The Elements of Public-Key Cryptography

Persistence

In the fall of 1974, an undergraduate at the University of California, Berkeley submitted two proposals for his term project in CS244, a course on computer security offered in the University's department of Computer Science. One of the projects would concern data compression, and the other secure communications over insecure channels.

The student, a senior by the name of Ralph C. Merkle, was lukewarm about the project involving data compression. Devoting just two sentences to it in the proposal, he wrote, "At this point, I must confess, that I am not entirely thrilled by the prospect of engaging in this project, and will expand on it only if prodded." The other proposal, the one having to do with secure communications, occupied seven type-written pages.

Unfortunately for Merkle, his professor, a Computer Science faculty member named Lance Hoffman, chose to prod. In the margin of Merkle's proposal he scribbled, "Project 2 [data compression] looks more reasonable, maybe because your description of Project 1 [secure communications] is muddled terribly."

Undeterred, Merkle submitted a second, much simplified proposal for his preferred topic, condensing it to just over a single page. This time, Hoffman scribbled, "It's getting there, but not good enough yet," and, in a display of intellectual snobbery worthy of a true academic, "All bloat," referring apparently to all but the final paragraph of the refined proposal.

Most college seniors in Merkle's position would have given in, wont as they are simply to drag themselves over the finish line. But not only was Merkle sufficiently entranced by his idea not to abandon it, he dropped Hoffman's course in order to pursue it!

The following summer, having completed the project on his own and packaging it into publishable form, Merkle submitted it for review to the editor of the *Communications of the Association for Computing Machinery*—better known simply as the ACM—a distinguished society of academics and professionals at the forefront of the field.

The response, solicited by the editor from an "experienced cryptography expert," was disappointing. Among other things, it offered, "I am sorry to have to inform you that the paper is not in the mainstream of present cryptography thinking and I would not recommend that it be published in the *Communications of the ACM*." As if to pile on, the editor who'd submitted the paper on Merkle's behalf remarked that she too was bothered, particularly "by the fact that there are no references to the literature."

In the editor's remark Merkle took some encouragement. The reason there were no references is because none existed! And the criticism, coming from someone so highly placed, confirmed to Merkle not only that his idea was a novel one, but therefore one for which he would receive full credit, and perhaps even a measure of recognition from respected thinkers in the field.

Merkle continued to badger the ACM until finally, in April 1978—nearly four years after first floating his idea to professor Hoffman—it published his paper.[1]

Recalling his years-long effort to trigger a Eureka moment in the minds of skeptics, Merkle cited the brilliant 20th century biologist Peter Medawar, who said, "The human mind treats a new idea the same way the body treats a strange protein; it rejects it."

Alas, publication did not earn Merkle enduring recognition. Today, few outside the field of cryptography recognize his name. Instead, the names Whitfield Diffie and Martin Hellman are the ones that are remembered, but their work nevertheless owes a great deal to Merkle's novel idea. To their great credit, Diffie and Hellman graciously cite Merkle's contributions in what was to become the seminal treatise in the brand new field of *public-key cryptography*.

Keys and the Key Distribution Problem

The need to communicate in secret is as old as communication itself.[2] 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 method of scrambling, and unscrambling, is generally referred to as the *key*.[3]

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.[4] 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.[5]

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.[6]

---

[1] Ralph C. Merkle (April 1978). Secure Communications Over Insecure Channels. *Communications of the ACM*.
[2] King to general, via trusted courier: *Attack at dawn!*
[3] 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.
[4] The quintessential example of such an insecure channel is the public internet.
[5] The classic *chicken-or-egg* problem.
[6] Accepting that each *n*-party communication requires a separate key, 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, and so on. As the number of participants increases, the number of keys grows quadratically.

A Clever Solution

In 1976, two Stanford University cryptographers proposed an elegant solution to the key distribution problem in a groundbreaking academic article.[7] This solution became, and remains to this day, the de facto standard for exchanging keys securely over public, digital communication channels. It is commonly known as the *Diffie-Hellman key exchange protocol*, or DH for short.[8]

DH is a foundational element of *public-key* cryptography, a breakthrough without which the conveniences of e-commerce and online banking would not be possible at internet scale.[9]

DH enables previously unacquainted 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 (or otherwise derived) by an eavesdropper.

Essential Diffie-Hellman

DH can be implemented by means of a number of algorithms. Most examples in the literature cite the original implementation, which uses the *multiplicative group of integers modulo a prime number*, to demonstrate DH. This is unfortunate, because the mathematics of multiplicative groups are not immediately intuitive.

The graphic in *Figure 1* demonstrates DH using a much simpler algorithm: *multiplication*. With this we will be better prepared to grasp a real-world implementation based on multiplicative groups.

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.[10]

---

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

[8] The Diffie-Hellman protocol was a souped-up variation of the approach invented by Ralph C. Merkle, which employed puzzles (colloquially referred to as *Merkle's Puzzles*).

