Elements of Public Key Cryptography

# Keys and Key Exchange

The need to communicate in secret is as old as communication itself.[[1]](#footnote-1) The dominant approach has been for the sender to scramble the message before sending it to the intended receiver, who then unscrambles it on receipt. Without possession of the method by which to unscramble the message, an eavesdropper cannot decipher its contents. The method of unscrambling is generally referred to as the *key*.[[2]](#footnote-2)

More formally, for two or more parties to communicate securely over an *insecure* channel, each must possess a key that is used to encrypt and decrypt messages transmitted over that channel.[[3]](#footnote-3) Further, it is vital that this key be kept secret, so that the eavesdropper cannot use it to decrypt the messages he observes on the insecure channel. In other words, for the messages to be secure, the key must be known only to parties authorized to participate in the conversation.

The requirement that all parties authorized to participate in a secure conversation share a secret key poses a practical problem: How can the key be distributed securely to the authorized parties; that is, without it being stolen by the eavesdropper? For this you need a *secure* channel. Transmitting the key, unencrypted, over the insecure channel is not an option because it can be observed by an eavesdropper. Neither is encrypting the key, since you cannot decrypt it without first having the key (the classic chicken-and-egg problem).

One approach is to write keys down on pieces of paper, and hand-deliver them to each person one wishes to communicate with securely, but this doesn’t scale in the internet age.[[4]](#footnote-4)

# A Clever Solution

In 1976, two Stanford University cryptographers named Whitfield Diffie and Martin Hellman proposed an elegant solution to the key distribution problem in a seminal paper titled *New Directions in Cryptography*. This solution became, and remains to this day, the de facto standard for exchanging keys securely over insecure channels. It is commonly known as the *Diffie-Hellman key exchange protocol*.[[5]](#footnote-5)

The Diffie-Hellman key exchange protocol (hereafter shortened to DH) is a crucial component of public-key cryptography; [[6]](#footnote-6) namely, that of secure key exchange over insecure channels. Without it, secure communication on the internet—or any public channel for that matter—would be very difficult if not impossible in the internet age.

Informally, DH enables two or more parties to exchange information publicly over an insecure channel, and then to combine that information with private information to compute an identical, shared key that cannot easily be deduced by an eavesdropper observing the public information on the insecure channel.

The DH protocol can be implemented by means of any number of algorithms. Many examples in the literature cite the original implementation, which uses a multiplicative group of integers modulo *p*, where *p* is a prime number, to demonstrate DH. This is unfortunate because the mathematics of multiplicative groups modulo *p* are dense, thus obscuring an otherwise intuitive understanding of DH.

# Intuitive Diffie-Hellman

The graphic in *Figure 1* demonstrates DH using rudimentary multiplication, which should facilitate an intuitive grasp of DH. With that we can have a look at more complex implementations, and discuss why they are necessary in the real world.

Assume Alice wants to perform a secure key exchange with Bob over an insecure channel. Meanwhile, Eve observes all traffic passing between Alice and Bob, presumably for malicious purposes.[[7]](#footnote-7)



In steps 1 and 2, Alice selects a random integer (2) and transmits it to Bob. Let’s call this number the *generator*, because it will be used by Alice and Bob to generate another number; namely by multiplying it by a secret number she selects. Because the channel is insecure, Eve observes the value of the generator (2).

In steps 3, 4 and 5, Alice selects another random integer (3), multiplies it by the generator (2), and transmits the product of this multiplication (6) to Bob. Let’s call the random number that Alice selects her *private* key, and the product of its multiplication by the generator her *public* key (public because it is visible to anyone observing the insecure channel). As expected, Eve observes Alice’s public key (6). But Eve does not observe Alice’s private key, because Alice does not transmit it to Bob.

In steps 6, 7 and 8, Bob selects a random integer (4), multiplies it by the generator (2), and transmits the product (8) to Alice. These are Bob’s private and public keys, only the latter of which does he transmit to Alice. Eve observes Bob’s public key (8), but not his private key (4). Eve now knows the generator (2), Alice’s public key (6) and Bob’s public key (8).

