Asymmetric cryptography[1]

Previously we saw symmetric-key cryptography, where Alice and Bob share a secret key K. However, symmetric-key cryptography can be inconvenient to use, because it requires Alice and Bob to get together in advance to establish the key somehow. *Asymmetric cryptography*, also known as *public-key cryptography*, is designed to address this problem.

In a public-key cryptosystem, the recipient Bob has a publicly available key, his *public key*, that everyone can access. When Alice wishes to send him a message, she uses his public key to encrypt her message. Bob also has a secret key, his *private key*, that lets him decrypt these messages. Bob publishes his public key but does not tell anyone his private key (not even Alice).

Public-key cryptography provides a nice way to help with the key management problem. Alice can pick a secret key K for some symmetric-key cryptosystem, then encrypt K under Bob's public key and send Bob the resulting ciphertext. Bob can decrypt using his private key and recover K. Then Alice and Bob can communicate using a symmetric-key cryptosystem, with K as their shared key, from there on.

Public-key cryptography is a remarkable thing. Consider a function that, for a given public key, maps the message to the corresponding ciphertext. In a good public-key cryptosystem, this function must be easy to compute, and yet **very hard to invert**. In other words, it must form a *one-way function*: a function f such that given x, it is easy to compute $f(x)$, but given y, it is hard to find a value x such that $f(x) = y$. We need the computational equivalent of a process that turns a cow into hamburger: given the cow, you can produce hamburger, but there's no way to restore the original cow from the hamburger. It is by no means obvious that it should be possible to accomplish this, but it turns out it is, as we'll soon discuss.

The known methods for public-key cryptography tend to rely heavily upon number theory, so we begin with a brief number theory refresher, and then develop an encryption algorithm based on public-key cryptography.

[1] Material courtesy of David Wagner

1 Algorithms for modular arithmetic

1.1 Simple modular arithmetic

Two n-bit integers can be added, multiplied, or divided by mimicking the usual manual techniques taught in elementary school. For addition, the schoolkid's algorithm takes a constant amount of time to produce each bit of the answer, since each such step only requires dealing with three bits—two input bits and a carry—and anything involving a constant number of bits takes $O(1)$ time. The overall time to add two n-bit integers is therefore $O(n)$, or linear in the bitlength of the integers. Multiplication and division take $O(n^2)$ time, i.e., quadratic time. Also recall that n, the number of bits it takes to represent an integer a in binary, satisfies $n \leq \lceil \log_2 a \rceil$.

Recall that $a \bmod p$ is the remainder of the number a modulo p. For instance, $37 \bmod 10 = 7$.

Modular arithmetic can be implemented naturally using addition, multiplication, and division algorithms. To compute $a \bmod p$, simply return the remainder after dividing a by p. By reducing all inputs and answers modulo p, modular addition, subtraction, and multiplication are easily performed. These operations can all be performed in $O(\log^2 p)$ time, since the numbers involved never grow beyond p and therefore have size at most $\lceil \log_2 p \rceil$ bits.

1.2 Modular exponentiation

Modular exponentiation is the task of computing $a^b \bmod p$, given a, b, and p.

A naive algorithm for modular exponentiation is to repeatedly multiply by a modulo p, generating the sequence of intermediate products $a^2 \bmod p$, $a^3 \bmod p$, $a^4 \bmod p$, $\ldots$, $a^b \bmod p$. Each intermediate product can be computed from the prior one with a single modular multiplication, which takes $O(\log^2 p)$ time to compute, so the overall running time to compute $a^b \bmod p$ products is $O(b \log^2 p)$. This is linear in b and thus exponential in the size (bitlength) of b—really slow when b is large.

There is a better way to do it. The key to an efficient algorithm is to notice that the exponent of a number a^j can be doubled quickly, by multiplying the number by itself. Starting with a and squaring repeatedly, we get the powers $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log_2 b \rfloor}}$, all modulo p. Each takes just $O(\log^2 p)$ time to compute, and they are all we need to determine $a^b \bmod p$: we just multiply together an appropriate subset of them, those corresponding to ones in the binary representation of b. For instance,

$$a^{25} = a^{11001_2} = a^{10000_2} \cdot a^{1000_2} \cdot a^{1_2} = a^{16} \cdot a^8 \cdot a^1.$$

This repeated squaring algorithm is shown in Algorithm 1. The overall running time is $O(\log^2 p \log b)$. When p and b are n-bit integers, the running time is *cubic* in the input size. This is efficient enough that we can easily perform modular exponentiation on numbers that are thousands of bits long.

Algorithm 1 ModExp1(a, b, p): modular exponentiation using repeated squaring.