[9] Public-key cryptography is based on the principle that different, though mathematically related, keys—one public and one secret—can be used to secure communications; whereas traditional methods use symmetric keys.

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

Step	Alice	Eve	Bob	Computations
1	2	->		
2		2	2	
3	3			
4	6	->		3 * 2 = 6
5		6	6	
6			4	
7		<-	8	4 * 2 = 8
8	8	8		
9	**24**		**24**	8 * 3 = 24, 6 * 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.[11] 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 another random integer (3), multiplies it by the generator (2), and transmits the product of the multiplication (6) to Bob. Let's call this second random number 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.

In steps 6, 7 and 8, Bob selects a random integer (4), multiplies it by the generator Alice sent to him (2), and transmits the product of the multiplication (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's 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).[12]

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.

[11] Since computers operate on numbers—and even more specifically *integers* in cryptographic implementations—we use integers in this and all subsequent examples to represent messages and keys.

[12] 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.

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*.[13]

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

DH for 6th Graders

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

Step	Alice	Eve	Bob	Computations
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 is really just another way of saying repeated multiplication, as in $10^3 = 10 \times 10 \times 10 = 1000$). 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.

---

[13] 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 publicly known will open it to the widest possible scrutiny, and that this is the surest way to expose its weaknesses. Indeed, the best cryptosystems in the world are those that have defied successful attempts to defeat them over a long period of time.

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

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 only x and z are known.[15]

Solving for y in this equation is known as taking the *logarithm* of z.[16] 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.[17]

With this we get closer to an effective DH implementation, but we're not quite there yet.

DH for Cryptographers

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

[15] In the present example, Eve must find the value of y in the equation $2^y = 8$.
[16] Strictly speaking, it is called taking the *logarithm of z to the base x*.
[17] 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	Computations
1	3			
2	7	->		
3		(3,7)	3	
4			7	
5	3			
6	6	->		$3^3 = 6 \bmod 7$
7		6	6	
8			4	
9		<-	4	$3^4 = 4 \bmod 7$
10	4	4		
11	**1**		**1**	$4^3 = 1 \bmod 7$, $6^4 = 1 \bmod 7$

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.[18] If you compare the graphic in this example with the one in *Figure 2* 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 a number by which to divide in order to find a remainder. We call this operand the *divisor* or, alternatively, the *modulus*. 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 modulus (7) that will be used to compute remainders. Both the generator and the modulus are observed by Eve.

The modular exponentiation of real-world DH leads to a very interesting property of the keys it generates. 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 consecutive integers from 1 to 6.[19]

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 ***Remainder*** 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 ***Remainder*** column.

[18] A modulo operation simply finds the remainder after division of two numbers. For example, 7 = 1 mod 3, because 7 divided by 3 equals 2, leaving a remainder of 1.

[19] 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	Remainder	Computations
1	3	3	$3^1 = 3 \bmod 7$
2	9	2	$3^2 = 2 \bmod 7$
3	27	6	$3^3 = 6 \bmod 7$
4	81	4	$3^4 = 4 \bmod 7$
5	243	5	$3^5 = 5 \bmod 7$
6	729	1	$3^6 = 1 \bmod 7$
7	2187	3	$3^7 = 3 \bmod 7$
8	6561	2	$3^8 = 2 \bmod 7$
9	19683	6	$3^9 = 6 \bmod 7$
10	59049	4	$3^{10} = 4 \bmod 7$
11	177147	5	$3^{11} = 5 \bmod 7$
12	531441	1	$3^{12} = 1 \bmod 7$

Figure 4. The multiplicative group of integers modulo 7

Figure 4 reveals some more interesting properties of the values in the ***Remainder*** column (recall this is the set from which our generated keys come). First, if you look at the values in the ***Remainder*** column from top to bottom, 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 modulus. Finally, the cyclic group is unordered, or *non-monotonic*, relative to the order of the exponents.[20]

To keep things simple, it will suffice to keep the following rule in mind: Given a carefully chosen generator *g*, and a modulus *p* that is a prime number, we can generate keys with the aforementioned properties. And it is these properties that are required for an effective DH implementation.[21]

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 large enough values of *p*. This additional complexity is due 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 set.[22]

There is no *known*, efficient algorithm for finding logarithms in the group of integers modulo *p* (i.e., finding *y* in the equation $x^y = z \bmod p$, 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 as

[20] These properties are described formally in the area of number theory; specifically multiplicative groups modulo *p*, where *p* is a prime number.

[21] A carefully chosen generator is one that 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*, and the set 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 modulus should be a very large, randomly-chosen prime number. An artificially small value is used in the example to keep the math simple.

[22] *O(n)*, or linear, versus *O(log n)*, or logarithmic, time.

the *discrete logarithm problem* (or DLP), and the efficacy of DH is based, at least in part, on the difficulty of solving it.[23]

Toward a Complete Cryptosystem

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 a workable implementation for one of them.