The magic of DH appears in step 9. Alice multiplies Bob’s public key (8) by her private key (3). The product of this multiplication is 24. Similarly, Bob multiplies Alice’s public key (6) by his private key (4). The product of this multiplication is also 24. By using a combination of public and private information, Alice and Bob have agreed that the number 24 will be the shared key with which to encrypt and decrypt future messages between them.

Where does this leave Eve? Having only seen the value of the generator (2), Alice’s public key (6) and Bob’s public key (8), but neither Alice’s nor Bob’s private keys (3 and 4, respectively), Eve does not know by what factors Alice’s and Bob’s public keys were multiplied to arrive at the shared encryption key (24).[[8]](#footnote-8)

Of course, in this minimal implementation, Eve can deduce the private keys very easily; and with either Bob’s or Alice’s private key she can compute the shared key and decrypt the messages.

To do this, Eve must first know the algorithm used by Alice and Bob. This is a perfectly reasonable assumption given modern cryptography depends on the secrecy of *keys*, not *algorithms*.[[9]](#footnote-9) With knowledge of the algorithm, Eve simply divides Alice’s public key (6) by the generator (2) (both of which she observed) to derive Alice’s private key (3). Here Eve performs the *inverse* of the multiplication Alice used to generate her public key. Similarly, Eve can divide Bob’s public key (8) by the generator (2) to derive Bob’s private key (4). With either Alice’s or Bob’s private key, Eve can compute the shared key and use it to decrypt all messages exchanged between Alice and Bob.

Clearly this is a flawed DH implementation. But recall the object of this example was not to demonstrate an effective DH implementation, but rather to demonstrate how Alice and Bob can agree on an encryption key using a combination of public and private information. To make an effective DH implementation, we need to make the derivation of Alice or Bob’s private keys more difficult for Eve.

# DH Using Exponentiation

Let’s look at a second example that uses slightly more sophisticated math in an effort to thwart Eve.



Instead of multiplication to generate public keys, this time we use exponentiation (exponentiation in the figure is denoted by the ‘^’ symbol; as in 10 ^ 3 = 10 x 10 x 10 = 1000). Except for the calculations, all the steps are the same as in the previous example, so it’s not necessary to repeat them here. What is different this time is a) the algorithm used to compute the keys (exponentiation rather than multiplication) and b) the public parameters observed by Eve.

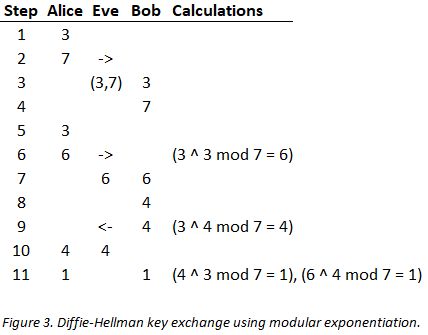
As before, Eve observes the generator (2), Alice’s public key (8) and Bob’s public key (16). With this information Eve must be able to compute the shared secret key (4096) to break the encryption. Because Eve knows the algorithm, she knows that Alice raised the generator (2) to the power of some exponent to compute her public key (8). To find that exponent, Eve must solve for *y* in the equation *x* ^ *y* = *z*, where *x* and *z* are known; i.e. what is the value of *y* in the equation 2 ^ *y* = 8?

Solving for *y* in this equation is known as taking the logarithm of *z*. Taking a logarithm is the inverse of exponentiation, in the same way that the division Eve used in the previous example to break the encryption is the inverse of multiplication. For small values of *z* (8 and 16 in the present example) solving for *y* is trivial; it simply requires trying consecutive exponents until the right answer is found. The requisite number of calculations to find Alice’s private key in the present example is three. For larger values of *z*, however, the complexity of Eve’s task grows proportionately with the size of *z* (whereas the complexity of the task in the implementation using multiplication is constant).

If *z* is large enough, solving for its logarithm becomes computationally infeasible, which is precisely the quality we are looking for in an effective DH implementation.

# DH Using Modular Exponentiation

Modular exponentiation brings us finally to the realm of real-world implementations. It is depicted graphically in *Figure 3*.



