**Introduction to RSA and How It Can Use Square and Multiply**

In this workbook you will learn about the RSA public key algorithm and why its implementation may be vulnerable to a timing side channel attack, including the mathematical background required.

**Private and Public Key Cryptography**

Modern cryptographic systems can be split up into two types: symmetric/private key cryptography and asymmetric/public key cryptography.

Private key cryptography involves some “shared secret” between the two parties sending messages – the key. We can think of this type of system like having a strong box and padlock to which both you and your friend have a key. If you want to send a message to your friend, you simply put the message in the box, use your key to lock it with the padlock and then send the box to them. You friend can use their copy of the key to unlock the padlock and read the message in the box. As long as the two of you are the only people with the key your message with be safe from anyone else who might want to read it. An example of this type of cryptosystem is the Advanced Encryption Standard, AES, which you’ll see in more detail later if time allows.

Now, what if your friend doesn’t already have a key to the padlock? Or you’re sending a message to someone that you don’t trust with a copy of your key? If we think about our padlocks again, the analogue begins the same: You write your message and lock it in the box using your padlock and key. You send the box to your friend, but this time they instead lock the box with their own padlock and key and send it back to you. Now the box has two padlocks on it, so you unlock yours and send the box back to your friend one more time. You friend now has a box with only their own padlock on it, which they can unlock and open the box to read your message. You have securely sent a message to your friend without needing to have any shared secret between you. This is the basic idea behind public key cryptography.

This is the type of cryptography that RSA fits into which, is the system you’ll be cheating your way into in this section. For this algorithm, the person receiving the message with have two keys: one public and one private. As you might guess, the public key is available to everyone and is used to encrypt a message. The private key is known only to the recipient and is used for decryption. Before we go into any further details, there are a couple of mathematical concepts you’ll need to know about first.

**Prime numbers**

A prime number is a number with only two factors: 1 and itself. For example:

* 7 is prime because we can only factor 7 = 1 x 7.
* 24 is not prime because we have 24 = 1 x 24 and 24 = 2 x 24, 3 x 8 and 4 x 6.

A pair of numbers are called coprime if they do not share any prime factors. For example:

* 12 = 3 × 4 and 22 = 2 × 11 are coprime.
* 14 = 2 × 7 and 24 = 23 × 3 are not coprime.

**Modular arithmetic**

Modular arithmetic is a form of arithmetic on integers in which numbers wrap around when they reach a certain value, which we call the modulus. Luckily, you already know exactly how to do this for certain moduli! We all do this for the modulus of 12 or 24 when we tell time. Think about the 24-hour clock: every day at midnight the time resets to 0. For example, 3 hours after 22:00, the time is 01:00, not 25:00. This process of wrapping a number around a modulus is called reduction modulo N, where N is the modulus. In the above example, we have reduced 25 modulo 24.

If two numbers a and b reduce to the same number modulo N, we say that a and b are congruent modulo N. We write this as 𝑎 ≡ 𝑏 mod 𝑁. Mathematically, this is equivalent to there being an integer 𝑘 such that 𝑎 = 𝑏 + 𝑘𝑛. For example:

* 49 ≡ 25 ≡ 1 mod 24.
* 3 ≡ −1 mod 4.
* 3 × 4 ≡ 0 mod 12.

An integer 𝑎 is an inverse of an integer 𝑏 modulo 𝑁, if 𝑎 × 𝑏 ≡ 1 mod 𝑁. If 𝑎 is an inverse of 𝑏 modulo 𝑁, then 𝑏 is an inverse of 𝑎 modulo 𝑁. For example:

* 5 is an inverse of 2 modulo 9 because 5 × 2 ≡ 1 mod 9.
* 2 is an inverse of 5 modulo 9 because 2 × 5 ≡ 1 mod 9.
* 14 is also an inverse of 2 modulo 9 because 14 × 2 ≡ 1 mod 9

In particular, an integer a will have an inverse modulo N if a and N are coprime. Like a = 5 and N = 9 = 32 in the above, for example.

The RSA algorithm uses the following result in particular:

If r and s are prime numbers and a is an integer that has no common divisors with either r or s, then .

For example, r = 3 and s = 7 are both prime, and a = 4 = 2 x 2 is an integer which does not have r or s as a divisor. We have r × s = 21 and so

***Exercise:*** *Think of three other numbers r, s and a that satisfy the conditions and show that the relation above holds.*

**The RSA Algorithm**

Like most public key encryption systems, RSA depends on a problem that is easy to solve in one direction but hard to solve in the other. Specifically, RSA uses large prime numbers and the fact that multiplying them together is much easier than factoring them out.