Beyond secure key exchange, encryption was the second component of the public-key cryptosystem promised, but not delivered, by Diffie and Hellman in their 1976 article.[24] For public-key encryption, they and the rest of the crypto community would have to wait another two years.

Public-Key Encryption

While doing academic research at MIT in 1978, Ronald Rivest, Adi Shamir and Leonard Adleman proposed a novel solution for transforming a message from plaintext to ciphertext and back again. This alone would have been an unremarkable feat—it pretty much described every previously known cipher. What *was* remarkable is that their method worked without the upfront exchange of secret keys; the sine qua non of traditional ciphers.[25] Not only had this trick never been accomplished before, unshackling encryption from key exchange offered the potential of securing communications at otherwise impossibly massive scale.

Rivest, Shamir and Adleman published their method in an article titled *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*.[26] In it, they picked up where Diffie and Hellman left off, introducing public-key implementations for encryption and digital signatures.[27] The scheme is known simply by the initials of the surnames of its authors, or RSA. More than 40 years on, the contributions of Rivest, Shamir and Adleman form the basis the most widely used and battle-tested public-key cryptosystem in the world.

RSA Encryption in a Nutshell

RSA encryption can be understood in terms of a very simple state-transition diagram. In this diagram, *M* represents a plaintext message, *C* represents a ciphertext transformation of that message, and the arrow symbol (->) represents a transition from one state—or *form*, if you prefer—to another. The state transitions should be read from left to right.[28]

---

[23] 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*.

[24] The third component was *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.

[25] Traditional ciphers were burdened by the complexity associated with the exchange of shared, secret keys; a problem that was solved by Diffie and Hellman, and that is described in detail in a previous section of this paper.

[26] 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.

[27] Digital signatures are a topic unto themselves, and are the subject of a subsequent section of this paper.

[28] Keep in mind the message *M* here is in fact a numeric representation of the message fit for computer consumption (e.g., ASCII or UTF8).

(1) $M \rightarrow C \rightarrow M$

Here, some plaintext message *M* is transformed to a ciphertext *C*, and then back again to its original state *M*.

For encryption to be useful, the second transformation, from *C -> M*, must be easy to compute for the intended receiver of the message, but *not* for an eavesdropper.[29]

To see how this requirement is enforced without a shared key, as would be required by a traditional cipher, let's look at how the transformations are implemented in RSA. First, let's substitute the term *C* from diagram (1) with a new, equivalent term M^e.

(2) $M \rightarrow M^e \rightarrow M$

Here, the term M^e denotes that we are raising the message *M* to the power of some positive exponent *e* to transform it to ciphertext. So, in diagram (2), the middle term *C* becomes M^e.

Now let's substitute the rightmost term *M* from the previous diagram with the equivalent term $(M^e)^d$. Again, in diagrams (1) and (2), the rightmost *M* is equal to the leftmost *M*, after having been recovered from its ciphertext form.

(3) $M \rightarrow M^e \rightarrow (M^e)^d$

Here, the term $(M^e)^d$ denotes that we are raising the term M^e (otherwise known as *C*) to the power of some other positive exponent *d* to recover the plaintext *M*. So, in diagram (3), the rightmost term *M* becomes $(M^e)^d$.

If we accept that each of the preceding state-transition diagrams is equivalent, just with a substitution of terms, then by definition the terms $(M^e)^d$ and *M* must also be equivalent; that is, $(M^e)^d = M$.

But how can this be? If *e* and *d* are positive exponents of the base term *M*, then it must be the case that $M < M^e < (M^e)^d$ and, therefore, *M* cannot be equal to $(M^e)^d$.

In fact they cannot be equal in the infinite domain of integers. But in a *finite* integral domain—specifically the finite group of integers modulo *n* first visited in our discussion of DH—such an equivalence is very possible indeed.

To see how, let's rewrite the state transition diagram in the modular arithmetic of finite groups.

(4) $M \rightarrow M^e \bmod n \rightarrow (M^e)^d \bmod n$

It should be clear now how the leftmost term *M* can in fact be equal to the rightmost term $(M^e)^d$, *if* we reduce it *mod n*.

To make the point concrete, let's plug some real values into the diagram, where the integer *5* represents the plaintext message *M*.

(5) $5 \rightarrow 5^7 \bmod 77 \rightarrow (5^7)^{43} \bmod 77$

[29] As a public-key scheme, it is assumed that all parties—authorized or not—possess the public key required for the first transformation, from *M -> C*, but that only the receiving party possesses the private key required for the second, from *C -> M*.

(6) *5 -> 47 -> 5*

If there is any doubt, this math can easily be verified on any calculator with a *mod* function.

Circling back now to the requirement that the transformation from *C -> M* be easy to compute for the message's recipient, but not for the eavesdropper, this property is enforced by confining knowledge of the exponent *d—43* in diagram featuring real values—to just the message's recipient.

This brilliant round trip, from *M* to *C* back to *M* again, is what is known as a *one-way, trap-door* function. It is one-way because it is easy to transform from *M -> C,* and from *C -> M*. But the transformation from *C -> M* is possible only if some trap-door information is known. In this case, that trap-door information is the exponent *d*.