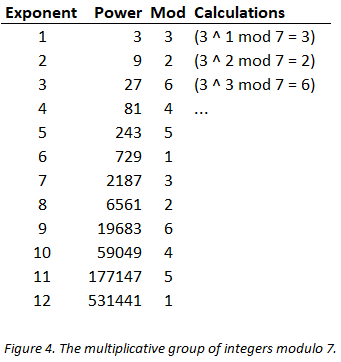
Basically, modular exponentiation is the same as exponentiation, but with an additional step. This additional step is called taking a modulus, which is done with the modulo operation.[[10]](#footnote-10) If you compare the graphic in this example with the one that uses exponentiation only, you will find the only difference is that in this one the modulo step is added to all the calculations.

The modulo operation requires an operand—namely the number by which to divide in order to find a remainder—and this explains why Alice transmits two numbers to Bob (steps 1 and 2) instead of one, like she did in the previous examples.

As before, Alice transmits the generator (3). But she also transmits the number by which to divide (7) in order to compute remainders. We call this second number the *modulus*. Note that the modulus too can be observed by Eve; i.e. it is a public parameter.[[11]](#footnote-11)

The modular exponentiation of real-world DH leads to a very interesting property of the generated keys. Compare the values of the public and encryption keys in this example (6, 4 and 1) to those of the previous, exponentiation-only example (8, 16 and 4096). In modular exponentiation the keys are smaller; notably, they are confined to the set of positive integers from one to the modulus minus one (in the present example {1,2,…,6}.[[12]](#footnote-12)

The graphic in *Figure 4* should clarify why this is so. The values in the *Power* column are the results of raising the generator (3) to the power of the values in the *Exponent* column. The values in the *Mod* column are the result of performing the modulo operation on the corresponding value in the *Power* column. Note that all our keys (6, 4 and 1) are present in the *Mod* column.



The keys we generate in this scheme exhibit two more interesting properties. First, if you read straight down the *Mod* column, you see that the sequence of remainders repeats after a while (3, 2, 6, 4, 5, 1, *3*, *2*, *6*, *4*, *5*, *1*). Second, each repeating sequence contains all integers in the range 1 to *p* - 1. These properties are described formally in the language of abstract algebra; specifically number theory and multiplicative groups modulo *p*, where *p* is a prime number.

To keep the discussion simple, it will suffice to keep the following rule in mind: Given a carefully chosen generator, together with a prime modulus, we can generate keys with the aforementioned properties, which is exactly what we want for an effective DH implementation.[[13]](#footnote-13)

Recall that in the previous example Eve had to solve the logarithm problem to break the encryption key. In the finite group of integers modulo *p*, this is called the *discrete* logarithm problem, or DLP. There is currently no known efficient algorithm for solving the DLP. In this fact lies the efficacy of DH.[[14]](#footnote-14)

DH solves the problem of secure key exchange, allowing two or more untrusted parties to agree on a shared key with which to encrypt and decrypt their private communications. But what of encryption and digital signatures; the other two promises of public-key cryptosystems introduced by Diffie and Hellman in their 1976 paper? The crypto community would have to wait another two years for the answer.

[Add discussion about attacks; e.g. man-in-the-middle.]

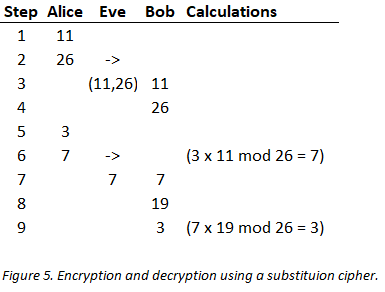
# RSA

In 1978, Ronald Rivest, Adi Shamir and Leonard Adelman published a paper called *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. In it they presented practical implementations for the elegant ideas introduced by Diffie and Hellman in their paper two years earlier. More than 40 years later, these implementations form the basis the most widely known (if not used) public-key cryptosystem in the world.[[15]](#footnote-15) This cryptosystem is known by the initials of the surnames of the paper’s authors, or *RSA*.

