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] 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 it cannot unscramble the message. The method of scrambling, and unscrambling, is generally referred to as the *key*.[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] This key must be possessed only by the parties authorized to send and receive the messages; otherwise an eavesdropper with possession of the key will be able to read them.

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]

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]

A Clever Solution

In 1976, two Stanford University cryptographers proposed an elegant solution to the key distribution problem in a groundbreaking academic article titled *New Directions in Cryptography*.[6] 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*, or DH for short.[7]

---

[1] King to general, via trusted courier: *Attack at dawn!*

[2] The key can be a mechanical device, a number, a puzzle; anything possessed by both sender and receiver that enables the sender to encipher, and the receiver to decipher, a message.

[3] The quintessential example of such an insecure channel is the public internet.

[4] The classic *chicken-or-egg* problem.

[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.

[6] Diffie, Whitfield; Hellman, Martin E. (November 1976). *IEEE Transactions on Information Theory*. **22** (6): 644-654.

[7] Although Diffie and 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 DH is based (see *Merkle's Puzzles*).

DH is a foundational element of *public-key* cryptography;[8] namely, that of secure key exchange over insecure channels. 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 on the insecure channel. Because private information is used on either side of the channel to generate the shared key, the shared key cannot be observed by an eavesdropper.

DH can be implemented by means of a 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 the simplest of algorithms: rudimentary multiplication. This should facilitate the understanding of a more complex, real-world implementation presented in a subsequent example.

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.[9]

Step	Alice	Eve	Bob	Calculations
1	2	->		
2		2	2	
3	3			
4	6	->		(3 x 2 = 6)
5		6	6	
6			4	
7		<-	8	(4 x 2 = 8)
8	8	8		
9	24		24	(8 x 3 = 24), (6 x 4 = 24)

Figure 1. Diffie-Hellman key exchange using multiplication.

In steps 1 and 2, Alice selects a random integer (2) and transmits it to Bob.[10] Let's call this number the *generator*, because it will be used by Alice and Bob to generate another number; namely, by multiplying the generator by a private number each will select independently. 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). 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.

---

[8] 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.

[9] The examples in this paper feature the cast of fictional characters ubiquitous in the literature: Alice, Bob and Eve.

[10] Since computers operate with numbers, we use integers in this and all subsequent examples to represent messages and keys.

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. 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 she does not know Alice or 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 clever way, Alice and Bob have agreed that the number 24 will be the shared key with which to encrypt and decrypt messages sent 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).[11]

Of course, in this highly simplified 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 guessing a private key, Eve must also know the *algorithm* used by Alice and Bob to compute the shared key (i.e., *multiplication* of the generator by a private key).

Cryptographers always assume the algorithm is known by an attacker, and this is perfectly reasonable given that the efficacy of modern cryptography relies wholly on the secrecy of *keys*, not *algorithms*.[12] 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).

By using division, Eve performs the *inverse* of the multiplication Alice used to generate her public key.[13] 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's or Bob's private key, Eve can compute the shared key and use it to decrypt messages between Alice and Bob.

It should not be surprising that a DH implementation using multiplication to generate shared keys is not secure. 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.

---

[11] Alice and Bob could have selected *any* private keys (besides 3 and 4, respectively) and the effect in step 9 would have been the same: they would have computed identical secret keys.

[12] This fact is formalized in Kerckhoffs's principle, proposed by Auguste Kerckhoffs in 1883, which turned 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 ciphers depended on the secrecy of their algorithms. One important implication of Kerckhoffs's principle is that a cipher whose algorithm is widely-known will invite attacks by very clever cryptanalysts, and that this is the only effective way to test its efficacy. Indeed, the best cryptosystems in the world are those that have defied successful attacks over a long period of time.

[13] Division is the inverse of multiplication, just as subtraction is the inverse of addition.

DH With Exponentiation

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

Step	Alice	Eve	Bob	Calculations
1	2	->		
2		2	2	
3	3			
4	8	->		(2 ^ 3 = 8)
5		8	8	
6			4	
7		<-	16	(2 ^ 4 = 16)
8	16	16		
9	4096		4096	(16 ^ 3 = 4096), (8 ^ 4 = 4096)

Figure 2. Diffie-Hellman key exchange using exponentiation.