If it weren't obvious, not just any exponents *e* and *d*, and modulus *n*, will do for these equivalences to hold, and it is in the derivation of these parameters that the sausage of RSA is made.

Textbook RSA

Figure 5 models what is referred to in the literature as *Textbook RSA*.[30] As with previous examples, the parameters are made artificially small to keep the concepts manageable.

				Parameters		
Step	**Alice**	**Eve**	**Bob**	**Public**	**Private**	**Computations**
1	7				p	
2	11				q	
3	77			n		7 * 11 = 77
4	60				t(n)	(7 - 1)(11 - 1) = 60
5	7	->		e		gcd(7, 60) = 1
6		(7, 77)	7	(e, n)		
7			77			
8	43				d	7 * 43 = 1 mod 60
9			**5**	M		
10		<-		C		$47 = 5^7 \bmod 77$
11	47	47				
12	**5**			M		$5 = 47^{43} \bmod 77$

Figure 5. Encryption and decryption using textbook RSA.

If you read the previous section, the last four steps (9 through 12) of *Figure 5* should look familiar. But let's restate them anyway to put them into the context of this example. Bob transforms a plaintext message *M* to a ciphertext message *C*, which he transmits to Alice; whereupon Alice transforms the ciphertext *C* back to the original plaintext *M*. Meanwhile, Eve observes the ciphertext *C*, but without the decryption exponent *d*, she cannot recover the plaintext *M*. To understand how Alice and Bob generated the keys to encrypt and decrypt *M*, however, we need to back up to the beginning.

[30] The *textbook* model provides a good theoretical basis for understanding RSA encryption, but lacks some features a real-world implementation would have.

In steps 1, 2 and 3, Alice selects two random integers, *p* and *q* (7 and 11), and multiplies them to produce a modulus *n* (77). Alice's requirements for a suitable *n* are that its factors *p* and *q* both be distinct prime numbers, and that the numerical value of the message she encrypts be less than *n* - 1.[31]

In step 4, Alice applies *Euler's totient function* to the modulus *n*; this is denoted symbolically as *t(n)*. Given some positive integer *n*, Euler's totient function tells us the number of integers from 1 to *n* with which *n* is *relatively prime* (or *coprime*).[32] For a prime number *p*, the result is simple: the totient of *p*, or *t(p)*, is equal to *p* - 1. It should be clear why this is so. If *p* is prime, we know that the only integers that divide it are 1 and *p* itself. Therefore, every integer in the set 1 to *p* - 1 must be relatively prime to *p*.

But here Alice applies the totient function to the *semiprime* modulus *n*, which is the product of the two primes *p* and *q*.[33] 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 (7 - 1)(11 - 1) = 6 * 10 = 60.

That is, Euler's totient function tells us that there are 60 integers in the set 1 to 77 that are relatively prime to 77. Indeed, the number of highlighted entries in the table in *Figure 6* is 60 (the table has been truncated to conserve space). As with the values of *p* and *q*, Alice must keep the totient *t(n)* secret.

[31] In the real world the values of *p* and *q* would be very large, randomly chosen primes that are close in magnitude (but not equal).

[32] For two integers to be *relatively prime* (or *coprime*) the largest integer that can divide both evenly—i.e., their *greatest common divisor*—is 1.

[33] 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 are 1, 7, 11 and 77.

a	n	gcd(a, n)
1	77	1
2	77	1
3	77	1
4	77	1
5	77	1
6	77	1
7	77	7
8	77	1
9	77	1
10	77	1
11	77	11
12	77	1
13	77	1
14	77	7
15	77	1
…	…	…
77	77	77

Figure 6. Euler's totient function for values of 'a' relatively prime to 'n'

In step 5, Alice chooses her encryption exponent *e*. It will be helpful here (and in step 8, where Alice chooses her decryption exponent *d*) to refer to *Figure 7*. Alice's requirements for a suitable *e* are that it be greater than 1, and that it be relatively prime to *t(n)*; that is, it must satisfy the equation *gcd(e, t(n)) = 1*. Crucially, the latter condition guarantees that *e* has a multiplicative inverse in the finite group of integers modulo *t(n)*. Referring to the first and third columns in *Figure 7*, we see that 7 meets both requirements, so that is the value Alice chooses for *e*.

Exponent (e, d)	Totient t(n)	Coprime gcd(e, t(n))	Inverse e * d = 1 mod t(n)	Computations
1	60	1	7	7 * 1 = 7 mod 60
2	60	2	14	7 * 2 = 14 mod 60
3	60	3	21	7 * 3 = 21 mod 60
4	60	4	28	7 * 4 = 28 mod 60
5	60	5	35	7 * 5 = 35 mod 60
6	60	6	42	7 * 6 = 42 mod 60
7	60	1	49	7 * 7 = 49 mod 60
8	60	4	56	7 * 8 = 56 mod 60
9	60	3	3	7 * 9 = 3 mod 60
10	60	10	10	7 * 10 = 10 mod 60
11	60	1	17	7 * 11 = 17 mod 60
…	…	…	…	…
43	60	1	1	7 * 43 = 1 mod 60

Figure 7. Finding the multiplicative inverse of 7 in the group of integers modulo 60

Having selected her encryption exponent, in steps 6 and 7 Alice transmits it, along with the modulus—which together comprise her public key *(e, n)*—to Bob, which Bob can then use to encrypt messages to Alice.