Before we delve too deeply into the guts of RSA, let’s start with a primitive example. Then we can work our way up to a real-world example having established a good conceptual model. We’ll start with encryption first, and then look at digital signatures.

# Substitution Cipher

*Figure 5* depicts a message exchange between Alice and Bob using a simple substitution cipher. Alice wants to send a private message to Bob over the insecure channel. In order to prevent Eve from reading the message, Alice encrypts it with the cipher prior to sending it to Bob. On receipt of the message, Bob decrypts it by inverting the encrypted message back to its original, unencrypted form.



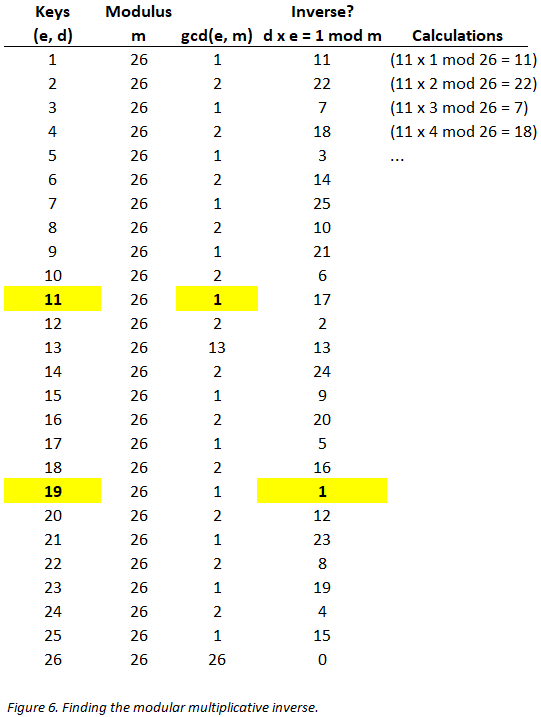
In steps 1 through 4, Alice selects an encryption key (11) and a modulus (26), and transmits both to Bob.[[16]](#footnote-16) Eve observes both values.

In step 5, Alice creates a message (3). (Since computers operate with numbers rather than letters, let the integer 3 here represent the letter *c*; and assume that this is the message Alice wants to send to Bob.)

In steps 6 and 7, Alice encrypts the *plaintext* (3) by multiplying it by her encryption key (11), and then takes the modulus of the result to arrive at the *ciphertext* (3 x 11 mod 26 = 7). Alice transmits the ciphertext to (7) to Bob, which Eve also observes.[[17]](#footnote-17)

In steps 8 and 9, Bob decrypts the ciphertext. He multiplies the ciphertext (7) by the *inverse* of Alice’s encryption key to compute a decryption key (19), and takes the modulus (26) to arrive back at the plaintext (3). But where does Bob get the decryption key (19) from?

The graphic in *Figure 6* gives us the answer.



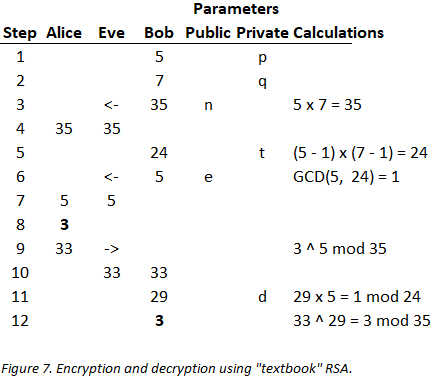
The *Keys (e, d)* column contains the set from which our encryption (*e*) and decryption (*d*) keys are chosen. To select *e*, Alice need only ensure that the greatest common divisor (*gcd*) between her private key *e* and the modulus *m* is equal to 1. This property guarantees that there exists a modular multiplicative inverse for *e*, because in the realm of modular multiplication, not all members of the set necessarily have this property. Alice selected 11 for her private key *e*, but she could have selected any value between 1 and 26, where *gcd(e, m) = 1* (e.g. 1, 3, 5, 7, 9…).

Now bob must compute the modular multiplicative inverse of *e* to use for the decryption key *d*. The modular multiplicative inverse of *e* is satisfied by the equation *d x e = 1 mod m*, where *d* is the decryption key, *e* is the encryption key and *m* is the modulus. Looking down the *Inverse? d x e = 1 mod m* column, we see that the only key that meets this criterion is 19, because 19 x 11 = 1 mod 26.

