Public Key Cryptography

Federico Casu

December 28, 2023

1 Introduction

As we are becoming familiar with it, let's delve into the application of public key cryptography in securing communications. Figure 1 illustrates the fundamentals of a public key-based communication system:

1. Alice, who wants to send a confidential message to Bob, knows Bob's public key K^B_{pub}. To encrypt the message x, Alice executes the encryption algorithm $E(\cdot)$, taking as input the plaintext x and Bob's public key K^B_{pub}.
2. Bob, who wants to read the incoming message, executes the decryption algorithm $E^{-1}(\cdot)$, taking as input the ciphertext y and Bob's private key K^B_{priv}.

Let's establish a formal definition of a public key encryption scheme.

> A public key encryption scheme is a triple of algorithms, $\langle E, D, G\rangle$, such that they fulfill the following properties:
>
> 1. G is a randomized algorithm that outputs a pair of keys, namely $\langle K_{pub}, K_{priv}\rangle$.
>
> $$G : \{0,1\}^k \rightarrow K = \{0,1\}^n \times \{0,1\}^n$$
>
> 2. E is a randomized algorithm that, given inputs of a plaintext $x \in M$ and a public key K_{pub}, outputs a ciphertext $y \in C$.
>
> $$E : K \times M \rightarrow C$$
>
> 3. D is a **deterministic** algorithm that, given inputs of a ciphertext $y \in C$ and a private key K_{priv}, outputs a plaintext $x \in M$.
>
> $$D : K \times C \rightarrow M$$
>
> 4. The encryption scheme fulfills the **consistency property**, *i.e.*
>
> $$\forall\ \langle K_{pub}, K_{priv}\rangle, \forall\ x \in M \rightarrow E^{-1}(K_{priv}, E(K_{pub}, x)) = x$$

Being an encryption scheme means provide some security properties. We would like to give you an informal definition of the security properties of a public key encryption scheme:

1. Given any ciphertext y and the public key used to encrypt it, K_{pub}, it must be infeasible to obtain the plaintext x such that $y = E(K_{pub}, x)$.
2. Given any public key, it must be infeasible to obtain the corresponding private key.

Such properties rely on some algebraic constructs. In particular, public key cryptography exploits a certain type of mathematical functions called **one-way** functions. The *one-wayness* property states that a function f is said to be one-way if:

1. f is easy to compute
2. f^{-1} is hard to compute

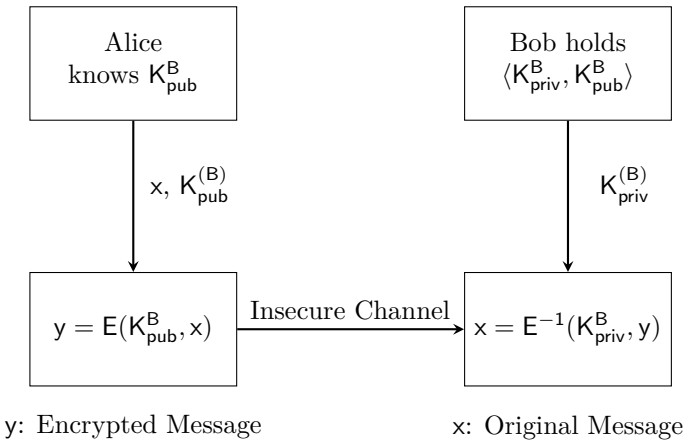

Figure 1: Public Key Cryptography - Simple communication scenario.

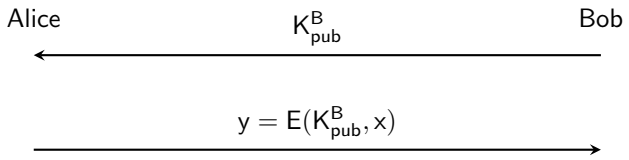

Figure 2: Simple Public Key Encryption (PKE) scheme.

In what way one-way functions are related to public key cryptography? Let's make an example.

Example 1. The RSA cryptosystem exploits the integer factorization as the underlying one-way function: multiplying two primes is easy but factoring the resulting product is computationally infeasible. RSA key generation algorithm security rely on the integer factorization problem.