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.[14]

Solving for *y* in this equation is known as taking the *logarithm* of *z.*[15] Taking a logarithm is the inverse of exponentiation, just as division is the inverse of multiplication, which 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. For larger values of *z*, the complexity of Eve's task increases somewhat, but not sufficiently to thwart her attacks.[16]

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.

[14] In the present example, Eve must find the value of *y* in the equation 2 ^ *y* = 8?
[15] Strictly speaking, it is called taking the *logarithm of z base x*.
[16] Contrast this with the version of DH using multiplication, in which the complexity of Eve's task remains constant; that is, Eve must divide the value of the generator by that of the public key only once, regardless of the size of *z*.

Step	Alice	Eve	Bob	Calculations
1	3			
2	7	->		
3		(3,7)	3	
4			7	
5	3			
6	6	->		(3 ^ 3 = 6 mod 7)
7		6	6	
8			4	
9		<-	4	(3 ^ 4 = 4 mod 7)
10	4	4		
11	1		1	(4 ^ 3 = 1 mod 7), (6 ^ 4 = 1 mod 7)

Figure 3. Diffie-Hellman key exchange using modular exponentiation.

Modular exponentiation is the same as exponentiation, but with an additional step. This additional step is called taking a *modulus*, which is done using the *modulo* operation.[17] 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 a number by which to divide in order to find a remainder. We call this operand the *divisor*. In this version of DH, Alice must transmit two numbers to Bob instead of one. As before, Alice transmits the generator (3), but she also transmits the divisor (7) that will be used to compute remainders. Both values are observed by Eve.

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.[18]

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.

[17] 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.

[18] Exponentiation of a generator *g* modulo *p*, where *g* is greater than 1 and *p* is a prime number, guarantees the result will be in 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.

Exponent	Power	Mod	Calculations
1	3	3	(3 ^ 1 = 3 mod 7)
2	9	2	(3 ^ 2 = 2 mod 7)
3	27	6	(3 ^ 3 = 6 mod 7)
4	81	4	...
5	243	5	
6	729	1	
7	2187	3	
8	6561	2	
9	19683	6	
10	59049	4	
11	177147	5	
12	531441	1	

Figure 4. The multiplicative group of integers modulo 7.

Figure 4 reveals some more interesting properties of the values in the *Mod* column (recall this is the set from which our generated keys come). 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*), forming what are known as *cyclic groups*. Second, each cyclic group contains every integer in the set 1 to *p* - 1, where *p* is the divisor. Finally, the cyclic group is unordered, or *non-monotonic*, relative to the order of the exponents.[19]

To keep things simple, it will suffice to keep the following rule in mind: Given a carefully chosen generator *g*, and a prime divisor *p*, we can generate keys with the aforementioned properties. And it is these properties that are required for an effective DH implementation.[20]

Recall from the previous example that Eve had to solve for the logarithm of *z* base *x* (that is, solve for *y* in the equation *x* ^ *y* = *z*) to find the shared key; a task of manageable complexity, even for very large values of *z*. In the finite group of integers modulo *p*, however, Eve's task becomes intractable given big enough values of *p*. This additional complexity is due largely to the *non-monotonicity* of values in a well-formed cyclic group; finding a value in an unordered set is much more difficult than finding one in an ordered one.[21]

There is no *known*, efficient algorithm for finding logarithms in the group of integers modulo *p* (i.e., finding *z* in the equation *x* ^ *y* = *z* mod *p*, where the values of *x*, *z* and *p* are known). If *p* is large enough, the task becomes too computationally expensive to be feasible for an attacker. Formally, this is known

[19] 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.

[20] 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 divisor. Any generator that fulfills this property is called a *primitive root*, and the group it produces a *cyclic* group. 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 cyclic group of integers modulo 7. In real-world DH, the divisor should be a very large, randomly-chosen prime number. An artificially small value is used in the example to keep the math simple.

