) by following the recurrence below. The method is called **exponentiation by repeated squaring**, for obvious reasons:

```plaintext
BetterExp(m, e):
if e = 0: return 1
if e even:
    return BetterExp(m, \frac{e}{2})^2
if e odd:
    return BetterExp(m, \frac{e-1}{2})^2 \cdot m
```

```plaintext
m^e = \begin{cases} 
1 \quad & \text{if } e = 0 \\
(m^{\frac{e}{2}})^2 \quad & \text{if } e \text{ even} \\
(m^{\frac{e-1}{2}})^2 \cdot m \quad & \text{if } e \text{ odd} 
\end{cases}
```
BETTEREXP divides the $e$ argument by two (more or less) each time it recurses, until reaching the base case. Hence, the number of recursive calls is $O(\log e)$. In each recursive call there are only a constant number of multiplications (including squarings). So overall this algorithm requires only $O(\log e)$ multiplications (compared to $e - 1$ multiplications by just multiplying $m$ by itself $e$ times). In the case where $e \sim 2^{1000}$, this means a few thousand multiplications.

2. We only care about $m^e \% N$, not the intermediate integer value $m^e$. One of the most fundamental features of modular arithmetic is that you can reduce any intermediate values mod $N$ if you only care about the final expression mod $N$.

Revisiting our previous example:

$$m^{17} \% N = m^{16} \cdot m \% N = (((m^2 \% N)^2 \% N)^2 \% N)^2 \cdot m \% N.$$  

More generally, we can reduce all intermediate values mod $N$:

```plaintext
ModExp(m, e, N): // compute m^e \% N
if e = 0: return 1
if e even:
    return ModExp(m, e/2, N)^2 \% N
if e odd:
    return ModExp(m, (e-1)/2, N)^2 \cdot m \% N
```

This algorithm avoids the problem of computing the astronomically huge integer $m^e$. It never needs to store any value (much) larger than $N$.

**Warning:** Even this ModExp algorithm isn’t an ideal way to implement exponentiation for cryptographic purposes. Exercise 13.8 explores some interesting properties of this exponentiation algorithm.

**Security Properties**

In these notes we will not formally define a desired security property for RSA. Roughly speaking, the idea is that even when $N$ and $e$ can be made public, it should be hard to compute the operation $c \mapsto c^d \% N$. In other words, the RSA function $m \mapsto m^e \% N$ is:

- easy to compute given $N$ and $e$
- hard to invert given $N$ and $e$ but not $d$
- easy to invert given $d$

**to-do** more details
13.3 Digital Signatures

to-do expanded discussion/motivation for digital signatures goes here, as well as specifics of RSA signatures

L_{\Sigma}^{\Sigma}\text{-sig-real}

(\nu k, sk) \leftarrow \Sigma.\text{KeyGen}

\text{GETVK()}: \quad \text{return } \nu k

\text{GETSIG}(m): \quad \text{return } \Sigma.\text{Sign}(sk, m)

\text{VER}(m, \sigma): \quad \text{return } \Sigma.\text{Ver}(\nu k, m, \sigma)

L_{\Sigma}^{\Sigma}\text{-sig-fake}

(\nu k, sk) \leftarrow \Sigma.\text{KeyGen}

S := \emptyset

\text{GETVK()}: \quad \text{return } \nu k

\text{GETSIG}(m): \quad \sigma := \Sigma.\text{Sign}(sk, m)

S := S \cup \{(m, \sigma)\}

\text{return } \sigma

\text{VER}(m, \sigma): \quad \text{return } (m, \sigma) \not\in S

to-do RSA signatures discussion forthcoming

13.4 Chinese Remainder Theorem

The multiplicative group $\mathbb{Z}_N^*$ has some interesting structure when $N$ is the product of distinct primes. We can use this structure to optimize some algorithms related to RSA.

History. Some time around the 4th century CE, Chinese mathematician Sun Tzu\(^3\) in his book *Sun Tzu Suan Ching* discussed problems relating to simultaneous equations of modular arithmetic:

"We have a number of things, but we do not know exactly how many. If we count them by threes we have two left over. If we count them by fives we have three left over. If we count them by sevens we have two left over. How many things are there?"