Public key cryptography is not *perfect*. Let's prove it:

1. Let's consider that an attacker managed to obtain a ciphertext y.
2. Let's consider a plaintext $\hat{x} \in M$ such that $0 < P(X = x) < 1$. Knowing the receiver's public key (by definition, as the name suggests, is public), we encrypt the choosen plaintext $\hat{x}$ with the public key, *i.e.*, $\hat{y} = E(K_{pub}, \hat{x})$. Now we are in front of two scenarios:
   (a) $y = \hat{y} \rightarrow x = \hat{x} \rightarrow P(X = \hat{x} \mid Y = y) = 1 \neq P(X = x)$
   (b) $y \neq \hat{y} \rightarrow x \neq \hat{x} \rightarrow P(X = \hat{x} \mid Y = y) = 0 \neq P(X = x)$

> A public key encryption scheme is **not** perfect because it's always possible to find a plaintext whose a-posteriori probability is either 0 or 1, which is different from $P(X = x)$.

Let's explore some real-world applications of public key cryptography:

1. The first application is in digital communications. The primary distinction between a symmetric key encryption scheme and a public key encryption scheme is that the former requires an initial phase during which the communicating parties securely agree on the key used for encryption. In a public key encryption scheme, there is no need to securely agree on a session key. The encryption phase utilizes the public key, and the decryption phase uses the private key. Importantly, even if the public key is publicly known, the private key does not need to be transmitted over an insecure channel: only the owner can decrypt any messagge encrypted with the corresponding public key.
2. Moving forward, we will delve into the study of digital signatures. The asymmetric nature of the encryption scheme makes public key cryptography more suitable for digital signature schemes compared to symmetric ones.

Public key encryption schemes have a big problem: public key encryption is 2-3 orders of magnitude slower than symmetric key encryption! A very simple communication scheme based on public key

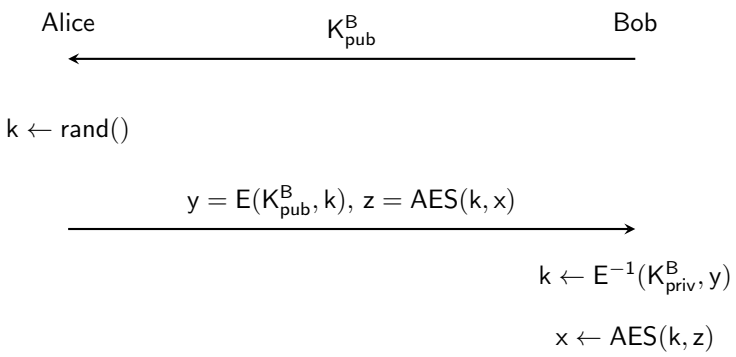

Figure 3: Digital Envelope communication protocol.

Algorithm Family	Cryptosystem	Security Level			
		80	128	192	256
Integer Factorization	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete Logarithm	DH, DSA, ElGamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

Figure 4: Security level of the most used encryption schemes.

cryptography (Figure 2) cannot not be used in practice. A solution to this problem is given by the digital envelope: public key cryptography is used only to encrypt the symmetric key (Figure 3). What about the *security level* of a public key encryption scheme? The formal definition of security level is the following:

Definition 1. An encryption scheme has security level of n bits if the best known algorithm to break it requires 2^n steps.

The relationships between the key length and the security level in a symmetric encryption scheme is straightforward: if the key is n bits long then the encryption scheme has security level of n bits. For public key encryption schemes is a bit different: there is no linear relation between key lenght and security level: as a rule of thumb, the computational complexity grows roughly with the cube bit length.

1.1 Key Authentication: MITM Attacks

With respect to a passive attacker, any public key encryption scheme is secure. If an attacker intercepts any ciphertext, they must solve the underlying mathematical problem to be able to decrypt the intercepted cyphertext. Therefore, if the encryption scheme is well-designed (meaning that the encryption scheme exploits a good *one-way* function), the attacker will not be able to decrypt or obtain the private key in a reasonable amount of time.
Now, considering an active attacker, let's explore the following scenario (Figure 5):

1. Oscar is a "bad guy" who wishes to secretly read the messages that Alice and Bob are going to exchange.
2. Bob needs to let Alice know his current public key: he simply sends his public key through an insecure channel. Bob doesn't worry about the public key and he thinks that Alice correctly receives his public key.
3. Oscar manages to intercept Bob's public key. He's a smart guy and he thinks to impersonate Bob by sending to Alice a freshly generated public key K^O_{pub}.
4. Alice receives Oscar's public key but she thinks that it was Bob who sent the message. So she encrypts the confidential message with the received public key and sends it through the insecure channel.

