The Elements of Public-Key Cryptography

# Keys and the Key Distribution Problem

The need to communicate in secret is as old as communication itself.[[1]](#footnote-1) The time-honored approach is to scramble a message before sending it, which the receiver unscrambles on receipt. Without knowledge of the method by which the message is scrambled, unauthorized parties who intercept the message cannot unscramble it. The mechanism for scrambling, and 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 can be used to encipher and decipher messages transmitted over the insecure channel.[[3]](#footnote-3) This key must be known only to the parties authorized to send and receive the messages; otherwise an eavesdropper with knowledge of the key will be able to read the messages.

The requirement that only the parties authorized to participate in a secure conversation share a secret key poses a problem: How is the key distributed to the authorized parties securely; that is, without it being stolen by an unauthorized party? For this you need a *secure* channel. Transmitting the key over an insecure channel is not an option, because if the channel is insecure the key can be stolen. Neither is enciphering the key before transmitting it, since you cannot decipher it without first having the key.[[4]](#footnote-4)

The most obvious and effective solution is to hand-deliver the key in advance to the party you wish to communicate with. But this is also the least efficient solution, and not at all practical in the internet age.[[5]](#footnote-5)

# A Clever Solution

In 1976, two Stanford University cryptographers proposed an elegant solution to the key distribution problem in a groundbreaking 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*.[[6]](#footnote-6)

The Diffie-Hellman (DH) key exchange protocol is a foundational element of *public-key* cryptography;[[7]](#footnote-7) namely, that of secure key exchange over insecure channels. Informally, DH enables two (or more) parties to exchange public information over an insecure channel, and then combine it with private information to compute an identical, shared key with which to encrypt and decrypt messages sent to each other subsequently on the insecure channel. Because private information is used on either side of the channel to generate the shared key, the key cannot be observed by an eavesdropper listening on the insecure channel.

DH can be implemented by means of any number of algorithms. Most examples in the literature cite the original implementation, which uses a *multiplicative group of integers modulo a prime number*, to demonstrate DH. This is unfortunate, because the mathematics of multiplicative groups modulo a prime are complex, and thus hinder a conceptual understanding of DH.

# Simplified Diffie-Hellman

The graphic in *Figure 1* demonstrates DH using rudimentary multiplication. This example depicts a simplified implementation of DH. This should prove useful in comprehending a more complex, real-world implementation described in subsequent sections of this paper.

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.[[8]](#footnote-8)



In steps 1 and 2, Alice selects a random integer (2) and transmits it to Bob.[[9]](#footnote-9) 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 private number they will each select. Because the channel is insecure, Eve observes the value of the generator (2).

In steps 3, 4 and 5, Alice selects a random integer (3), multiplies it by the generator (2), and transmits the product of the 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 can be observed by Eve on the insecure channel). As expected, Eve observes Alice’s public key (6). But Eve does not observe Alice’s private key (3), because Alice never transmits her private key 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, respectively. Eve observes Bob’s public key (8). Eve now knows the generator (2), Alice’s public key (6) and Bob’s public key (8), but neither Alice nor Bob’s private keys (3 and 4).

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 in this way, Alice and Bob have agreed that the number 24 will be the shared key with which to encrypt and decrypt 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), Eve does not know by what factors Alice’s and Bob’s public keys were multiplied to compute the shared encryption key (24).[[10]](#footnote-10)

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

But in addition to correctly guessing a private key, Eve must also know the *algorithm* used by Alice and Bob to compute the shared key; i.e. multiplication. We must assume that Eve knows the algorithm. This happens to be a perfectly reasonable assumption, because the efficacy of modern cryptography relies wholly on the secrecy of *keys*, not *algorithms*.[[11]](#footnote-11) With knowledge of the algorithm, Eve simply divides Alice’s public key (6) by the generator (2), both of which she observed on the insecure channel, to derive Alice’s private key (3).

Here Eve has performed the *inverse* of the multiplication Alice used to generate her public key.[[12]](#footnote-12) Similarly, Eve can divide Bob’s public key (8) by the generator (2) to derive Bob’s private key (4). The important point is that with either Alice or Bob’s private key, Eve can compute the shared key and use it to decrypt all messages exchanged between Alice and Bob.

It should not be a surprise that a DH implementation using multiplication to generate shared keys is badly flawed. But the object of this example is not to demonstrate an effective DH implementation, but to demonstrate how Alice and Bob can generate an identical, secret key using a combination of public and private information.