[21] *Linear* versus *logarithmic* time.

as the *discrete* logarithm (or DL) problem, and the efficacy of DH is based, at least in part, on the difficulty of solving it.[22]

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 the second important component of the public-key cryptosystem described, but not solved, by Diffie and Hellman in their 1976 article.[23] 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 article only presented an actual implementation for one of them.

While doing academic research at MIT in 1978, Ronald Rivest, Adi Shamir and Leonard Adleman published an article titled *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*.[24] 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 Adleman form the basis the most widely known and battle-tested public-key cryptosystem in the world. It is known simply by the initials of the surnames of its authors, or RSA. From the early 1990s well into the 2000s, RSA paved the way for an explosion of ecommerce on the internet that would not have otherwise been possible.

Substitution Cipher

As with DH, we'll start with a simple example to build a conceptual model before moving to a more realistic one.

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 it, Bob encrypts the message prior to sending it to Alice. On receipt, Alice decrypts the message by inverting it to its original, unencrypted form.

---

[22] It is possible there is some other, as yet unknown (or at least unpublished), way to break DH; that is, besides solving the DL problem efficiently. Until or unless such a method is found, no distinction is made between the DL problem and the so-called *Diffie-Hellman problem, or DH, problem*.

[23] There was a third component, *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.

[24] Rivest, Ronald; Shamir, Adi.; Adleman Leonard. (February 1978). "A Method for Obtaining Digital Signatures and Public Key Cryptosystems". *Communications of the ACM*. **21** (2): 121-126.

Step	Alice	Eve	Bob	Calculations
1	11			
2	26	->		
3		(11,26)	11	
4			26	
5			**3**	
6		<-	7	(3 x 11 = 7 mod 26)
7	7	7		
8	19			
9	**3**			(7 x 19 = 3 mod 26)

Figure 5. Encryption and decryption using a substituion cipher.

In steps 1, 2, 3 and 4, Alice selects a public encryption key (11) and a divisor (26), and transmits both to Bob.[25] 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 Eve.

In steps 5, 6 and 7, Bob creates a plaintext message (3),[26] 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).[27] 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 Eve.

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

[25] 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—i.e., their *greatest common divisor*—must be 1.

[26] 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).

[27] *Plaintext* and *ciphertext* are the terms of art for unencrypted and encrypted messages, respectively.

Keys (e, d)	Divisor n	gcd(e, n)	Inverse? d x e = 1 mod n	Calculations
1	26	1	11	(11 x 1 = 11 mod 26)
2	26	2	22	(11 x 2 = 22 mod 26)
3	26	1	7	(11 x 3 = 7 mod 26)
4	26	2	18	(11 x 4= 18 mod 26)
5	26	1	3	...
6	26	2	14	
7	26	1	25	
8	26	2	10	
9	26	1	21	
10	26	2	6	
11	26	**1**	17	
12	26	2	2	
13	26	13	13	
14	26	2	24	
15	26	1	9	
16	26	2	20	
17	26	1	5	
18	26	2	16	
19	26	1	**1**	(11 x 19 = 1 mod 26)
20	26	2	12	
21	26	1	23	
22	26	2	8	
23	26	1	19	
24	26	2	4	
25	26	1	15	
26	26	26	0	

Figure 6. Finding the modular multiplicative inverse.

In the figure, the *Keys (e, d)* column contains the set from which Alice selects her public encryption key *e* and her private decryption key *d*. 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 there exists an inverse—more specifically, a *modular multiplicative inverse*—for *e* in the set of integers modulo 26.[28] 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, 11, 15, 17, 19, 21, 23 or 25).

Now that Alice has selected a suitable *e*, she must find its inverse to act 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 satisfies this requirement is 19 (19 x 11 = 1 mod 26).[29]

[28] The term *modular multiplicative inverse* sounds scary, but it is to modular exponentiation what division is to multiplication, and logarithm is to exponentiation. Unlike with multiplication and exponentiation, however, not all members of a set of integers modulo *n* have an inverse; and only an encryption key *e* that has such an inverse will be invertible by its corresponding decryption key *d*.