In step 8, Alice chooses her decryption exponent *d*. A suitable *d* must be greater than *e*, and must be the multiplicative inverse of *e* in the finite group of integers modulo *t(n)*; that is, it must satisfy the equation $e * d = 1 \bmod t(n)$.[34] Looking at the first and fourth columns in *Figure 7*, we see that 43 is the first value that meets these requirements (here again the table has been truncated to conserve space).[35]

But where does this leave Eve? Eve has observed the value of the encryption key *(e, n)* and the ciphertext. But she has not observed the prime factors *p* and *q*, and therefore cannot efficiently compute the totient *t(n)*. Without *t(n)*, Eve cannot derive the private exponent *d* of the decryption key *(d, n)* and break the encryption.

The reason Eve cannot efficiently compute the totient *t(n)* is because there is no known, efficient way to derive the prime factors of large, semiprime integers. In the context of the present example, this means that given the modulus *n* (77), Eve must determine that its factors are *p* (7) and *q* (11) in order to break the encryption.[36] This is known as the *integer factorization problem*.

Totients and Inverses

The foregoing example demonstrates the effect of RSA encryption; that is, given some special values *e* and *d* that are *inverses* in the group of integers modulo the *totient* of *n*, we can reliably recover a message *M* from its encrypted form *C*. But what does that all mean, exactly?

To help understand the mechanism at work here, let's resurrect state-transition diagram (4) from the previous section (repeated here as diagram (1)).

(1) $M \rightarrow M^e \bmod n \rightarrow (M^e)^d \bmod n$

Recall that the term $(M^e)^d \bmod n$ is equivalent to the term *M*. Specifically, $(M^e)^d \bmod n$ is the operation that must be carried out on the encrypted form of *M* (expressed in the middle term as $M^e \bmod n$) to recover the original, unencrypted *M*.

---

[34] The term *multiplicative inverse* here means the value by which the encryption exponent *e* must be multiplied to yield 1 or, more precisely 1 *mod n*, where *n* is the number of elements in (or the *order* of) the group. For example, in the real numbers, the multiplicative inverse of 3 is 1/3 (or .333), because 3 * 1/3 = 1. But in the domain of finite integer groups the rules are different; notably, fractional numbers are not allowed. So, the inverse of an element is the factor that, when multiplied by the element, equals 1 mod *n*. Multiplicative inverse is to finite integer groups what division is to multiplication, logarithm is to exponentiation, or subtraction is to addition in the real numbers.

[35] Given the artificially small values used in the example, it is possible to demonstrate the derivation of *e* and *d* with the aid of a simple table like the one in *Figure 7*. For the much larger values used in the real world, however, we need a better way. That better way is the *Extended Euclidean Algorithm*, which provides an efficient means of finding inverses in large-order multiplicative groups.

[36] Given a small, semiprime modulus *n*—e.g., 77 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.

Now, let's see where the totient of *n*, or *t(n)*, enters the picture. First, recall that *e* and *d* are inverses in the group of integers modulo *t(n)*, and *not* the group of integers modulo *n*. This can be confusing because the encryption and decryption operations shown in the diagrams are carried out modulo *n*.

Keeping this distinction in mind, since we know that *e* and *d* are inverses in the group of integers modulo *t(n)*, and that *M* equals $(M^e)^d$ in the group of integers modulo *n*, then we can rewrite the diagram as follows without affecting the outcome:

(2) $M \rightarrow M^e \bmod n \rightarrow (M^e)^{d \bmod t(n)} \bmod n$

Note the only difference between diagram (1) and diagram (2) is that in the latter we've added *mod t(n)* to the exponent (or, more precisely, to the product of the exponents *e* and *d*) in the third and final term. And since $e * d = 1 \bmod t(n)$, then $(M^e)^{d \bmod t(n)} \bmod n$ is really just a longhand way of writing $M^1 \bmod n$.[37]

Let's see what happens when we insert real values into the diagram.

(3) $5 \rightarrow 5^7 \bmod 77 \rightarrow (5^7)^{43 \bmod 60} \bmod 77$

Focusing on the third term, since the product of the exponents is $7 * 43 = 301$, and $301 = 1 \bmod 60$, then reducing the exponent *301* modulo *60* thus gives us:

(4) $5 \rightarrow 5^7 \bmod 77 \rightarrow 5^1 \bmod 77$

And since $5^1 = 5$, and $5 \bmod 77 = 5$, we end up ultimately with a successful decryption of the message.