For an effective DH implementation, we need to make the task of guessing Alice and Bob’s private keys more difficult for Eve.

# DH With Exponentiation

Let’s look at a second example using slightly more sophisticated math to make Eve’s task more difficult.



Instead of using multiplication to generate public keys, this time Alice and Bob 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 versus 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.[[13]](#footnote-13)

Solving for *y* in this equation is known as taking the logarithm of *z*. Taking a logarithm is the inverse of exponentiation, just as division is the inverse of the multiplication Eve used in the previous example to break the encryption. 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 tries 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*. Contrast this with the implementation using multiplication, where the complexity is constant.

As the value of *z* increases, solving for its logarithm requires more tries, and if *z* is large enough it becomes computationally infeasible. With this we get closer to an effective DH implementation, but we’re not quite there yet.

# DH With Modular Exponentiation

Modular exponentiation brings us finally to the realm of real-world DH.



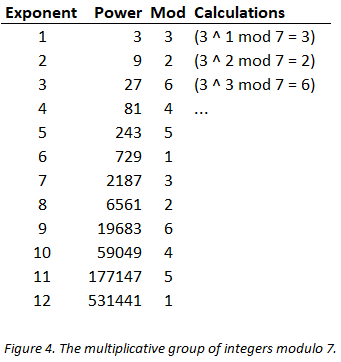
Modular exponentiation is the same as exponentiation, but with an additional step. This additional step is called *taking a modulus*. This is done with the modulo operation.[[14]](#footnote-14) If you compare the graphic in this example with the one that uses exponentiation only (*Figure 2*), you will 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. We call this operand the *divisor*.

As before, Alice transmits the generator (3). But she also transmits the divisor (7) that will be used to compute remainders. Note that the divisor too can be observed by Eve; i.e. it is a public parameter.[[15]](#footnote-15)

The modular exponentiation of real-world DH leads to a very interesting property of the generated keys. Compare the values of the public and shared keys in this example (6, 4 and 1) to those of the example in *Figure 2* (8, 16 and 4096). With modular exponentiation the keys are smaller; notably, they are confined to the set of positive integers starting at one, and ending at one less than the divisor.[[16]](#footnote-16)

The graphic in *Figure 4* should clarify why this is so. The values in the *Power* column are the result of raising the generator (3) to the power of the values in the *Exponent* column, and 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 exhibit two more interesting properties: First, if you read straight down the *Mod* column, you find 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 every integer in the set 1 to *p* - 1.[[17]](#footnote-17)

To keep things simple, it will suffice to keep the following rule in mind: Given a carefully chosen generator, and a prime divisor, we can generate keys with the aforementioned properties. And it is these properties that are required for an effective DH implementation.[[18]](#footnote-18)

Recall from the previous example that Eve had to solve for the logarithm of *z* in the equation *x* ^ *y* = *z* to find the shared key. In the finite group of integers modulo *p*, this is qualified a bit; it is called the *discrete* logarithm problem, which is often abbreviated in the literature to DLP.

There is no *known*, efficient algorithm for solving the DLP. Consecutive integers must be tried in the equation *x* ^ *y* = *z* mod *p*, where the values of *x*, *z* and *p* are known, until the correct *y* is found. If z is large enough, the number of tries becomes too computationally expensive to be feasible for an attacker. The efficacy of DH is based, at least in part, on the difficulty of solving the DLP.[[19]](#footnote-19)

# Toward a More Complete Cryptosystem

Whereas DH solves the problem of secure key exchange, what of encryption itself; the principal use case for cryptography? Besides secure key exchange, encryption was one of the key elements of the public-key cryptosystem described, but not solved, by Diffie and Hellman in their 1976 paper.[[20]](#footnote-20) For the answer to that question the crypto community would have to wait another two years.

# Public-Key Encryption

The public-key cryptosystem conceived by Diffie and Hellman in *New Directions* consists of three distinct but interrelated elements: secure key exchange, encryption and digital signatures. But the paper only presented an actual implementation for secure key exchange.

While working together at MIT in 1978, the cryptographers Ronald Rivest, Adi Shamir and Leonard Adelman published a paper called *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. In it they picked up where Diffie and Hellman left off, presenting practical implementations for public-key encryption and digital signatures.

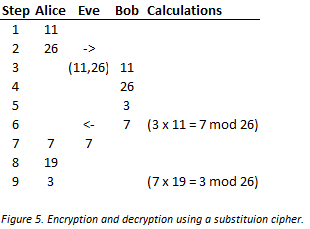
To this day, the contributions of Rivest, Shamir and Adelman form the basis the most widely used and battle tested public-key cryptosystem in the world, known simply by the initials of the surnames of its authors’, *RSA*.