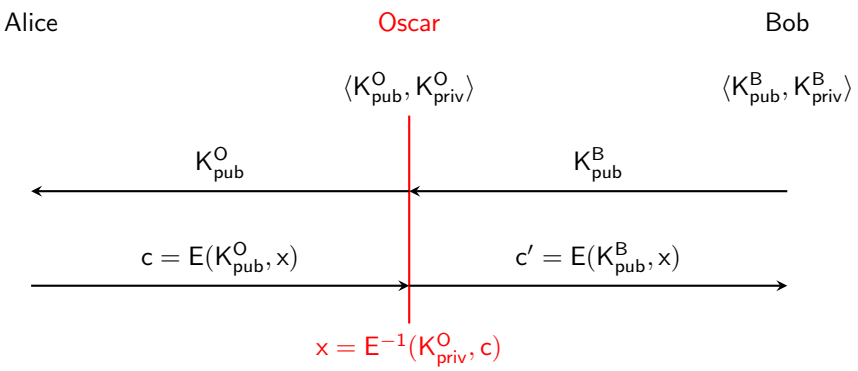

Figure 5: Man In The Middle (MITM) attack.

5. Oscar, once again, intercepts the encrypted message. Since the latter was encrypted using his public key, he is able to correctly decrypt Alice's message, read x and send $c' = E(K^B_{pub}, x)$ to Bob.

6. Bob receives Oscar's messagge thinking that it was sent by Alice: both Alice and Bob are scammed by Oscar and no one knows it!

How can we prevent a *man-in-the-middle* attack? Let's study a possible solution:

Example 2. Instead of sending their own public key, each user registers their public key with a *trusted read-only repository*. Whenever Alice wants to send a confidential message to Bob, she queries the trusted repository, asking for Bob's public key.

Note that a trusted repository does not secure a public encryption scheme from MITM attacks: an adversary could intercept the query for a public key and craft a fake response substituting the legit public key with any key they want. Later on, we will study how public key encryption schemes exploit digital signatures to mitigate MITM attacks.

1.2 Encryption Randomization

There are some scenarios that require to be studied because a plain public key encryption scheme is vulnerable. Let's examine it:

1. Consider an auction. Alice, a bidder participating in Bob's auction, sends her bid to Bob by encrypting it with Bob's public key.

2. Oscar, a bad guy, wishes to win the auction by spending the least amount of money possible. He does this by sending a bid with an amount equal to the maximum bid from honest bidders plus 1 dollar. Since Oscar knows the market value of the goods being auctioned, he knows that the bids can be represented with 10 bits, meaning each bid is 10 bits long.

3. Oscar manages to obtain each bid sent to Bob. Since Oscar knows Bob's public key (if Oscar couldn't obtain Bob's public key, then the latter would be a private key!), he performs the attack described in Algorithm 1.

4. The attack complexity is very low: $O(2^{10})$. Oscar can brute-force each intercepted message (perhaps in parallel) and obtain the highest bid that was sent to Bob. Once he has done that, he can device all the participants by sending to Bob $y = E(K^B_{pub}, \max + 1)$.

Algorithm 1 Small plaintext attack: algorithm

```
for i ← 0 to 2^10 do
    y' ← encrypt(K_pub, i)
    if y = y' then return i
    end if
end for
```

Is there a way to prevent Bob and the bidders from being scammed? Yes:

1. Each bidder generates a random number $g \in \{0,1\}^r$.
2. Now, instead of encrypting only the bid, the message to be encrypted is the following:

$$x' = x \mid g$$

3. Oscar, to obtain the bid x from the encrypted message, needs to executes Algorithm 2.
4. The attack's complexity gets multiplied by 2^r!