***Exercise:*** *What is the product of the prime numbers 17 and 23? What are the prime factors of the number 247? Which calculation took you longer?*

The primes in the exercise aren’t very big at all, but factoring still takes much longer than multiplying. The primes used in RSA are much larger, around 512 bits, 1,024 bits or 2,048 bits long. In the exercise you probably needed to check the primes in order starting from 2 to see which ones would divide 247 – this would take an extremely long time for such large prime factors!

Here is how the RSA algorithm works:

* Starting with two large primes r and s, calculate N = r × s, this will be our modulus.
* Choose an integer e which has an inverse modulo (r-1)(s-1), this will be the encryption exponent.
* Find the inverse of e and call it d, this will be the decryption exponent.
* To encrypt a plaintext message P, calculate

to obtain the ciphertext C.

* To decrypt a ciphertext C, calculate

to recover the plaintext P.

We mentioned earlier that RSA uses two keys, one public and one private. Here the public key will be the pair of integers (n,e) which anyone can use as above to encrypt their plaintext message P. The private key will be the integer d, which can be used with n from the public key to decrypt a ciphertext message C as above.

We’ll explain this process more thoroughly by following a worked example with some smaller primes: r = 13 and s = 29.

* First, we calculate N = r × s = 377.
* Next we need an integer e that has an inverse modulo (r-1)(s-1) = 12 ×28 = 336. Recall that this is the same as choosing an integer which is coprime with 336 = 24 × 3 × 7. We’ll choose e = 65 = 5 × 13 for this example.
* Now we need to find the inverse of 65 modulo 336 (we can do using the extended Euclidean algorithm, don’t worry if you haven’t seen this), which is d = 305.
* Let’s say the message we want to send is “secret”. To encrypt we start by writing this in ASCII (look this up if you haven’t seen it before) as 73 65 63 72 65 74. We then split this into blocks that are smaller than N (since our N is small we will leave them as two digit blocks, but they could be bigger if N allowed) and encrypt each block as :

7365 mod 377 229

6565 mod 377 = 78

6365 mod 377 = 33

7265 mod 377 = 206

7465 mod 377 = 198

So our ciphertext becomes C = 229 78 33 206 78 198

* If instead we had received this ciphertext C, we can use the decryption exponent d we found earlier to calculate for each block of the ciphertext:

229305 mod 377 = 73

78305 mod 377 = 65

…etc

Thus, we have recovered the original plaintext 73 65 63 72 65 74 = “secret”.

***Exercise:*** *Encrypt a short message using RSA and then decrypt to check that you get back to the original plaintext. You can come up with your own primes and encryption exponent, then find the decryption exponent, or use the following: r = 17, s = 31, e = 91, d = 211. You’ll want to encode your message into numbers using ASCII or similar first, just make sure if you’re choosing your own primes r and s, that your modulus N is bigger than any number you’re representing letters with, or your message will be lost!*

**RSA Square and Multiply**

As you’ll have noticed from the previous exercise, RSA involves calculating a lot of exponents. The obvious way to do these is to simply multiply the number by itself the number of times specified by the exponent, so for example

45 = 4 × 4 × 4 × 4 × 4 = 1024.

This is a lot of calculations to do when our exponents are large so it would be nice if we had a more efficient way of doing it.

In the RSA algorithm, the approach taken to achieve this is called square and multiply. The first step is to write the exponent in binary, which you’ll be familiar with from earlier in this project. So in our example above, we write 5 = 0b101. Now, for the first 1 in this representation we simply list the number. Then whenever we see a 0 we will calculate a square, and whenever we see a 1 we will square and then multiply by the original number. So we have

1. For the first 1, write the number: 4
2. For the 0, square: 42 = 16
3. For the 1, square and multiply: (16)2 × 4 = 256 × 4 = 1024

By using square and multiply we have done our calculation in only 3 steps instead of five. This might not seem like much of a saving, but once the exponents become large it can be a big saving indeed. For example, if our exponent was 65, this is 0b1000001 in binary, so we’d write our number for the first 1, square five times for the 0’s, then square and multiply for the final 1. By doing this we’d be able to calculate a number to the power of 65 in only seven steps.

This efficiency increase seems great, but it actually also introduces an opportunity for a timing side-channel attack. Recall that this type of attack uses the timing differences of a crypt implementation to deduce information about the data being encrypted. For example, RSA’s square and multiply algorithm, the two possible calculations “square” and “square and multiply” take different amounts of time to perform. As these calculations are related to the exponent we’re calculating, we can use this to read off the bits of the exponent by looking at a power trace and noticing how long a calculation is taking. Short calculations imply a 0 bit and long calculations imply a 1 bit. Other more general conditional paths in the multiplication or squaring instructions could also be manipulated in a way that introduces a timing difference and thus allows bits of d to be leaked (as you will see in the next exercises!) In RSA, the exponent is the key that needs to be kept secret for the information to be secure, so being able to do this will give us a huge advantage in finding it!