In our notation, he is asking for a solution $x$ to the following system of equations:

\[
\begin{align*}
x &\equiv_3 2 \\
x &\equiv_5 3
\end{align*}
\]

\(^3\)Not the same Sun Tzu who wrote *The Art of War.*

In the west, the method for solving these kinds of systems of equations are known as the **Chinese Remainder Theorem** (CRT). Below is one of the simpler formations of the Chinese Remainder Theorem, involving only two equations/moduli (unlike the example above, which has three moduli 3, 5, and 7):

**Theorem 13.6 (CRT)**

Suppose \( \gcd(r, s) = 1 \). Then for all integers \( u, v \), there is a solution for \( x \) in the following system of equations:

\[
\begin{align*}
x &\equiv_r u \\
x &\equiv_s v
\end{align*}
\]

Furthermore, this solution is *unique* modulo \( rs \).

**Proof**

Since \( \gcd(r, s) = 1 \), we have by Bezout’s theorem that \( 1 = ar + bs \) for some integers \( a \) and \( b \). Furthermore, \( b \) and \( s \) are multiplicative inverses modulo \( r \). Now choose \( x = var + ubs \). Then,

\[
x = var + ubs \equiv_r (va)0 + u(s^{-1}s) = u
\]

So \( x \equiv_r u \), as desired. Using similar reasoning mod \( s \), we can see that \( x \equiv_s v \), so \( x \) is a solution to both equations.

Now we argue that this solution is *unique* modulo \( rs \). Suppose \( x \) and \( x' \) are two solutions to the system of equations, so we have:

\[
\begin{align*}
x &\equiv_r x' \\
x &\equiv_s x'
\end{align*}
\]

Since \( x \equiv_r x' \) and \( x \equiv_s x' \), it must be that \( x - x' \) is a multiple of \( r \) and a multiple of \( s \). Since \( r \) and \( s \) are relatively prime, their least common multiple is \( rs \), so \( x - x' \) must be a multiple of \( rs \). Hence, \( x \equiv_{rs} x' \). So any two solutions to this system of equations are congruent mod \( rs \). \( \blacksquare \)

We can associate every pair \( (u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s \) with its corresponding system of equations of the above form (with \( u \) and \( v \) as the right-hand-sides). The CRT suggests a relationship between these pairs \( (u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s \) and elements of \( \mathbb{Z}_{rs} \).

For \( x \in \mathbb{Z}_{rs} \), and \( (u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s \), let us write

\[
x \xleftarrow{\text{crt}} (u, v)
\]

to mean that \( x \) is a solution to \( x \equiv_r u \) and \( x \equiv_s v \). The CRT says that the \( \xleftarrow{\text{crt}} \) relation is a bijection (1-to-1 correspondence) between elements of \( \mathbb{Z}_{rs} \) and elements of \( \mathbb{Z}_r \times \mathbb{Z}_s \).

In fact, the relationship is even deeper than that. Consider the following observations:

1. If \( x \xleftarrow{\text{crt}} (u, v) \) and \( x' \xleftarrow{\text{crt}} (u', v') \), then \( x + x' \xleftarrow{\text{crt}} (u + u', v + v') \). You can see this by adding relevant equations together from the system of equations. Note here that the addition \( x + x' \) is done mod \( rs \); the addition \( u + u' \) is done mod \( r \); and the addition \( v + v' \) is done mod \( s \).
2. If \( x \xleftarrow{\text{crt}} (u, v) \) and \( x' \xleftarrow{\text{crt}} (u', v') \), then \( xx' \xleftarrow{\text{crt}} (uu', vv') \). You can see this by multiplying relevant equations together from the system of equations. As above, the multiplication \( xx' \) is mod \( rs \); \( uu' \) is done mod \( r \); \( vv' \) is done mod \( s \).

3. Suppose \( x \xleftarrow{\text{crt}} (u, v) \). Then \( \gcd(x, rs) = 1 \) if and only if \( \gcd(u, r) = \gcd(v, s) = 1 \). In other words, the \( \xleftarrow{\text{crt}} \) relation is a 1-to-1 correspondence between elements of \( \mathbb{Z}_{rs}^* \) and elements of \( \mathbb{Z}_r^* \times \mathbb{Z}_s^* \).