Algorithm 2 Small plaintext attack: algorithm (with salt)

```
for i ← 0 to 2^10 do
    for j ← 0 to 2^r do
        y' ← encrypt(K_pub, {i, j})
        if y = y' then return i
        end if
    end for
end for
```

2 RSA Cryptosystem

RSA cryptosystem was invented by a group of researchers (R. L. Rivest, A. Shamir and L. Adleman) working at MIT Laboratory for Computer Science. The encryption scheme was first presented in a paper called "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (February 1978). Let's delve into the fundamentals of the RSA encryption scheme.

2.1 Key Generation Algorithm

Formally, the RSA key generation algorithm is composed by 5 steps:

Algorithm 3 RSA Key Generation Algorithm

1: Choose two large prime numbers, namely p and q.
2: Compute the quantity $n = p \cdot q$ called *modulus*.
3: Compute the quantity $\phi = (p-1)(q-1)$.
4: Choose e such that $1 < e < \phi$ and $\gcd(e, \phi) = 1$.
5: Choose d such that $1 < d < \phi$ and $e \cdot d \equiv 1 \pmod{\phi}$.

The key generation algorithm outputs two quantities:

$$\text{public key} : \langle e, n \rangle$$
$$\text{private key} : \langle d, n \rangle$$

Please remember that the correctness of RSA is assured by Step 4 of Algorithm 3.

2.2 Encryption and Decryption Algorithm

RSA encrypts plaintext x, where we consider the bit string representing x to be an element in $Z_n = \{0, 1, \ldots, n-1\}$. As a consequence, the binary value of the plaintext x must be less than n. The same holds for the ciphertext.

$$\text{Encryption} : y = x^e \bmod n$$
$$\text{Decryption} : x = y^d \bmod n$$

2.3 RSA Performance

Now we need to discuss the performance of the key generation algorithm. In particular, generating a pair of keys $\langle K_{pub}, K_{priv} \rangle$ involves some operations:

1. Find two large prime numbers. Obviously, each prime needs to be randomly generated (otherwise an attacker could guess the prime with a probability larger than 0.5).
2. Test whether or not the randomly generated number is a prime number.
3. Find a public and a private exponent which fulfill the required properties: $\gcd(e, \phi) = 1$ and $e \cdot d \equiv 1 \pmod{\phi}$.

So the prime generation algorithm could be pseudo coded as follows:

Algorithm 4 Prime Generation Algorithm

```
repeat
    p ← random()
until is_prime(p) == False
```

To establish the complexity of Algorithm 4 we need to answer the following questions:

1. How many iterations do we need to execute? In other words, how likely is to find a prime $p < x$?
2. How difficult is it to generate a random number?
3. How difficult is it to test whether an integer is prime or not?

Let's answer the previous questions:

Fact 1. Let's call $\pi(x)$ the number of primes which are $\leq x$. If x is sufficiently big, then $\pi(x)$ could be approximated to $\pi(x) = \frac{x}{\ln(x)}$.

If we test only odd integers, meaning that we test half the numbers $\in [1, x]$, the probability to find a prime $p \in [1, x]$ is:

$$P(p \text{ is prime}) = \frac{\pi(x)}{\text{number of odds} \in [2, x]} = \frac{x/\ln(x)}{x/2} = \frac{2}{\ln(x)} = O\left(\frac{1}{\ln(x)}\right)$$

So, the expected number of trials needed to find a prime is $O(\ln(x))$, which is quite good (and easy). What about the complexity of a primality test? Primality tests are computationally much easier than factorization. In particular, practical implementations of primality tests are *probabilistic*: at the question "is $p*$ prime?" they answer

1. $p*$ is composed which is always a true statement;
2. $p*$ is prime, which is only true with a high probability.