Now, any message in the range 1 to 26 (or *a* to *z*) that Alice encrypts with *e*, Bob can decrypt with *d*. And unless Eve knows *d*, she cannot break the encryption having only seen *e* and *m*. Of course, in this trivial example Eve can perform the same operation on *e* that Bob did to derive *d*, thereby defeating the encryption.

# Textbook RSA

The example in *Figure 7* gives us what is referred to in the literature as *textbook* RSA. In addition to the parameters being set to artificially small values, this version of RSA lacks more advanced features that would be present in real-world implementations.[[18]](#footnote-18) However, what we see in *Figure 7* is in essence RSA encryption.



The first thing to note is that there appears to be a lot going on here. And there is, but it’s not that complicated if we take it one step at a time.

Again, Alice wants to transmit a message to Bob (*c* again, which is encoded as 3 on our fictional computer). But in order for Alice to transmit an encrypted message to Bob, she first needs his public key.

\*\*\*

In the DH key exchange protocol, the objective was to make computing the inverse of the public information so computationally difficult as to render it infeasible for the attacker (Eve) to recover the private information from it. But with encryption we want to make computing the inverse of the public information (in this case the ciphertext) easy, but *only* for the intended recipient Bob.

This distinction can be summarized as follows: Whereas DH uses a *one-way* function to compute public information, RSA uses a *one-way trap-door* function. In the present example, that trap-door is the modular multiplicative inverse of 11 modulo 26, or 19. If Bob knows this trap-door information, he can compute the inverse of the ciphertext easily. But without it, Eve’s task becomes very difficult.

Recall that DH is based on a one-way function: given two integers *g* and *p*, it is easy to compute *g* ^ *x* mod *p*. But given *g* ^ *x* mod *p*, it is not easy to compute *x*. RSA uses a one-way function as well, but the function features a *trap door* not present in DH. RSA states that, given two integers *e* and *n*, it is easy to compute *m* ^ *e* mod *n*. But given *m* ^ *e* mod *n*, it is not easy to compute *m*; that is *unless* you have the trap-door information.[[19]](#footnote-19) The factorization of the modulus *n* is the trap-door information.

Whereas the efficacy of DH lies in the difficulty of solving the discrete log problem (DLP), in RSA it lies in the difficulty of solving the *integer factorization problem*. Specifically, if you know the factorization of *n*, you can easily invert *m* ^ *e* mod *n*.

If you are paying attention you might have noticed an apparent contradiction in the phrase *factorization of n* from previous paragraphs. If *n* is prime, then the only factors of *n* are 1 and *n*. Therefore, by definition, if you know *n* you know the factorization of *n*. But unlike DH, wherein the modulus *p* is prime, in RSA the modulus *n* is not prime. Rather, it is *semiprime*, which means simply that it is the product of two primes.[[20]](#footnote-20) Let’s call these prime factors of *p* and *q*, so that *p* x *q* = *n*. The modulus *n* is part of the public key, but the factors of *n*—*p* and *q*—must be kept private.

But *n* is only part of the public key. We also need the encryption exponent *e*.

[Explain how plaintext messages boil down to integers.]

[Add section on elliptic curves.]