The bottom line is that the CRT demonstrates that \( \mathbb{Z}_{rs} \) and \( \mathbb{Z}_r \times \mathbb{Z}_s \) are essentially the same mathematical object. In the terminology of abstract algebra, the two structures are isomorphic.

Think of \( \mathbb{Z}_{rs} \) and \( \mathbb{Z}_r \times \mathbb{Z}_s \) being two different kinds of names or encodings for the same set of items. If we know the “\( \mathbb{Z}_{rs} \)-names” of two items, we can add them (mod \( rs \)) to get the \( \mathbb{Z}_{rs} \)-name of the result. If we know the “\( \mathbb{Z}_r \times \mathbb{Z}_s \)-names” of two items, we can add them (first components mod \( r \) and second components mod \( s \)) to get the \( \mathbb{Z}_r \times \mathbb{Z}_s \)-name of the result. The CRT says that both of these ways of adding give the same results.

Additionally, the proof of the CRT shows us how to convert between these styles of names for a given object. So given \( x \in \mathbb{Z}_{rs} \), we can compute \( (x \mod r, x \mod s) \), which is the corresponding element/name in \( \mathbb{Z}_r \times \mathbb{Z}_s \). Given \( (u, v) \in \mathbb{Z}_r \times \mathbb{Z}_s \), we can compute \( x = u\mod r + v\mod s \mod rs \) (where \( a \) and \( b \) are computed from the extended Euclidean algorithm) to obtain the corresponding element/name \( x \in \mathbb{Z}_{rs} \).

From a mathematical perspective, \( \mathbb{Z}_{rs} \) and \( \mathbb{Z}_r \times \mathbb{Z}_s \) are the same object. However, from a computational perspective, there might be reason to favor one over the other. In fact, it turns out that doing computations in the \( \mathbb{Z}_r \times \mathbb{Z}_s \) realm is significantly cheaper.

to-do: more examples

Application to RSA

In the context of RSA decryption, we are interested in taking \( c \in \mathbb{Z}_{pq} \) and computing \( c^d \in \mathbb{Z}_{pq} \). Since \( p \) and \( q \) are distinct primes, \( \gcd(p, q) = 1 \) and the CRT is in effect.

Thinking in terms of \( \mathbb{Z}_{pq} \)-arithmetic, raising \( c \) to the \( d \) power is rather straightforward. However, the CRT suggests that another approach is possible: We could convert \( c \) into its \( \mathbb{Z}_p \times \mathbb{Z}_q \) representation, do the exponentiation under that representation, and then convert back into the \( \mathbb{Z}_{pq} \) representation. This approach corresponds to the bold arrows in Figure 13.1, and the CRT guarantees that the result will be the same either way.

Now why would we ever want to compute things this way? Performing an exponentiation modulo an \( n \)-bit number requires about \( n^3 \) steps. Let’s suppose that \( p \) and \( q \) are each \( n \) bits long, so that the RSA modulus \( N \) is \( 2n \) bits long. Performing \( c \mapsto c^d \) modulo \( N \) therefore costs about \( (2n)^3 = 8n^3 \) total.

The CRT approach involves two modular exponentiations — one mod \( p \) and one mod \( q \). Each of these moduli are only \( n \) bits long, so the total cost is \( n^3 + n^3 = 2n^3 \). The CRT

\[ \phi(pq) = |\mathbb{Z}_{pq}| = |\mathbb{Z}_p \times \mathbb{Z}_q| = (p - 1)(q - 1). \]

5Fun fact: this yields an alternative proof that \( \phi(pq) = (p - 1)(q - 1) \) when \( p \) and \( q \) are prime. That is, \( \phi(pq) = |\mathbb{Z}_{pq}| = |\mathbb{Z}_p \times \mathbb{Z}_q| = (p - 1)(q - 1). \]
approach is 4 times faster! Of course, we are neglecting the cost of converting between representations, but that cost is very small in comparison to the cost of exponentiation.