To understand RSA encryption, we’ll use the same approach we took with DH; by starting with a primitive example before advancing to a more realistic one.

# Substitution Cipher

*Figure 5* depicts a message exchange between Alice and Bob using a simple substitution cipher. This cipher employs the now-familiar mathematics of modular exponentiation.

Here, Bob wants to transmit a private message to Alice over an insecure channel. In order to prevent Eve from reading the message, Bob encrypts the message prior to sending it to Alice. On receipt, Alice decrypts the message by inverting it to its original, unencrypted form.

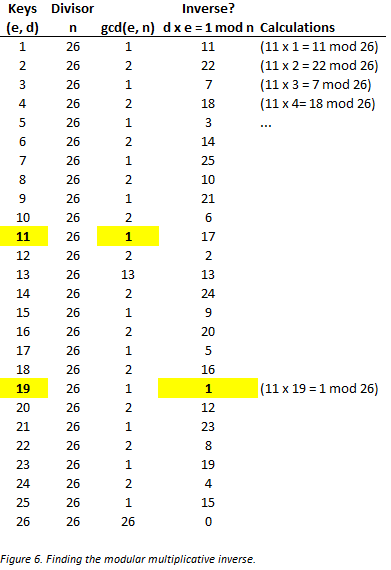


In steps 1, 2, 3 and 4, Alice selects a public encryption key (11) and a divisor (26), and transmits both to Bob.[[21]](#footnote-21) Eve observes the values of both the public encryption key and the divisor. We call the encryption key *public* because it can be observed by an eavesdropper.

In steps 5, 6 and 7, Bob creates a plaintext message (3),[[22]](#footnote-22) multiplies it by Alice’s public encryption key (11), and takes the modulus of the product to compute the ciphertext (3 x 11 = 7 mod 26).[[23]](#footnote-23) Bob transmits the ciphertext (7) to Alice, which Eve also observes.

In steps 8 and 9, Alice decrypts the ciphertext. She multiplies the ciphertext (7) she received from Bob by her private decryption key (19), and takes the modulus of the product to arrive back at the plaintext (7 x 19 = 3 mod 26). We call Alice’s decryption key *private* because it is never transmitted to Bob, and therefore cannot be observed by an eavesdropper.

This is brilliant, but where does Alice’s private decryption key (19) come from? The graphic in *Figure 6* gives us the answer.



The *Keys (e, d)* column contains the set from which Alice selects her public encryption (*e*) and private decryption (*d*) keys. To select her encryption key *e*, Alice need only ensure that the *greatest common divisor* (*gcd*) of her encryption key *e* and the divisor *n* is 1. This property of *e* ensures that ithas a *modular multiplicative inverse* in the set of integers modulo 26.[[24]](#footnote-24) Alice selected 11 as her public key *e*, but she could have selected any value between 1 and 26 where *gcd(e, n) = 1* (e.g. 1, 3, 5, 7, 9…).

Now that Alice has selected a suitable *e*, she must find its modular multiplicative inverse to use as her private decryption key *d*. The modular multiplicative inverse of *e* is satisfied by the equation *d x e = 1 mod n*, where *d* is the decryption key, *e* is the encryption key and *n* is the divisor. Looking down the *Inverse? d x e = 1 mod n* column, we find the only value that meets this criterion is 19 (19 x 11 = 1 mod 26).[[25]](#footnote-25)

Now, any message in the range 1 to 26 (or *a* to *z*) that Bob encrypts with Alice’s public key *e*, Alice can decrypt with her private key *d*. And having only seen *e* and *n*, and not *d*, Eve cannot break the encryption.

Of course, in this primitive substitution cipher, Eve could very quickly do an exhaustive search of the group of integers modulo 26 to find the inverse of *e*, thereby defeating the encryption.

# Textbook RSA

*Figure 7* depicts what is referred to in the literature as *textbook* RSA. For our purposes, textbook RSA is close enough to a real-world implementation that we’ll conclude our discussion of RSA encryption with it. As with the previous examples, the parameters are set to artificially small values to keep the math manageable.