(5) $5 \rightarrow 47 \rightarrow 5$

We used the number *5* for *M* in this example, but the procedure works just the same for any value of *M* in the set of consecutive integers from 1 to *n*. That is, we can raise any such *M* to the power of *e*, and expect to recover *M* by raising that result to the power of *d* (both operations modulo *n*, of course).

To reiterate, this inversion only works if the exponents *e* and *d* are inverses in the group of integers modulo *t(n)*, and it only works *effectively* if *n* is the product of two prime numbers.[38]

Computational Security

The public-key schemes described thus far share the property that the computations required by Alice and Bob to send each other messages securely—identifying large primes, computing greatest common divisors, computing multiplicative inverses and integer exponentiation—can be done relatively inexpensively; whereas the computations required for Eve to defeat these schemes—computing discrete logarithms and factoring large integers—cannot. It is on this asymmetry, or *one-wayness*, that the efficacy of public-key cryptosystems relies.

As such, these schemes are secure only to the extent that they are too computationally expensive to be feasible for would-be attackers. But, given unlimited time and/or computing resources, they are not

---

[37] Note that $(M^e)^{d \bmod t(n)}$ is equivalent to $M^{e*d \bmod t(n)}$.

[38] If the modulus *n* were *individually* prime, as opposed to the semiprime product of two individually prime numbers, then revealing *n*—which is required by the sender, along with the encryption exponent *e*, to encrypt a message—would allow easy recovery of the decryption exponent *d* by an attacker, thus rendering the encryption useless.

unbreakable; with enough of one or the other or both, an exhaustive search of the keyspace—what is known as a *brute-force* attack—will eventually yield the correct key.

Given this mainly theoretical limitation, these schemes are therefore not *unconditionally* secure; rather, they are *computationally* secure, and may someday wither under attack from ever more powerful computers that are capable of reducing searches of the entire keyspace to a reasonable amount of time, and/or analytical techniques that have not yet been discovered.[39] In fact progress on both fronts has been made in the last half-century, and because of this the recommended key-lengths for RSA have increased several-fold over the years to keep apace of it.

RSA as an Alternative to DH

One interesting consequence of the advent of RSA was its potential to replace DH as a key-exchange mechanism. With RSA encryption, two parties now had the ability to exchange symmetric keys securely, rather than derive them independently in the manner specified by DH. But if RSA provided encryption, why did we need symmetric encryption keys at all anymore, never mind the ability to exchange them?

The short answer is that public-key encryption is much more computationally expensive than its symmetric-key brethren, thus making it less suitable as a scheme for message encryption.[40] As a result, one approach that emerged was to employ public keys to encrypt and exchange symmetric keys and, thus securely exchanged, the symmetric keys to encrypt message content. Such schemes are in fact in wide use, and are referred to as *hybrid* encryption schemes because they combine the advantages of both public- and symmetric-key encryption, the former for key exchange and the latter for message encryption.[41]

As to the 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 almost always referred to 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

Because of this distinction, it turns out that DH is still very much relevant. The *agreement* property of DH provides something that the *exchange* property of RSA does not; namely, a safeguard known as *perfect forward secrecy.* Perfect forward secrecy guarantees that messages, once encrypted, remain so even after they have been transmitted; that is, after they have been stored on a persistent medium such as a database or a filesystem.

---

[39] There is only one known cryptosystem that is unconditionally secure, and that is the *one-time pad*. Its key-length requirements are so cumbersome, however, as to make it impractical for all but the most sensitive applications.
[40] At current clock speeds, RSA requires 3,072-bit key lengths to achieve the same level of security as 128-bit AES, the current standard for symmetric-key encryption. The longer the key lengths, the larget the numbers, and hence the slower the computations required to generate keys.
[41] Schemes that combine DH with symmetric-key encryption are also called hybrid schemes.

Imagine that Eve intercepts, and stores, every message Alice ever sends to Bob. Then, at some future date, Eve manages to steal Bob's private key. Now Eve can use Bob's private key to decrypt every symmetric key Alice ever sent to Bob using RSA encryption, as in the hybrid scheme just described. And once Eve has the symmetric keys, she can decrypt the messages as well.

The loss of perfect forward secrecy is not confined to key-exchange in scenarios that use hybrid schemes. Any scheme in which messages—whether they contain keys or actual message content—are encrypted using long-term, public keys is not perfectly forward secret.

If computational inefficiency weren't reason enough not to use RSA as an encryption protocol, the absence of perfect forward secrecy certainly is. For these reasons, most modern cryptographic implementations eschew RSA entirely for encryption and key exchange alike.[42]

Instead, the standard that has emerged is for two parties to generate a symmetric key independently, using DH, and to use that key to encrypt and decrypt one, and only one, message using a fast and secure symmetric protocol like AES.[43] For additional security, a fresh key can be generated for each new message. Because is it used only once and then discarded, the symmetric key in this scheme is referred to as an *ephemeral* key. Now, not only is recovery by an attacker of the symmetric key no longer possible with possession of the recipient's private key, for practical purposes it is not feasible by any other means. Moreover, it is scarcely worth the attacker's effort to crack a symmetric key that is used to encrypt only one message.