It’s worth pointing out that this speedup can only be done for the RSA inverse function. One must know $p$ and $q$ in order to exploit the Chinese Remainder Theorem, and only the party performing the RSA inverse function typically knows this.

13.5 The Hardness of Factoring $N$

Clearly the hardness of RSA is related to the hardness of factoring the modulus $N$. Indeed, if you can factor $N$, then you can compute $\phi(N)$, solve for $d$, and easily invert RSA. So factoring must be at least as hard as inverting RSA.

Factoring integers (or, more specifically, factoring RSA moduli) is believed to be a hard problem for classical computers. In this section we show that some other problems related to RSA are “as hard as factoring.” What does it mean for a computational problem to be “as hard as factoring?” More formally, in this section we will show the following:

**Theorem 13.7** Either all of the following problems can be solved in polynomial-time, or none of them can:

1. Given an RSA modulus $N = pq$, compute its factors $p$ and $q$.
2. Given an RSA modulus $N = pq$ compute $\phi(N) = (p - 1)(q - 1)$.
3. Given an RSA modulus $N = pq$ and value $e$, compute its inverse $d$, where $ed \equiv 1 \pmod{\phi(N)}$.
4. Given an RSA modulus $N = pq$, find any $x \not\equiv 1, -1 \pmod{N}$ such that $x^2 \equiv 1 \pmod{N}$.

To prove the theorem, we will show:

- if there is an efficient algorithm for (1), then we can use it as a subroutine to construct an efficient algorithm for (2). This is straightforward: if you have a subroutine factoring $N$ into $p$ and $q$, then you can call the subroutine and then compute $(p - 1)(q - 1)$.

---

6A polynomial-time algorithm for factoring is known for quantum computers.
if there is an efficient algorithm for (2), then we can use it as a subroutine to construct an efficient algorithm for (3). This is also straightforward: if you have a subroutine computing $\phi(N)$ given $N$, then you can compute $d$ exactly how it is computed in the key generation algorithm.

if there is an efficient algorithm for (3), then we can use it as a subroutine to construct an efficient algorithm for (4).

if there is an efficient algorithm for (4), then we can use it as a subroutine to construct an efficient algorithm for (1).

Below we focus on the final two implications.

**Using square roots of unity to factor $N$**

Problem (4) of Theorem 13.7 concerns a new concept known as square roots of unity:

**Definition 13.8 (Sqrt of unity)**

$x$ is a square root of unity modulo $N$ if $x^2 \equiv 1 \pmod{N}$. If $x \not\equiv -1 \pmod{N}$, then we say that $x$ is a non-trivial square root of unity.

Note that $\pm 1$ are always square roots of unity modulo $N$, for any $N ((\pm 1)^2 = 1$ over the integers, so it is also true mod $N$). But if $N$ is the product of distinct odd primes, then $N$ has 4 square roots of unity: two trivial and two non-trivial ones (see the exercises in this chapter).

**Claim 13.9**

Suppose there is an efficient algorithm for computing non-trivial square roots of unity modulo $N$. Then there is an efficient algorithm for factoring $N$. (This is the (4) $\Rightarrow$ (1) step in Theorem 13.7.)

**Proof**

The reduction is rather simple. Suppose ntsru is an algorithm that on input $N$ returns a non-trivial square root of unity modulo $N$. Then we can factor $N$ with the following algorithm:

```
FACTOR(N):
  x := ntsru(N)
  return gcd(N, x + 1) and gcd(N, x - 1)
```

The algorithm is simple, but we must argue that it is correct. When $x$ is a nontrivial square root of unity modulo $N$, we have the following:

$x^2 \equiv_p 1 \quad \Rightarrow \quad pq \mid x^2 - 1 \quad \Rightarrow \quad pq \mid (x + 1)(x - 1)$

$x \not\equiv_p 1 \quad \Rightarrow \quad pq \nmid (x - 1)$

$x \not\equiv_p -1 \quad \Rightarrow \quad pq \nmid (x + 1)$

So the prime factorization of $(x+1)(x-1)$ contains a factor of $p$ and a factor of $q$. But neither $x + 1$ nor $x - 1$ contain factors of both $p$ and $q$. Hence $x + 1$ and $x - 1$ must each contain factors of exactly one of $\{p, q\}$. In other words, $\{\gcd(pq, x - 1), \gcd(pq, x + 1)\} = \{p, q\}$. ■
Finding square roots of unity

Claim 13.10  If there is an efficient algorithm for computing $d \equiv \phi(N)^{-1}$ given $N$ and $e$, then there is an efficient algorithm for computing nontrivial square roots of unity modulo $N$. (This is the (3) $\Rightarrow$ (4) step in Theorem 13.7.)