[29] Efficient algorithms exist for finding the greatest common divisor of two integers, and for finding the modular multiplicative inverse of an integer in a set 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 prohibitively slow.

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 quickly do an exhaustive search of the set 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.

				Parameters		
Step	**Alice**	**Eve**	**Bob**	**Public**	**Private**	**Calculations**
1	5				p	
2	7				q	
3	35	->		n		5 x 7 = 35
4		35	35			
5	24				t	(5 - 1) x (7 - 1) = 24
6	5	->		e		gcd(5, 24) = 1
7		5	5			
8			**3**			
9		<-	33			(3 ^ 5 = 33 mod 35)
10	33	33				
11	29				d	(29 ^ 5 = 1 mod 24)
12	**3**					(33 ^ 29 = 3 mod 35)

Figure 7. Encryption and decryption using "textbook" RSA.

Again, Bob wishes to transmit a message to Alice (the letter *c* again which, as in the previous example, 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 (in the real world the values of *p* and *q* would be enormous; somewhere on the order of 600 decimal digits in length!).[30]

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 can divide both—or their *greatest common divisor*—must be 1.

[30] 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.

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 previous footnote for the definition of *semiprime*). The formula therefore becomes the product of the totients of *n*'s prime factors *p* and *q*; i.e., (*p* - 1) (*q* - 1). Plugging in the values from the example we get (5 - 1) (7 - 1) = (4 x 6) = 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 highlighted entries in the table is 24.

e	n	gcd(e, n)
1	35	1
2	35	1
3	35	1
4	35	1
5	35	5
6	35	1
7	35	7
8	35	1
9	35	1
10	35	5
11	35	1
12	35	1
13	35	1
14	35	7
15	35	5
16	35	1
17	35	1
18	35	1
19	35	1
20	35	5
21	35	7
22	35	1
23	35	1
24	35	1
25	35	5
26	35	1
27	35	1
28	35	7
29	35	1
30	35	5
31	35	1
32	35	1
33	35	1
34	35	1
35	35	35

Figure 8. Euler's totient function.

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).[31] 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 = 33 mod 35). 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 = 3 mod 35).

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

[31] 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.

Keys (e, d)	Totient t(n)	gcd(e, t)	Divisor n	Inverse? d x e = 1 mod t	Calculations
1	24	1	35	5	(5 x 1 = 5 mod 24)
2	24	2	35	10	(5 x 2 = 10 mod 24)
3	24	3	35	15	(5 x 3 = 15 mod 24)
4	24	4	35	20	(5 x 4 = 20 mod 24)
5	24	**1**	35	1	(5 x 5 = 1 mod 24)
6	24	6	35	6	(5 x 6 = 6 mod 24)
7	24	1	35	11	...
8	24	8	35	16	
9	24	3	35	21	
10	24	2	35	2	
11	24	1	35	7	
12	24	12	35	12	
13	24	1	35	17	
14	24	2	35	22	
15	24	3	35	3	
16	24	8	35	8	
17	24	1	35	13	
18	24	6	35	18	
19	24	1	35	23	
20	24	4	35	4	
21	24	3	35	9	
22	24	2	35	14	
23	24	1	35	19	
24	24	24	35	0	
25	24	1	35	5	
26	24	2	35	10	
27	24	3	35	15	
28	24	4	35	20	
29	24	1	35	**1**	(5 x 29 = 1 mod 24)
30	24	6	35	6	
31	24	1	35	11	
32	24	8	35	16	
33	24	3	35	21	
34	24	2	35	2	
35	24	1	35	7	

Figure 9. Finding the modular multiplicative inverse.

In the figure, 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 set of integers modulo *t* (24). The only value that fits the bill is 29, because that is the first *d* in the equation $d \times e = r \bmod t$ that leaves a remainder *r* of 1[32], and recall that this is the equation for a valid 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.

[32] Actually 5 works, too, but using the same *e* and *d* would be a silly choice.

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).[33] This is known as the *integer factorization problem*.