Now we need to discuss how difficult is to find a pair of keys such that fulfill the properties:

$$1)\ \gcd(e, \phi) = 1$$
$$2)\ e \cdot d \equiv 1 \pmod{\phi}$$

To find the public exponent e and the private exponent d at the same time, we can exploit the Extended Euclidean Algorithm. To find a public exponent e we can use the plain version of the Euclidean Algorithm but it would very inefficient. Let's consider the following *Diophantine equation*:

$$\gcd(e, \phi) \equiv t \cdot e + s \cdot \phi \tag{1}$$

If $\gcd(e, \phi) = 1$ then Equation 1 could be re-written as follows:

$$\gcd(e, \phi) \equiv 1 + s \cdot \phi \tag{2}$$

with s any positive integer. Do you see anything useful? We have just found the private exponent: exploiting the Extended Euclidean Algorithm we can compute both public and private exponent at once. But where is the private exponent? It's hiding from us behind t:

$$\gcd(e, \phi) \equiv 1 + s \cdot \phi \rightarrow t \cdot e \equiv 1 \pmod{\phi} \tag{3}$$

Fact 2. The average number of trials needed to find the public (and the private) exponent with the Extended Euclidean Algorithm is linear with the number of digits of the input parameter ($O(\log(x))$ where x is any integer representing the desidered key-length).

2.4 Encryption and Decryption Optimizations

So far we have seen "how much time" is required to compute a pair of RSA keys. What about the encryption and decryption complexity?

1. To encrypt a message x are required $e - 1$ integer multiplications.
2. Also, to decrypt a message y are required $d - 1$ integer multiplications.

Unfortunately, the exponents e and d are in general very large numbers. The exponents are typically chosen in the range of 1024–3072 bit or even larger. Straightforward exponentiation as shown above would thus require around 2^{1024} or more multiplications.
Is there a way to reduce the number of integers multiplication needed to encrypt/decrypt? Yes:

Fast Exponentiation. Let's consider a modular exponentiation $x^e \bmod n$. The exponent is an integer whose binary representation is the following:

$$e = 2^{k-1}e_{k-1} + 2^{k-2}e_{k-2} + \cdots + 2^1 e_1 + e_0$$

The modular exponentiation $x^e \bmod n$ can be rewritten as:

$$\begin{aligned}
x^e \bmod n &= x^{2^{k-1}e_{k-1}+2^{k-2}e_{k-2}+\cdots+2^1e_1+e_0} \bmod n \\
&= x^{2^{k-1}e_{k-1}} x^{2^{k-2}e_{k-2}} \ldots x^{2e_1} x^{e_0} \bmod n \\
&= (x^{2^{k-2}e_{k-1}} x^{2^{k-3}e_{k-2}} \ldots x^{e_1})^2 x^{e_0} \bmod n \\
&= ((x^{2^{k-3}e_{k-1}} x^{2^{k-4}e_{k-2}} \ldots)^2 x^{e_1})^2 x^{e_0} \bmod n \\
&= (((x^{2^2 e_{k-1}} x^{2e_{k-2}} x^{e_{k-3}})^2 \ldots)^2 x^{e_1})^2 x^{e_0} \bmod n \\
&= ((((x^{2e_{k-1}} x^{e_{k-2}})^2 x^{e_{k-3}})^2 \ldots)^2 x^{e_1})^2 x^{e_0} \bmod n \\
&= (((((x^{e_{k-1}})^2 x^{e_{k-2}})^2 x^{e_{k-3}})^2 \ldots)^2 x^{e_1})^2 x^{e_0} \bmod n
\end{aligned}$$

Algorithm 5 Fast Exponentiation: algorithm

```
y ← x
for i ← k − 1 to 0 do
    y ← y^2 mod n
    if e_i = 1 then
        y ← y · x mod n
    end if
end for
return y
```

What are performance gains given by fast exponentiation algorithm? If the binary representation of the exponent e requires k bits, the fast exponentiation algorithm performs (always) k squares (modular exponentiation) and a number of multiplications proportional to the *Hamming weight* of the exponent. To summarize:

1. k square operations.
2. $\#\{1s \text{ in } e\}$ integer multiplications. On average, $0.5k$ integer multiplications.
3. To keep small the size of the intermediate results, at each round a modular operation is performed.

Exploiting fast exponentiation allow us to decrease the number of integer operations needed to encrypt/decrypt. Since the number of multiplications are proportional to the number of 1 in the exponent, is there a way optimize more the encryption/decryption operations? Yes.