Proof  Suppose we have an algorithm $\text{FIND\_D}$ that on input $(N, e)$ returns the corresponding exponent $d$. Then consider the following algorithm which uses $\text{FIND\_D}$ as a subroutine:

$sru(N)$:
- choose $e$ as a random $n$-bit prime
- $d := \text{FIND\_D}(N, e)$
- write $ed - 1 = 2^s r$, with $r$ odd
  // i.e., factor out as many 2s as possible
- $w \leftarrow \mathbb{Z}_N$
- if $\gcd(w, N) \neq 1$: // $w \notin \mathbb{Z}_N^*$
  - use $\gcd(w, N)$ to factor $N = pq$
  - compute a nontrivial square root of unity using $p$ & $q$
- $x := w^r \mod N$
- if $x \equiv_N 1$ then return 1
- for $i = 0$ to $s$:
  - if $x^2 \equiv_N 1$ then return $x$
- $x := x^2 \mod N$

There are several return statements in this algorithm, and it should be clear that all of them indeed return a square root of unity. Furthermore, the algorithm does eventually return within the main for-loop, because $x$ takes on the sequence of values:

$$w^r, w^{2r}, w^{4r}, w^{8r}, \ldots, w^{2^s r}$$

and the final value of that sequence satisfies

$$w^{2^s r} = w^{ed - 1} \equiv_N w^{(ed - 1) \phi(N)} = w^{1 - 1} = 1.$$

Conditioned on $w \in \mathbb{Z}_N^*$, it is possible to show that the algorithm returns a square root of unity chosen uniformly at random from among the four possible square roots of unity. So with probability 1/2, the output is a nontrivial square root. We can repeat this basic process $n$ times, and eventually encounter a nontrivial square root of unity with probability $1 - 2^{-n}$.  ■

to-do  more complete analysis
Exercises

13.1. Prove by induction the correctness of the \(extgcd\) algorithm. That is, whenever \((d, a, b) = extgcd(x, y)\), we have \(gcd(x, y) = d = ax + by\). You may use the fact that the original Euclidean algorithm correctly computes the GCD.

13.2. Prove that if \(a \equiv_{n} 1\) and \(b \equiv_{n} 1\), then \(gcd(a, b) \equiv_{n} 1\).

13.3. Prove that \(gcd(2a - 1, 2b - 1) = 2gcd(a, b) - 1\).

13.4. Prove that \(x^a \% n = x^a \% \phi(n) \% n\). In other words, when working modulo \(n\), you can reduce exponents modulo \(\phi(n)\).

13.5. In this problem we determine the efficiency of Euclid’s GCD algorithm. Since its input is a pair of numbers \((x, y)\), let’s call \(x + y\) the size of the input. Let \(F_k\) denote the \(k\)th Fibonacci number, using the indexing convention \(F_0 = 1; F_1 = 2\). Prove that \((F_k, F_{k-1})\) is the smallest-size input on which Euclid’s algorithm makes \(k\) recursive calls. *Hint:* Use induction on \(k\).

Note that the size of input \((F_k, F_{k-1})\) is \(F_k + 1\), and recall that \(F_k \approx \phi^{k+1}\), where \(\phi \approx 1.618\ldots\) is the golden ratio. Thus, for any inputs of size \(N \in [F_k, F_{k+1})\), Euclid’s algorithm will make less than \(k \leq \log_2 N\) recursive calls. In other words, the worst-case number of recursive calls made by Euclid’s algorithm on an input of size \(N\) is \(O(\log N)\), which is linear in the number of bits needed to write such an input.\(^7\)

13.6. Consider the following symmetric-key encryption scheme with plaintext space \(M = \{0, 1\}^\lambda\). To encrypt a message \(m\), we “pad” \(m\) into a prime number by appending a zero and then random non-zero bytes. We then multiply by the secret key. To decrypt, we divide off the key and then strip away the “padding.”