The public-key schemes of secure key exchange (DH) and encryption share the property that the computations required by Alice and Bob to send each other messages securely—identifying large primes, computing greatest common divisors, identifying modular multiplicative inverses and integer exponentiation—all have efficient algorithms; whereas the computations required by Eve to defeat these schemes—computing discrete logs and factoring integers—do not. It is on this principle of *one-wayness* that the efficacy of public-key cryptosystems relies.

RSA as an Alternative to DH

An important implication of the advent RSA was its ability to replace DH as a key-exchange mechanism. To understand this, it is helpful to consider two observations. First, with RSA encryption, two parties now have the ability to exchange symmetric keys securely, rather than derive them independently in the manner specified by DH. Second, if RSA provides encryption, by means of its mathematically related public and private keys, why do we need symmetric encryption keys at all anymore, never mind the ability to exchange them?

To the second question, the short answer is that public-key encryption is far less computationally efficient than symmetric-key encryption, thus making it less suitable as a scheme for message encryption. Given this shortcoming, the approach that emerged was to use public keys to encrypt symmetric keys, and symmetric keys to encrypt actual messages. Such schemes in fact still exist, and are referred to as a *hybrid*, or *integrated*, encryption schemes because they combine both public- and symmetric-key encryption, the former for key exchange and the latter for message encryption.

As to the first observation—that RSA is a suitable replacement for DH as a mechanism for key-exchange—it helps to keep in mind that although DH is often thought of as a key *exchange* protocol, strictly speaking it is a key *agreement* protocol. That is, the product of DH—a shared, symmetric key that is used by two parties in a private message exchange—is never actually exchanged, but rather computed (or *agreed* to) by the parties independently. Crucially, in DH the symmetric key is never transmitted over an insecure channel, and therefore cannot be intercepted by an eavesdropper, whereas in RSA this is not the case.

Perfect Forward Secrecy

Replacing key *agreement* (DH) with key *exchange* (RSA) results in the loss of a critical means of attack mitigation; namely, a property of encrypted messages known as *perfect forward secrecy.* Perfect forward secrecy provides that messages, once encrypted, remain so even after transmission; that is, after they have been stored on some persistent medium such as a database or a filesystem.

To make this concrete, imagine that Eve intercepts and keeps every message Alice ever sends to Bob. Then, at some future date, Eve manages to steal Bob's private key. Eve can now use Bob's private key to

[33] 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.

decrypt every symmetric key Alice ever sent to Bob, and once Eve has the symmetric keys, she can decrypt every message Alice ever sent to Bob, too.

The loss of perfect forward secrecy is not confined to key-exchange in hybrid schemes such as this. Any scheme in which messages—containing symmetric keys or otherwise—are encrypted using long-term, public keys is vulnerable to loss of the private key, and therefore does not provide perfect forward secrecy. In addition to the inefficiency of public-key encryption mentioned previously, the absence of perfect forward secrecy lends even more weight to using symmetric-key cryptography for message encryption in general.

Given these shortcomings, the preferred approach is for two parties to use DH to generate a symmetric key independently, and to use that key to encrypt and decrypt one, and only one, message. For each subsequent message, a fresh key should be generated. Because is it used for a single message only, and then discarded, the symmetric key in this scheme is referred to as an *ephemeral* key.

Under such a scheme, recovery by an attacker of the symmetric key is no longer possible with possession of the recipient's private key, nor for practical purposes by any other means. Moreover, it is scarcely worth the attacker's effort to crack a symmetric key that is used only once to encrypt a single message.

Digital Signatures

Digital signatures are the third and final component of the public-key cryptosystem originally conceived by DH, and implemented by RSA. Digital signatures serve three main purposes in digital communication: *message integrity*, *authentication of origin* and *non-repudiation of origin*.[34] These properties prove to the recipient of a digitally signed message that (a) it was unaltered in transit, (b) it originated from its purported sender and (c) the purported sender cannot repudiate either (a) or (b).[35]

In this scheme, the sender of a message *signs* it using his or her private key, and on receipt of the message the receiver *verifies* it using the signer's public key.[36] If the verification fails, this means either the private key used to sign the message does not correspond to the public key used to verify it, or that the message was altered after the sender signed it. In either case, the signature is invalid. In short, a digital signature binds the identity of a message's signer to one, and only one, message.