Digital Signatures

Digital signature is the third and final component of the public-key cryptosystem imagined by DH, and later proved mathematically by RSA. A digital signature serves three main purposes in digital communication: *message integrity*, *authentication of origin* and *non-repudiation of origin*. These properties prove that a signed message (a) has not been altered by an unauthorized party, (b) originated from its sender and (c) the sender cannot repudiate (a) or (b).[44]

In this scheme, the sender of a message *signs* it using his or her private key, and on receipt the receiver *verifies* the signed message using the signer's public key. If the signature verification fails, then either the private key used to sign the message does not correspond to the public key used to verify it, or the message was altered after it was signed. In either case, the signature is invalid and the message cannot be trusted. In short, a digital signature binds a specific message to the identity of its signer.

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 private decryption key *d*; whereas in the former messages are *signed* with a decryption key *d*, and *verified* with a corresponding encryption key *e*.

---

[42] RSA is still quite relevant for digital signatures, however; mainly as an identity authentication mechanism on the worldwide web (this is the subject of the next section).

[43] AES stands for the *Advanced Encryption Standard*; the current standard for symmetric-key encryption.

[44] Whereas hand-written signatures can be forged, and/or the documents they belong to altered, neither is possible with a properly constructed digital signature.

State Transition Redux

To visualize this, let's revisit the state-transition diagrams we employed to understand encryption. Here, instead of the term *C* occupying the middle state, we now have the term *S*, which stands for *signature*.

(1) $M \rightarrow S \rightarrow M$

As with encryption, we start with some plaintext message *M*, transform it to a signature *S*, and then back again to its original state *M*.

What is different this time is that the first transformation (*M -> S*) needs to be difficult for an attacker, not the second (*S -> M*) (with encryption, it is the second transformation (*C -> M*) that must be difficult for the attacker).

Now, let's substitute the term *S* with its equivalent term M^d.

(2) $M \rightarrow M^d \rightarrow M$

Here, we are raising *M* to the power of the decryption exponent *d* to compute the signature. What makes this transformation difficult for an attacker is that the private exponent *d* used in the transformation is not (or at least should not be) known to anyone but the message's signer.

Next, let's substitute the rightmost *M* from the previous diagram with its equivalent term $(M^d)^e$.

(3) $M \rightarrow M^d \rightarrow (M^d)^e$

Here, the signature M^d is raised to the power of the public encryption exponent *e* to recover the message *M* from the signature M^d.

Finally, let's confine the computations to the group of integers modulo *n*.

(4) $M \rightarrow M^d \bmod n \rightarrow (M^d)^e \bmod n$

Plugging real values into the diagram yields the following transformations:

(5) $5 \rightarrow 5^{43} \bmod 77 \rightarrow (5^{43})^7 \bmod 77$
(6) $5 \rightarrow 26 \rightarrow 5$

If this all looks familiar, it should; it is the very mirror image of encryption. The difference is semantic; with encryption the public exponent *e* is used to transform a plaintext to a *ciphertext*, and with digital signature the private exponent *d* is used to transform a plaintext to a *signature*.[45]

In practice, encryption and digital signature can be combined to ensure the confidentiality, integrity and authenticity of a message in one fell swoop.

Resurrecting the example of encryption from *Figure 5*, assume that Bob has generated a key pair of his own and shared his public key with Alice; that is, he mimics Alice's actions in steps 1 through 8, sending his public key to Alice and keeping his private key secret. Now, Alice and Bob both possess each other's

[45] A signature is really just another name for a ciphertext, but one that is generated using a private key rather than a public key. Thought of another way, every message is some other message's ciphertext, and every ciphertext is itself a valid message. Whereas previously we referred to RSA encryption as a one-way trap-door function, the inversion that permits digital signature makes it a one-way trap-door *permutation*.

public keys, and each their own corresponding private keys. This time, however, before Bob encrypts his message *M*, he signs it with his private key to yield a signature *S*. Bob then encrypts the signature *S* with Alice's public key and transmits the resulting ciphertext to Alice.

On receipt of the ciphertext, Alice reverses the procedure. First, she decrypts the ciphertext using her private key to recover the signature *S*. She then verifies the signature using Bob's public key to recover *M* from *S*.

Again, let's use state transition diagrams to depict this graphically. In the diagrams we distinguish Alice's and Bob's exponents by adding their initials; that is, *eA* and *dA* are Alice's encryption and decryption exponents, and *eB* and *dB* are Bob's encryption and decryption exponents, respectively. Although they have been omitted from the diagrams to reduce conceptual clutter, all the operations are reduced modulo *n*.[46]

First, Bob signs the message *M* using M^{dB}, and then encrypts the signature M^{dB} using $(M^{dB})^{eA}$, which he transmits to Alice.