The idea is that decrypting a ciphertext without knowledge of the secret key requires factoring the product of two large primes, which is a hard problem.

<table>
<thead>
<tr>
<th>KeyGen:</th>
<th>Enc((k, m \in {0, 1}^\lambda)):</th>
</tr>
</thead>
<tbody>
<tr>
<td>choose random (\lambda)-bit prime (k)</td>
<td>(m' := 10 \cdot m)</td>
</tr>
<tr>
<td>return (k)</td>
<td>while (m') not prime:</td>
</tr>
<tr>
<td>Dec((k, c)):</td>
<td>(d \leftarrow {1, \ldots, 9})</td>
</tr>
<tr>
<td>(m' := c/k)</td>
<td>(m' := 10 \cdot m' + d)</td>
</tr>
<tr>
<td>while (m') not a multiple of 10:</td>
<td>return (k \cdot m')</td>
</tr>
<tr>
<td>(m' := \lfloor m'/10 \rfloor)</td>
<td></td>
</tr>
<tr>
<td>return (m'/10)</td>
<td></td>
</tr>
</tbody>
</table>

Show an attack breaking CPA-security of the scheme. That is, describe a distinguisher and compute its bias. *Hint:* ask for any two ciphertexts.

13.7. Explain why the RSA encryption exponent \(e\) must always be an odd number.

\(^7\)A more involved calculation that incorporates the cost of each division (modulus) operation shows the worst-case overall efficiency of the algorithm to be \(O(\log^2 N)\) — quadratic in the number of bits needed to write the input.
13.8. A **simple power analysis (SPA)** attack is a physical attack on a computer, where the attacker monitors precisely how much electrical current the processor consumes while performing a cryptographic algorithm. In this exercise, we will consider an SPA attack against the ModExp algorithm shown in Section 13.2.

The ModExp consists mainly of squarings and multiplications. Suppose that by monitoring a computer it is easy to tell when the processor is running a squaring vs. a multiplication step (this is a very realistic assumption). This assumption is analogous to having access to the printed output of this modified algorithm:

```
ModExp(m, e, N): // compute m^e % N
if e = 0: return 1
if e even:
    res := ModExp(m, \frac{e}{2}, N)^2 % N
    print “square”
if e odd:
    res := ModExp(m, \frac{e-1}{2}, N)^2 \cdot m % N
    print “square”
    print “mult”
return res
```

Describe how this SPA trace lets the attacker *completely* learn the value \( e \). Note that in RSA, it is indeed the exponent that is secret, so this attack leads to key recovery for RSA. **Hint:** think about what “\( e/2 \),” “\( (e - 1)/2 \),” and “\( e \) is odd” mean, in terms of the bits of \( e \).

13.9. The Chinese Remainder Theorem states that there is always a solution for \( x \) in the following system of equations, when \( \gcd(r, s) = 1 \):

\[
\begin{align*}
    x & \equiv_r u \\
    x & \equiv_s v
\end{align*}
\]

Give an example \( u, v, r, s \), with \( \gcd(r, s) \neq 1 \) for which the equations have no solution. Explain why there is no solution.

13.10. Consider a rectangular grid of points, with width \( w \) and height \( h \). Starting in the lower-left of the grid, start walking diagonally northeast. When you fall off end the grid, wrap around to the opposite side (i.e., Pac-Man topology). Below is an example of the first few steps you take on a grid with \( w = 3 \) and \( h = 5 \):
CHAPTER 13. RSA & DIGITAL SIGNATURES

Show that if \( \gcd(w, h) = 1 \) then you will eventually visit every point in the grid.

*Hint:* Derive a formula for the coordinates of the point you reach after \( n \) steps.