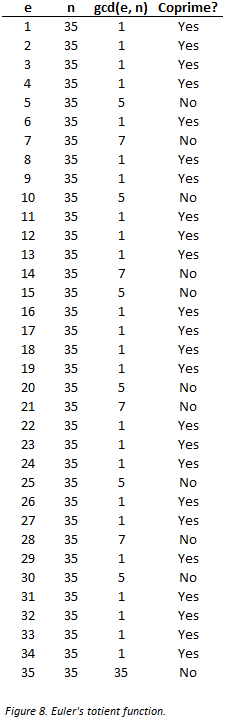
Again, Bob wishes to transmit a message to Alice (the letter *c* again, which we encode numerically as 3). In steps 1, 2, 3 and 4, Alice selects two random integers, *p* and *q* (5 and 7), multiplies them to produce a divisor *n* (35), and transmits the divisor to Bob. Eve observes the value of *n* (35), but not its factors *p* and *q*.

Note that in contrast to the previous example, where Alice selected 26 as a divisor (to correspond with the number of letters in the alphabet), she computes the divisor *n* this time by multiplying two factors, *p* and *q*. Alice’s only requirement for a suitable *n* is that its factors *p* and *q* both be prime numbers.[[26]](#footnote-26) In the real world these factors would be enormous; somewhere on the order of 600 decimal digits in length.

In step 5, Alice applies *Euler’s totient function* to the divisor *n*; let’s call the result of this function *t*. Given some positive integer *n*, Euler’s totient function tells us the number of positive integers less than *n* with which *n* is coprime. For any prime number *p*, the formula is simple: it is *p* - 1. It should be clear why this is so. If *p* is prime, we know that the only integers that divide *p* are 1 and *p* itself, so every integer in the set 1 to *p* - 1 must be coprime with *p*. Recall that for two integers to be coprime, the biggest integer that divides both—or their *greatest common divisor*—is 1.

But Alice needs to apply the totient function to the *semiprime* divisor *n*, which is the product of the two primes *p* and *q* (see footnote 26 for the definition of *semiprime*). The formula therefore becomes the product of the totients of *n*’s factors *p* and *q*;i.e. (*p* - 1)(*q* - 1). Plugging in the values from the example we get (5 - 1)(7 - 1) = 24.

In plain English, Euler’s totient function tells us that there are 24 integers in the set 1 to 35 that are coprime with 35. *Figure 8* depicts this graphically. Indeed, the number of *Yes* entries in the *Coprime?* column confirms the number is 24.



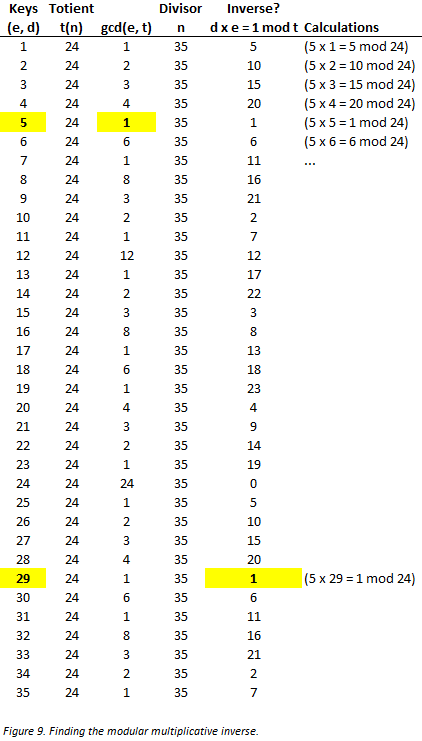
As with the values of *p* and *q*, Alice must keep *t* secret.

In steps 6 and 7, Alice computes a public key *e* (5) and transmits it to Bob. Bob will use this key to encrypt the plaintext (3) he transmits to Alice. For *e*, Alice can select any value in the set 1 to *t* where the *gcd* is 1. In the present example, she selects 5 (but 7, 11, 13, 17, 19 or 23 would do).[[27]](#footnote-27) Eve observes the value of *e*.

In steps 8, 9 and 10, Bob encrypts the plaintext (3) and transmits it to Alice. Using the public key *e* (5) he received from Alice, he raises his plaintext to the power of *e*, takes the modulus and transmits the resulting ciphertext (33) to Alice (3 ^ 5 mod 35 = 33). Eve observes the ciphertext 33.

In steps 11 and 12, Alice receives and decrypts the ciphertext (33). But in order to decrypt the ciphertext, she must invert it to plaintext. To do this she needs a decryption key *d*. To find *d*, Alice computes the modular multiplicative inverse of the public key *e* (5) in the set of integers modulo *t* (24). The modular multiplicative inverse of 5 in the set of integers modulo 24 is 29; therefore, *d* = 29. Alice raises the ciphertext (33) to the power of *d* (29), and mods the result with *n* (35) to retrieve the plaintext (33 ^ 29 mod 35 = 3).