1. King to general, via courier (who might be an enemy spy): *Attack at dawn*. [↑](#footnote-ref-1)
2. The key can be a mechanical device, a number, a puzzle; anything known to both sender and receiver that enables the sender to encipher, and the receiver to decipher, a message. [↑](#footnote-ref-2)
3. A modern example of such a channel is the public internet. [↑](#footnote-ref-3)
4. The number of keys required for a group of *n* participants to communicate securely is found by following formula: *n*(*n*-1)/2, where *n* is the number of participants. For a group of 10 the number of keys is 10(10-1)/2, or 45; for 100 the number is 4,950. As the number of participants increases, the number of keys increases quadratically. [↑](#footnote-ref-4)
5. Ralph Merkle’s name deserves mention here, for it is from his ideas that Diffie and Hellman’s work emerged (see *Merkle’s Puzzles*). [↑](#footnote-ref-5)
6. Also known as *asymmetric*-key cryptography, public-key cryptography is based on the principle that different (though mathematically related) keys can be used for (among other things) encryption and decryption; whereas *symmetric*-key cryptography relies on identical keys for encryption and decryption. Symmetric-key protocols are easier to implement. Public-key cryptography permits more sophisticated cryptographic techniques, such as the exchange of symmetric keys over insecure channels. [↑](#footnote-ref-6)
7. The examples in this paper feature the cast of fictional characters ubiquitous in the literature: Alice, Bob and Eve. [↑](#footnote-ref-7)
8. It is useful to point out here that Alice and Bob could have selected any private key (besides 3 and 4, respectively) and the result in step 9 would have been the same; i.e. equivalent secret keys. [↑](#footnote-ref-8)
9. This fact is formalized in Kerckhoffs’s principle, proposed by Auguste Kerckhoffs in 1883, which turned several millennia of cryptographic orthodoxy on its head. Kerckhoffs stated that, “A cryptosystem should be secure even if everything about the system, except the key, is public knowledge”. Prior to this, the efficacy of a cipher was believed to be based on the secrecy of its algorithm. One important implication of Kerckhoffs’s principle is that a cipher that is widely-known will invite attacks, and that this is desirable because very smart people know they will become famous if they find a way to defeat it. It should not be surprising that the best cryptosystems in the world are those that have defied successful attacks over a long period of time. [↑](#footnote-ref-9)
10. A modulo operation simply finds the remainder after division of two numbers. For example, 7 mod 3 = 1, because 7 / 3 = 2 with a remainder of 1; that is, 3 goes into 7 two times (to make 6) leaving 1 left over. [↑](#footnote-ref-10)
11. Public parameters in public-key cryptography are often referred to in the literature as *domain* parameters. [↑](#footnote-ref-11)
12. Exponentiation of a generator *g* modulo *p*, where *g* is greater than 1 and *p* is prime, guarantees results will be within the range 1 to *p* - 1. [↑](#footnote-ref-12)
13. A carefully chosen generator is one which generates the entire group of integers in the range 1 to *p* - 1, where *p* is the prime modulus. Any generator that fulfills this property is called a *primitive root*. The rules of multiplicative groups modulo *p* guarantee that at least one integer in the group 1 to *p* - 1 is a primitive root. In the current example, 3 is a primitive root of the group of integers modulo 7. In real-world DH, the modulus should be a very large, randomly-chosen prime number. We use 7 for the modulus here to demonstrate the concepts. [↑](#footnote-ref-13)
14. It is possible there is some other as yet unknown, or at least unpublished, way to break DH than finding an efficient algorithm for the discrete log problem. Until that fact is proven, a distinction is made between the DH problem and the DLP. [↑](#footnote-ref-14)
15. RSA, as we will see, can be used for secure key exchange as well. [↑](#footnote-ref-15)
16. The values Alice selects for her key and modulus are not arbitrary. For encryption to work, the modulus must be at least as large as the character set used in the message. From the modulus Eve selects (26), you can reasonably guess that this character set consists of the letters of the Latin alphabet. As for the key (11), it must be *coprime*, or *relatively prime*, with the modulus (26), which simply means that the biggest integer that divides both the key and the modulus is 1. [↑](#footnote-ref-16)
17. *Plaintext* and *ciphertext* are the terms of art for unencrypted and encrypted messages, respectively. [↑](#footnote-ref-17)
18. Cryptographic padding, e.g. to mitigate the threat of side-channel attacks. [↑](#footnote-ref-18)
19. In the RSA equation *m* ^ *e* mod *n*, *m* stands for *message* (as in the plaintext to encrypt), and *e* stands for the *encryption* exponent. [↑](#footnote-ref-19)
20. A number that is the product of two prime numbers is said to be semiprime, and is divisible by 1, the two primes multiplied to produce it (*p* and *q*), and itself (*n*). [↑](#footnote-ref-20)