All this comes with a very important caveat: If a private key is stolen from its owner, it can be used to sign messages the owner did not in fact sign. Because of this possibility, it is virtually impossible to prove in a court of law that, just because a message was signed with an owner's private key, the message originated from the owner. In practice, however, digitally signed messages demand a much lesser burden of proof.[37]

---

[34] They also have some novel use cases, such as securing ownership of digital tokens like Bitcoin.

[35] Digital signatures are to electronic documents what hand-written signatures are to paper documents; they prove that the signer authorizes the contents of the document. Whereas hand-written signatures can be forged, and/or the documents they belong to altered, neither is possible with digital signatures.

[36] Strictly speaking, in RSA it is the signature-message *pair* that is verified, not simply the message. A signature is in fact itself just a message; that is, it is a *permutation* of the unsigned message. This permutation is computed by raising the unsigned message to the power of the signer's decryption key modulo the public divisor.

[37] For example, to satisfy oneself that an executable file downloaded from a website can be trusted.

Digital signature is the *inverse* of encryption, in the sense that in the latter messages are encrypted with a public encryption key *e*, and decrypted with a corresponding decryption key *d*; whereas in the former messages are *signed* with a decryption key *d*, and *verified* with a corresponding encryption key *e*.[38]

Figure 10 depicts digital signature and verification. Recall from the example of encryption in *Figure 7* that Alice computed a public-private key pair; 5 and 29, respectively. Using the same key pair, Alice now signs a message (4) and transmits it to Bob, who verifies it on receipt.[39]

Step	Alice	Eve	Bob	Calculations
1	**4**			
2	9	->		(4 ^ 29 = 9 mod 35)
3		9	9	
4			**4**	(9 ^ 5 = 4 mod 35)

Figure 10. Digital signing.

In steps 1, 2 and 3, Alice signs the message (4) and transmits it to Bob. To do this she raises the message (4) to the power of her encryption key *d* (29), and takes the modulus from the divisor *n* (35) to compute the signed message (9).

In step 4, Bob verifies the signed message. He raises the signed message (9) to the power of Alice's public key *e* (5), mods it with the divisor *n* (35), and arrives back at the original, unsigned message (4).

Of course, Eve has observed the signed message (9), and because she also knows Bob's public key (5), she will be able to read it. In a real-world scenario, Alice would have signed the message with her private decryption key, and then encrypted it with Bob's public key, before transmitting the message to Bob. On receipt of the message, Bob would perform the inverse procedure; he would decrypt the signed message with his private decryption key, and then verify the signature with Alice's public encryption key.

Attacks

The schemes of the public-key cryptosystem described in this paper are at once powerful and elegant. Moreover, they are virtually ubiquitous in securing electronic commerce, online banking and all manner of sensitive communication on the internet. But a discussion of them would not be complete without pointing out a glaring weakness to which they are all susceptible. This vulnerability is known as the *man-in-the-middle* attack, or MITM for short.

Imagine the following familiar scenario: Bob wishes to send an encrypted message to Alice. As ever, Eve is listening. Alice computes her public-private key pair and transmits the public key to Bob. Eve, meanwhile, intercepts Alice's key, substitutes it for a public key of her own, and forwards it to Bob. Bob, *having no way of knowing that the public key he receives belongs to Eve and not Alice*, blithely encrypts the message intended for Alice using Eve's public key. Eve intercepts Bob's message and decrypts it with

[38] Because of this invertibility, encryption and digital signature are *permutations* of one another. Put another way, every message is some other message's ciphertext, and every ciphertext is itself a valid message.

[39] For Alice to encrypt the signed message, she would need Bob's public key. Since we already know the mechanism for public-key encryption, these steps are omitted from the diagram to keep the focus on digital signature.

her private key. Eve has thus broken RSA encryption and, what's more, she has done so without using any math.[40]

This simple example captures the essence of MITM, and a real-world version of it might look very much the same. *Every* public-key scheme—whether secure key exchange, encryption or digital signature—is vulnerable to this attack.