(1) $M \rightarrow M^{dB} \rightarrow (M^{dB})^{eA}$

Having received the encrypted signature $(M^{dB})^{eA}$ from Bob, Alice recovers the signature M^{dB} using $((M^{dB})^{eA})^{dA}$, and then recovers *M* from M^{dB} using $(M^{dB})^{eB}$.

(2) $(M^{dB})^{eA} \rightarrow ((M^{dB})^{eA})^{dA} \rightarrow M^{dB} \rightarrow (M^{dB})^{eB} \rightarrow M$

Caveats

The RSA-style digital signatures described here are not strong enough to pass cryptographic muster. RSA proved only in concept such schemes were possible, and soon after publication of the paper one was in fact implemented.

For one thing, that a signature can be *verified* in the aforementioned scheme is a dubious assertion. Suppose Alice selects a bit string at random, claiming to some third party it is a signed message from Bob. Alice can then use Bob's public key to recover a completely valid (albeit nonsensical) message from this random signature. But how is the third party to know the signature is not real? Legibility alone is not sufficient proof a message was signed with the private key corresponding to the public key used to verify it.

Here, Alice has performed what is known as an existential forgery attack. In practice, such an attack can be thwarted if the signer applies a cryptographic hash to the message, signs the hash, and sends the signed hash together with the message to the recipient.[47] The signature-message pair is then verified by recovering the signer's hash from the signature using signer's the public key, applying the same cryptographic hash to the original message, and comparing the results of both operations; if they agree, the signature is provably valid.[48]

---

[46] Note that Alice and Bob will each have computed their own *n*, as well.

[47] A *cryptographic hash* is a one-way function without a trap-door; that is, it can be computed in one direction but not the other.

[48] The hashes are usually padded for additional security (see *Full Domain Hash*, or RSA-FDH).

Even with protection from existential forgery, digital signatures are useless if a private key is stolen—or otherwise derived in some fashion—by an attacker. Then the private key can be used to sign any message the owner did not in fact sign. Because of this, it is virtually impossible to prove—in a court of law, for example—that just because a message was signed with a private key, that it was signed by that key's owner.

The Underbelly of Public-Key Cryptography

The public-key cryptosystems of DH and RSA are at once powerful and elegant. Moreover, industrial strength versions of them 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 her private key. Eve has thus broken encryption and, what's more, she has done so without using any math.[49]

Every public-key scheme—whether the secure key agreement of DH, the encryption and digital signature of RSA, or otherwise—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.

Public Key Infrastructure

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 their groundbreaking paper. Acutely aware of the threat MITM posed to their cryptosystem, 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.[50]

---

[49] 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 a weak link.

[50] PKI is not restricted to the CA model described in this section. Other solutions include *Web of Trust*, *Trust on First Use* and other variants offering varying degrees of trust and security. The CA model is interesting and relevant because it is the one used to secure the vast majority of communication on the worldwide web.

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*.[51] 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. On receipt of the certificate, Bob verifies it using the CA's public key. If the verification succeeds, Bob knows the public key contained within it in fact belongs to Alice, and proceeds confidently to encrypt messages to her 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, the CA-based 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.[52]

Going Forward

For all their elegance and power, cryptosystems based on the public-key services described in this paper are not without flaws. This is due in no small part to the complexity of the PKIs we trust to enforce good behavior in a hostile environment. Moreover, techniques based on the arithmetic of cyclic groups have become somewhat dated in the half-century since their introduction, and increasingly find themselves replaced by more modern and powerful ones.[53]

Nevertheless, public-key schemes based on the difficulty of solving the discrete log problem and large integer factorization are the best we have in the internet age. Indeed, they are responsible for securing the vast majority of sensitive communications on the internet today, and achieve this objective with a remarkable degree of success.

The advent of cryptocurrencies, in particular the distributed blockchains on which they are based, 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 databases, 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 identity authentication, lack.[54]

---

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

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

[53] Elliptic curve cryptography (ECC) provides a novel approach of the discrete log problem. ECC emerged as a countermeasure both to more effective attacks on traditional implementations, and more powerful computers to run them on. ECC achieves security levels equivalent to those of RSA with much shorter key lengths.

[54] This alignment of incentives, known as a *consensus algorithm*, works remarkably well in the realm of cryptocurrencies, where curators of blockchains are compensated for being honest with remunerative tokens (e.g., bitcoin).

Meanwhile, quantum computing looms menacingly on the horizon, and threatens the existence of public-key cryptography as we know it. Whereas even today's most powerful computers cannot reverse the one-way functions of the classical public-key cryptography described in this paper—at least not sufficiently fast to make the effort worthwhile—a fit-for-purpose quantum computer could break them in hours or even minutes. There is little doubt that well-funded, state-level actors are attempting to build quantum computers with such capabilities today. The invention of such a computer would render classical public-key cryptography instantly obsolete, and enable its inventors to break the cryptosystems on which the world's security-critical computing infrastructure largely relies.