To see why the modular multiplicative inverse of 5 in the group of integers modulo 24 is 29, a look at *Figure 9* should help.



The highlighted values in the *Keys (e, d)* column are Alice’s public encryption and private decryption keys, *e* and *d*, respectively. To select her encryption key *e*, she finds the first integer greater than one whose *gcd* with the totient *t* is 1, or 5 (although she could have selected the equally valid integers 7, 11, 13 and so on, as keys). To select her decryption key *d*, she finds the modular multiplicative inverse of her encryption key *e* (5) in the group of integers modulo *t* (24). The only value that fits the bill is 29, because that is the first *d* in the equation *d x e = r mod t* that leaves a remainder *r* of 1[[28]](#footnote-28), and recall that this is the equation that must be satisfied to find an inverse.

The result is that Bob has successfully transmitted an encrypted message to Alice, which Alice has successfully decrypted. But where does this leave Eve? Eve has observed the values of the divisor *n*, the public key *e* and the ciphertext. But she has not observed the prime factors *p* and *q*, and therefore cannot efficiently compute the totient *t*. Without *t*, Eve cannot derive the private key *d* and break the encryption. The reason Eve cannot efficiently compute the totient *t* is because there is no known, *efficient* way to factor integers. In the present example, this means that given the divisor *n* (35), there is no efficient way to derive its factors *p* (5) and *q* (7).[[29]](#footnote-29) This is known as the *integer factorization problem*.

Whereas the difficulty for Eve of solving the *discrete log problem* (DLP) is the bedrock on which secure key exchange lies, the difficulty of integer factorization serves the same purpose with public-key encryption.

# RSA Encryption as an Alternative to DH

It could be argued that RSA encryption renders DH key exchange obsolete, because RSA encryption can be used as an alternative to DH for secure key exchange.

To see why this is so, it first helps to realize that, although we refer to DH as a key *exchange* protocol, it is in fact more precisely a key *agreement* protocol. That is, the final product of DH—a symmetric encryption/decryption key to be used in a subsequent, private message exchange between two parties—is never actually exchanged, but rather computed by both parties independently. Simply, the encryption key in DH is agreed on, not exchanged.

But with RSA public-key encryption, we now have the opportunity to do a secure key *exchange*. Imagine a scenario whereby Alice randomly selects a shared encryption/decryption key, encrypts it with Bob’s public encryption key, and transmits the shared key to Bob. Now Alice and Bob both have an identical, shared key, but one that Alice randomly selected and *exchanged* with Bob, rather than one that was computed by Alice and Bob independently. By using RSA to exchange a key, rather than DH to agree on one, Alice and Bob have effectively achieved the same result.

The superiority of one method over the other is a matter for debate, and beyond the scope of this paper. The distinction between key agreement and key exchange is nonetheless useful to be aware of.

# Digital Signatures

Digital signatures are the third and final component of the public-key cryptosystem conceived by DH and implemented by RSA. Digital signatures serve two purposes in digital communication: *authentication* and *non-repudiation*. Whereas both properties are important in a secure message exchange, message authentication does not require the services of a public-key cryptosystem.[[30]](#footnote-30) It is non-repudiation that makes digital signatures so powerful. And it is the public-key cryptosystem of RSA that makes digital signatures possible.