How can this be if the security of the internet is based on public-key schemes? The answer is a Frankenstein-like bolt-on to the public-key cryptosystem called *public-key infrastructure*, or PKI for short.

Attack Mitigation

Not wanting their legacy to be relegated to one of interesting academic research, RSA's inventors founded a company seeking to commercialize their invention not long after publishing *A Method for Obtaining Digital Signatures*. Acutely aware of the MITM problem, the founders knew that *RSA Security Inc.*'s success would depend largely on an effective solution to it. That solution was to be PKI.

At a very high level, a PKI establishes a trust relationship between Alice, Bob and a third party known as a *certificate authority* (CA). The CA's role in the PKI is to vouch for the authenticity of public keys, which it accomplishes by binding public keys to the identities of their owners. For a PKI to be effective, the CA has to be trusted by both Alice and Bob.

To make this more concrete, take the recent example involving Alice and Bob, where Eve mounted an MITM attack, but this time in the context of a PKI.

After generating her public key (but before transmitting it to Bob), Alice submits it to a CA. The CA vets Alice, and if it concludes she is trustworthy, combines her public key with some name that uniquely identifies Alice into a digital document known as a *certificate*.[41] The CA signs this certificate with its own private key, and returns the signed certificate to Alice. Now, when Bob wants to send an encrypted message to Alice, instead of sending her public key to Bob she sends her CA-signed certificate, which contains her public key. On receipt of the certificate, Bob verifies it using the CA's public key. If the verification succeeds, Bob knows the public key contained in the certificate in fact belongs to Alice, and proceeds to encrypt messages with it. This time, if Eve intercepts Alice's certificate and forwards Bob a phony one of her own, Bob's verification step will fail and he will know he is being attacked.

Though effective at thwarting MITM attacks, PKI is not without weaknesses of its own; chief among them the trustworthiness (or lack thereof) of the CA. Indeed, there are hundreds of public CAs the world over, the vast majority of which have sterling reputations. However, CAs themselves are not impervious to attack, and in some cases have been compromised.[42]

---

[40] MITM is an illustration of the *weakest-link* maxim, which in the information security setting holds that a system is only as secure as its weakest link. From the perspective of an attacker, the most rational approach to defeating a cryptosystem is to attack its weakest link. In a public-key cryptosystem, the MITM vulnerability is its weakest link.

[41] The official specification for public-key certificates is defined by the International Telecommunications Union's (ITU) *X.509* standard.

[42] In some cases in spectacular fashion. See *DigiNotar* for the quintessential case study.

Going Forward

For all their power and elegance, cryptosystems based on public-key primitives are imperfect; chiefly because of the size and complexity of the PKI required to thwart MITM. Nevertheless, these cryptosystems are the best we have in the internet age. Indeed, public-key cryptography is responsible for securing the vast majority of sensitive communications on the internet today, and it achieves this objective with a remarkable degree of success.

The advent of cryptocurrencies, in particular the distributed blockchains on which they run, offers tantalizing prospects for the establishment of trust that today is centralized in a handful of global certificate authorities. A blockchain-based PKI would decentralize trust, spreading it across a distributed network of synchronized ledgers, instead of concentrating it in the hands of a vulnerable few. Progress on this front has been halting, not least because its efficacy depends on an alignment of incentives that many would-be applications of a distributed blockchain—such as public-key authentication—lack.[43]

Meanwhile, quantum computing looms menacingly on the horizon, and threatens the existence of public-key cryptography as we know it. Whereas the most powerful classical computer in the world would require years to reverse the one-way functions of public-key cryptography, a fit-for-purpose quantum computer could do it in a matter of seconds. You can bet that well-funded, state-level actors are trying to build a quantum computer with such capabilities right now. The invention of such a computer would render classical public-key cryptography instantly obsolete, and enable its owners to blow up the entire system of internet commerce, online banking and, likely, systems on the which the physical security of nation-states depends.

[43] This alignment of incentives, known as a *consensus algorithm*, works remarkably well with cryptocurrency; the blockchain's original use case. This is because curators (aka *miners*) of such blockchains are compensated with tokens (e.g., bitcoin), which have monetary value, for being honest.