Consider the following integers:

1. $e = 2^1 + 1 = 3$
2. $e = 2^4 + 1 = 17$
3. $e = 2^{16} + 1$

Are they good public exponents? Yes, because all of them are primes, so obviously, they are co-prime with respect to any ϕ we would have computed. Also, they have a very interesting property: the number of 1 in their binary representation is very small. If we had used such exponents, we would have obtained the following number of integer operations:

1. $e = 3$: 2 squares and 1 integer multiplication
2. $e = 17$: 4 squares and 1 integer multiplication
3. $e = 2^{16} + 1$: 16 squares and 1 integer multiplication

Keep in mind that RSA is still secure; the public exponent doesn't need to be a very large and complex integer: it just has to satisfy the requirements specified by the key generation algorithm.

2.4.1 Fast Decryption with Chinese Remainder Theorem

We cannot choose a short private key without compromising the security for RSA. If we were to select keys d as short as we did in the case of encryption in the section above, an attacker could simply brute-force all possible numbers up to a given bit length. What one does instead is to apply a method which is based on the Chinese Remainder Theorem (CRT). Our goal is to perform the exponentiation $x = y^d \bmod n$ efficiently. First we note that the party who possesses the private key also knows the primes p and q. The basic idea of the CRT is that rather than doing arithmetic with one "long" modulus n, we do two individual exponentiations modulo the two "short" primes p and q.

Fast Decryption with CRT. Consider the modular exponentiation $y = x^d \bmod n$. Instead of computing the aforementioned modular exponentiation, let's switch to different domain (CRT domain):

$$y_p = x^{d_p} \bmod p \text{ where } d_p = d \bmod (p-1)$$
$$y_q = x^{d_q} \bmod q \text{ where } d_q = d \bmod (q-1)$$

Now we can compute $x^d \bmod n$ by applying an inverse transformation to the quantities y_p and y_q:

$$y = [c_p \cdot p] \cdot y_p + [c_q \cdot q] \cdot y_q \bmod n$$

where $c_p \equiv q^{-1} \bmod p$ and $c_q \equiv p^{-1} \bmod q$.

What about the complexity of the CRT-based RSA decryption algorithm?

1. Since both p and q are half the length of the modulus n, each operation is performed on operands with half the length compared to the plain decryption algorithm.
2. To compute y_p and y_q, on average, $1.5 \cdot \frac{k}{2} \cdot 2 = 1.5 \cdot k$ integer multiplications are required.
3. The average number of multiplications is still the same as in the original version of the decryption algorithm, but each multiplication involves operands whose sizes are half the size of the original ones.
4. Finally, since the bit complexity of multiplication is $O(k^2)$, we managed to obtain a speedup of 4x!

2.5 RSA in Practice

Plain RSA is **malleable**! What does it mean that an encryption scheme is malleable?

Malleability: An encryption scheme is considered *malleable* if an attacker can apply a *perturbation* to a ciphertext, resulting in a predictable transformation of the plaintext.

Let's see in practice how the malleability property can be exploited by a malicious guy:

1. Consider a ciphertext y sent by Alice to Bob, encrypted using Bob's public key K^B_{pub}. Alice intends to communicate to Bob that he owes her 100 dollars.
2. Oscar, who has hard feelings towards Bob, seeks to scam him. Oscar is aware that Alice is sending the amount of money that Bob has borrowed from her, but he doesn't know the exact sum. Therefore, he chooses s such that $\gcd(s, n) = 1$ and computes $\hat{y} = s^e \cdot y \bmod n$.
3. Oscar sends $\hat{y}$ to Bob.
4. Upon receiving $\hat{y}$, Bob decrypts it, obtaining $\hat{y}^d \bmod n = (s^e \cdot y)^d \bmod n = s \cdot x$.

The malleability property could be exploited also in a very different scenario. Let's consider an attacker A that somehow is capable of sending encrypted messages and force the receiver R to send back the plaintext. R replays back each time receives a request but he does not replay if he receives a particular ciphertext y. The attacker, who is willing to find what is the plaintext associated to the ciphertext y, mounts the following attack:

1. A choose s such that $\gcd(\mathsf{s}, \mathsf{n}) = 1$.
2. He computes $\hat{\mathsf{y}} = \mathsf{y} \cdot \mathsf{s}^{\mathsf{e}} \bmod \mathsf{n}$.
3. The attacker sends $\hat{\mathsf{y}}$ to R.
4. R replays back with $\hat{\mathsf{x}} = \hat{\mathsf{y}}^{\mathsf{d}} \bmod \mathsf{n} = (\mathsf{y} \cdot \mathsf{s}^{\mathsf{e}})^{\mathsf{d}} \bmod \mathsf{n} = \mathsf{x} \cdot \mathsf{s}$.
5. A, since $\gcd(\mathsf{s}, \mathsf{n}) = 1$, can compute the plaintext as $\mathsf{x} = \hat{\mathsf{x}} \cdot \mathsf{s}^{-1}$.