13.11. Bob chooses an RSA plaintext \( m \in \mathbb{Z}_N \) and encrypts it under Alice’s public key as \( c \equiv m^e \). To decrypt, Alice first computes \( m_p \equiv_p c^d \) and \( m_q \equiv_q c^d \), then uses the CRT conversion to obtain \( m \in \mathbb{Z}_N \), just as expected. But suppose Alice is using faulty hardware, so that she computes a *wrong value* for \( m_q \). The rest of the computation happens correctly, and Alice computes the (wrong) result \( \hat{m} \). Show that, no matter what \( m \) is, and no matter what Alice’s computational error was, Bob can factor \( N \) if he learns \( \hat{m} \).

*Hint:* Bob knows \( m \) and \( \hat{m} \) satisfying the following:

\[
m \equiv_p \hat{m} \\
m \equiv_q \hat{m}
\]

13.12. (a) Show that given an RSA modulus \( N \) and \( \phi(N) \), it is possible to factor \( N \) easily.

*Hint:* you have two equations (involving \( \phi(N) \) and \( N \)) and two unknowns (\( p \) and \( q \)).

(b) Write a pari function that takes as input an RSA modulus \( N \) and \( \phi(N) \) and factors \( N \). Use it to factor the following 2048-bit RSA modulus. *Note:* take care that there are no precision issues in how you solve the problem; double-check your factorization!

\[
N = 133140272889335192221084096266217447630383165238367168854709484 \\
25323594058691714048266918225363682852699928294472079880183176174867 \\
62035895223096986447599330583492249263662729864038596531894556546 \\
013113154346823212271748927859647994534586133553218022983848188421 \\
465442089919096105423447682944817251837572222421917115971063026806 \\
58714128758703726515065366909432311668657453655886659164736105311 \\
0465160130696693686673412655801774439375116111219195769578488559 \\
88290239724830993391161475065854696820621069072502248333238754832 \\
69861623840522138125214513743991909880085955274389382721844956661 \\
1138745095472065761807
\]

\[
\phi = 133140272889335192221084096266217447630383165238367168854709484 \\
25323594058691714048266918225363682852699928294472079880183176174867 \\
62035895223096986447599330583492249263662729864038596531894556546 \\
013113154346823212271748927859647994534586133553218022983848188421 \\
465442089919096105423447682944817251837572222421917115971063026806 \\
58714128758703726515065366909432311668657453655886659164736105311 \\
0465160130696693686673412655801774439375116111219195769578488559 \\
88290239724830993391161475065854696820621069072502248333238754832 \\
69861623840522138125214513743991909880085955274389382721844956661 \\
1138745095472065761807
\]

13.13. True or false: if \( x^2 \equiv_n 1 \) then \( x \in \mathbb{Z}_N^* \). Prove or give a counterexample.

13.14. Discuss the computational difficulty of the following problem:

Given an integer \( N \), find a nonzero element of \( \mathbb{Z}_N \setminus \mathbb{Z}_N^* \).

If you can, relate its difficulty to that of other problems we’ve discussed (factoring \( N \) or inverting RSA).

235
13.15. (a) Show that it is possible to efficiently compute all four square roots of unity modulo \(pq\), given \(p\) and \(q\). \textit{Hint:} CRT!

(b) Implement a \texttt{pari} function that takes distinct primes \(p\) and \(q\) as input and returns the four square roots of unity modulo \(pq\). Use it to compute the four square roots of unity modulo \(1052954986442271985875778192663 \times 611174539744122090068393470777\).

13.16. Show that, conditioned on \(w \in \mathbb{Z}_N^*\), the \texttt{SqrtUnity} subroutine outputs a square root of unity chosen uniformly at random from the 4 possible square roots of unity. \textit{Hint:} use the Chinese Remainder Theorem.

13.17. Suppose \(N\) is an RSA modulus, and \(x^2 \equiv y^2 \mod N\), but \(x \neq_N \pm y\). Show that \(N\) can be efficiently factored if such a pair \(x\) and \(y\) are known.

13.18. Why are \(\pm 1\) the only square roots of unity modulo \(p\), when \(p\) is an odd prime?

13.19. When \(N\) is an RSA modulus, why is squaring modulo \(N\) a 4-to-1 function, but raising to the \(e\)th power modulo \(N\) is 1-to-1?