1. King to general, via trusted courier: *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. The quintessential example of such an insecure channel is the public internet. [↑](#footnote-ref-3)
4. The classic *chicken-and-egg* problem. [↑](#footnote-ref-4)
5. Accepting that each *n*-party communication requires a separate key, the number of key exchanges 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 key exchanges is 10(10-1)/2, or 45; for 100 the number is 4,950, and so on. As the number of participants increases, the number of key exchanges increases quadratically. [↑](#footnote-ref-5)
6. Although Whitfield Diffie and Martin Hellman co-authored the paper, and their names are attributed to the protocol, Ralph Merkle’s name deserves mention because it is on Merkle’s ideas that Diffie-Hellman is based (see *Merkle’s Puzzles*). [↑](#footnote-ref-6)
7. Also known as *asymmetric*-key cryptography, public-key cryptography is based on the principle that different, though mathematically related, keys can be used for encryption and decryption; whereas traditional encryption relies on identical, or *symmetric*, keys. [↑](#footnote-ref-7)
8. The examples in this paper feature the cast of fictional characters ubiquitous in the literature: Alice, Bob and Eve. [↑](#footnote-ref-8)
9. Since computers operate with numbers, we use integers in this and all subsequent examples to represent messages and keys. [↑](#footnote-ref-9)
10. It is useful to point out here that Alice and Bob could have selected any private key (besides 3 and 4, respectively) and the effect in step 9 would have been the same: i.e. identical secret keys. [↑](#footnote-ref-10)
11. 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 whose algorithm 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-11)
12. Division is the inverse of multiplication, just as subtraction is the inverse of addition. [↑](#footnote-ref-12)
13. In the present example, what is the value of *y* in the equation 2 ^ *y* = 8? [↑](#footnote-ref-13)
14. A modulo operation simply finds the remainder after division of two numbers. For example, 7 mod 3 = 1, because 7 divided by 3 equals 2, leaving 1 left over. [↑](#footnote-ref-14)
15. Public parameters in public-key cryptography are often also referred to in the literature as *domain* parameters. [↑](#footnote-ref-15)
16. Exponentiation of a generator *g* modulo *p*, where *g* is greater than 1 and *p* is a prime number, guarantees result will be within the set 1 to *p* - 1; in the present example, where *g* = 3 and *p* = 7, this set contains the integers 1, 2, 3, 4, 5, and 6. [↑](#footnote-ref-16)
17. 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. [↑](#footnote-ref-17)
18. 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 an artificially small value (7) for the divisor in the example to demonstrate the concepts. [↑](#footnote-ref-18)
19. It is possible there is some other, as yet unknown (or at least unpublished), way to break DH; that is, besides solving the DLP efficiently. Until or unless such a method is found, no distinction is made between the DLP and the so-called *Diffie-Hellman* *problem*. [↑](#footnote-ref-19)
20. There was a third element, *digital signatures*, which would enable the sender of a message to prove both that it originated from the sender, and that its contents were unaltered. [↑](#footnote-ref-20)
21. The values Alice selects for her encryption key and divisor are not arbitrary. For encryption to work, the divisor must be at least as large as the character set used in the message. From the divisor Eve selects (26), let’s assume this character set consists of the lowercase letters of the Latin alphabet, *a* to *z*. As for the public key (11), the only requirement is that its value be *coprime*, or *relatively prime*, with the selected divisor (26). For two numbers to be coprime, the biggest integer that divides both evenly must be 1. [↑](#footnote-ref-21)
22. Since computers operate on numbers and not letters, pretend the integer 3 represent the letter *c* (because *c* is the 3rd letter of the alphabet). [↑](#footnote-ref-22)
23. *Plaintext* and *ciphertext* are the terms of art for unencrypted and encrypted messages, respectively. [↑](#footnote-ref-23)
24. The term *modular multiplicative inverse* sounds scary, but it is to modular exponentiation what division is to multiplication, or logarithm is to exponentiation. Not all members of a set of integers modulo *n*, formally known as a *group*, contain a modular multiplicative inverse; and only an encryption key *e* that has such an inverse will work. [↑](#footnote-ref-24)
25. Efficient algorithms exist for finding the greatest common divisor between two integers, and for finding the modular multiplicative inverse of an integer in a group of integers. These are, respectively, the Euclidean algorithm and the *extended* Euclidean algorithm. This is important because, if there were not such algorithms, the computational performance of RSA would be too slow for it to have been widely adopted. [↑](#footnote-ref-25)
26. The product of the multiplication of any two prime numbers is said to be *semiprime*, because the only numbers that can divide it evenly are 1, the two primes multiplied to produce it (its *factors*), and the product itself; in the present example these numbers are 1, 5, 7 and 35. [↑](#footnote-ref-26)
27. Since the value of *e* will used as an exponent in the encryption procedure, it should generally be kept as small as possible to maximize computational performance. All else equal, a small encryption key will not compromise the security of RSA encryption even in a real-world implementation. [↑](#footnote-ref-27)
28. Actually 5 works, too, but using the same *e* and *d* would be a silly choice. [↑](#footnote-ref-28)
29. Given a small, semiprime divisor *n*—e.g. 35 in the present example—finding its factors is trivial. For very large values of *n*, however, factorization is difficult. As always, small values are used in the examples to keep the math simple. [↑](#footnote-ref-29)
30. In fact, message authentication had been solved long before the advent of a public-key cryptosystem. [↑](#footnote-ref-30)