The attack can be contrasted by using padding: R returns x if (and only if) it has a structure coherent with padding.

2.5.1 RSA Attacks

There are a few attacks that can defeat plain RSA. Let's delve into the details.

Small plaintext attack Consider a scenario on which the plaintext is very small, *i.e.*, $\mathsf{x} << \mathsf{n}$. In such a scenario it could be possible that $\mathsf{x}^{\mathsf{e}} < \mathsf{n}$: if an attacker manages to intercept the ciphertext y, the latter could be decrypted, without knowing the private key, by computing $\sqrt[e]{y}$.

Low Exponent Attack Consider a scenario in which an attacker manages to obtain three (or more) different ciphertexts generated by encrypting the same plaintext x using three different public keys. Note that such a scenario is not necessarily extremely lucky. Assuming that each public key has the same exponent $\mathsf{e} = 3$, we have the following system of simultaneous congruences:

$$\begin{aligned} \mathsf{y}_1 &= \mathsf{x}^3 \bmod \mathsf{n}_1 \\ \mathsf{y}_2 &= \mathsf{x}^3 \bmod \mathsf{n}_2 \\ \mathsf{y}_3 &= \mathsf{x}^3 \bmod \mathsf{n}_3 \end{aligned}$$

We can rename x^3 as z, thus obtaining:

$$\begin{aligned} \mathsf{y}_1 &= \mathsf{z} \bmod \mathsf{n}_1 \\ \mathsf{y}_2 &= \mathsf{z} \bmod \mathsf{n}_2 \\ \mathsf{y}_3 &= \mathsf{z} \bmod \mathsf{n}_3 \end{aligned}$$

Now, if we are lucky enough to have the moduli n_i pairwise coprime, we can exploit the Chinese Remainder Theorem:

Fact 3. Chinese Remainder Theorem Consider the following system of congruences:

$$\begin{aligned} \mathsf{y}_1 &= \mathsf{z} \bmod \mathsf{n}_1 \\ \mathsf{y}_2 &= \mathsf{z} \bmod \mathsf{n}_2 \\ &\dots \\ \mathsf{y}_\mathsf{i} &= \mathsf{z} \bmod \mathsf{n}_\mathsf{m} \end{aligned}$$

Let $\gcd(\mathsf{n}_\mathsf{i}, \mathsf{n}_\mathsf{j}) = 1$ for each $\mathsf{i} \neq \mathsf{j}$. The system has a unique solution modulo $\mathsf{n} = \mathsf{n}_1\mathsf{n}_2 \dots \mathsf{n}_\mathsf{m}$.

Having in mind the CRT, we can compute the unique solution modulo $\mathsf{n} = \mathsf{n}_1\mathsf{n}_2\mathsf{n}_3$. Then, hoping that $\mathsf{z} = \mathsf{x}^3 << \mathsf{n}_1\mathsf{n}_2\mathsf{n}_3$, we can compute the plaintext as $\mathsf{x} = \sqrt[e]{z}$.

2.6 RSA Security

The best mathematical cryptanalytical method we know is factoring the modulus. An attacker, Oscar, knows the modulus n, the public key e and the ciphertext y. His goal is to compute the private key d which has the property that $\gcd(\mathsf{e}, \phi) = 1$. It seems that he could simply apply the extended Euclidean algorithm and compute d. However, he does not know the value of $\gcd(\mathsf{e}, \phi) = 1$. At this point factoring comes in: the best way to obtain this value is to decompose n into its primes p and q. If Oscar can do this, the attack succeeds.
In order to prevent this attack, the modulus must be sufficiently large. This is the sole reason why moduli of 1024 or more bit are needed.