13.20. Implement a \texttt{pari} function that efficiently factors an RSA modulus \(N\), given only \(N\), \(e\), and \(d\). Use your function to factor the following 2048-bit RSA modulus. \textit{Note:} \texttt{pari} function \texttt{valuation(n, p)} returns the largest number \(d\) such that \(p^d \mid n\).

\[
\begin{align*}
N &= 15771389270555006490975063247569186987752676765283393212873561871172625613193643033170582677236726549900329193771645478888249992131117065951077245304317542978715216577264400482780645742941405647092530098401686213021840143101927659590154835888878786106624609937218511908418897990873152584082212461847511180666690936944585390792304663763886471865146718283897613617078370412411019301687974905838294389148932399661048471814117247898148039982257697888167601010511378647288478239379406136827038003563427159360951322605573614212415962670795238081910384512700791242895829113046494206822583621324213115022256956958205924967
\end{align*}
\]
\[
\begin{align*}
e &= 3275988664839226242682853753493159017722529826619266755049517723245018308645023365950867709254463108379780795236766113905163496666905258066495934057774395712118774014408282455244138049333893140361998542639919865601982731556307233588691392913730537367814886759427468288411988660382294707772963323546273242328708595852831517584489590815901478740429497984201739085133315157558366579073784876557843387314612619257092501513237378707481710620893090646766081314097886010670771307472326030259629322456826203119495358404553830594521756402746101322509998099867316014496771937442676416172816811394687608081636258360752718165973
\end{align*}
\]
\[
d &= 1384769997342637754981004435671327591821124457347447041195020109217275520780316201948448712786675422680481999889426593934193852825864234633738686176601555755842198986431725335631626097859462929596839116109082630845096923641858596338471746704714837349583807860867015736751482578304229734465052889825974575774123039929795232302127499782189937839800133705786918948873495185374832763188350213513952366499029633402320771390040868326243223664645438899178442633342438
\]

236
13.21. In this problem we’ll see that it’s bad to choose RSA prime factors \( p \) and \( q \) too close together.

(a) Let \( N = pq \) be an RSA modulus. Show that if you know \( N \) and \( \delta = |p - q| \) then you can efficiently factor \( N \).

(b) Alice generated the following RSA modulus \( N = pq \) and lets you know that \( |p - q| < 10000 \). Factor \( N \):

\[
N = 8746775183889666368698301429866315858010681593301504361505917406 \\
6796033865473978646639928231278257025792316921962329748948203153 \\
6330137181753809697696012524918354799230845322374618855425387176 \\
952865438204578597577886578695728149769838269751962961 \\
89933125540534657194681056148376499916124302583048808171824215 \\
46959489811921627100521215353092540247263578195579371323933498 \\
4944582832381081284358218758725674901184016564638718414715249093 \\
7576397558586257839279875912167586535344784950644107808310123 \\
93028257089819030166822729137968982546143104625315706571887837795 \\
31855302859423676881
\]

13.22. Here is a slightly better method to factor RSA moduli whose factors are too close together. As before, let \( N = pq \).

(a) Define \( t = (p + q)/2 \). Note that when \( p \) and \( q \) are close, \( t \) is not much larger than \( \sqrt{N} \). Show that:

- \( t^2 - N \) is a perfect square.
- Given \( t \), it is possible to efficiently factor \( N \). (Hint: write \( t^2 - N = s^2 \) for some \( s \).)

(b) Factor the following 2048-bit number (whose two prime factors are guaranteed to be close enough for the factoring approach to work in a reasonable amount of time, but far enough apart that you can’t do the trial-and-error part by hand). How close were the factors (how large was \( |p - q| \)?)

\[ N = 5142028666426265919867363048226343880193216864011643244558701956114 \\
553317880043298927487456460284103951463512024249329243228109624011 \\
9153924118872402640312768676725582505608189696295971582380690811 \\
13166833180822330775723858221018120959641916112573524246791879 \\
1313505969852560011034079059587975345573842266766492356686762134 \\
658383064511337433089249621256291078256814295739499101301135200 \\
9186652113944314387354865996785413693758878409138424399026159037108 \\
043724221865116794083419481223638129978639545727755987957552254116 \\
612726596118528071785474551058545959198869986780286733916614335663 \\
3723093246569630373323 \]