Require: a modulus p, a positive integer $a < p$, and a positive exponent b
Ensure: the return value is $a^b \bmod p$

Let $b_{n-1} \cdots b_1 b_0$ be the binary form of b, where $n = \lceil \log_2 b \rceil$.
// Compute the powers $t_i = a^{2^i} \bmod s$.
$t_0 := a \bmod p$
for $i := 1$ to $n - 1$ **do**
    $t_i := t_{i-1}^2 \bmod p$
end for
// Multiply together a subset of these powers.
$r := 1$
for $i := 0$ to $n - 1$ **do**
    if $b_i = 1$ **then**
        $r := r \times t_i \bmod p$
    end if
end for
return r

1.3 Selecting Large Primes

In the material to follow, we will be working with very large primes: primes that are thousands of bits long. Let's look at how to generate a random n-bit prime.

It turns out that it's easy to test whether a given number is prime. Fermat's Little Theorem forms the basis of a kind of litmus test that helps decide whether a number is prime or not: to test if a number M is prime, we select an a mod M at random and compute a^{M-1} mod M. If the result is not 1, then by Fermat's theorem it follows that M is not a prime. If on the other hand the result is 1, then this provides evidence that M is indeed prime. With a little more work this forms the basis of an efficient probabilistic algorithm for testing primality.

For the purpose of selecting a random large prime (several thousand bits long), it suffices to pick a random number of that length, test it for primality, and repeat until we find a prime of the desired length. The prime number theorem tell us that among the n-bit numbers, roughly a $\frac{1.44}{n}$ fraction of them are prime. So after $O(n)$ iterations of this procedure we expect to find a prime of the desired length.

2 Diffie-Hellman key exchange

Now we're ready to see our first public-key algorithm. Suppose Alice and Bob are on opposite sides of a crowded room. They can shout to each other, but everyone else in the room will overhear them. They haven't thought ahead to exchange a secret key in advance. How can they hold a private conversation?

It turns out there is a clever way to do it, first discovered by Whit Diffie and Marti Hellman in the 1970s. In high-level terms, the Diffie-Hellman key exchange works like this.

Alice and Bob first do some work to establish a few parameters. They somehow agree on a large prime p. For instance, Alice could pick p randomly and then announce it so Bob learns p. The prime p does not need to be secret; it just needs to be very large. Also, Alice and Bob somehow agree on a number g in the range $1 < g < p - 1$. The values p and g are parameters of the scheme that could be hardcoded or identified in some standard; they don't need to be specific to Alice or Bob in any way, and they're not secret.

Then, Alice picks a secret value a at random from the set $\{0, 1, \ldots, p-2\}$, and she computes $A = g^a \bmod p$. At the same time, Bob randomly picks a secret value b and computes $B = g^b \bmod p$. Now Alice announces the value A (keeping a secret), and Bob announces B (keeping b secret). Alice uses her knowledge of B and a to compute

$$S = B^a \bmod p.$$

Symmetrically, Bob uses his knowledge of A and b to compute

$$S = A^b \bmod p.$$

Note that

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \pmod p,$$

so both Alice and Bob end up with the same result, S. Finally, Alice and Bob can use S as a shared key for a symmetric-key cryptosystem (in practice, we would apply some hash function to S first and use the result as our shared key, for technical reasons).

The amazing thing is that Alice and Bob's conversation is entirely public, and from this public conversation, they both learn this secret value S—yet eavesdroppers who hear their entire conversation cannot learn S. As far as we know, there is no efficient algorithm to deduce $S = g^{ab} \bmod p$ from $A = g^a \bmod p$, $B = g^b \bmod p$, g, and p. (If there were an efficient algorithm to recover S from A, B, p, g, then this scheme would be insecure, because an eavesdropper could simply apply that algorithm to what she overhears.) In particular, the fastest known algorithms for solving this problem take $2^{cn^{1/3}(\log n)^{2/3}}$ time, if p is a n-bit prime. For $n = 2048$, these algorithms are far too slow to allow reasonable attacks.

The security of Diffie-Hellman key exchange relies upon the fact that the following function is one-way: $f(x) = g^x \bmod p$. In particular, it is easy to compute $f(\cdot)$ (that's just modular exponentiation), but there is no known algorithm for computing $f^{-1}(\cdot)$ in any reasonable amount of time.

Here is how this applies to secure communication among computers. In a computer network, each participant could pick a secret value x, compute $X = g^x \bmod p$, and publish X for all time. Then any pair of participants who want to hold a conversation could look up each other's public value and use the Diffie-Hellman scheme to agree on a secret key known only to those two parties. This means that the work of picking p, g, x, and X can be done in advance, and each time a new pair of parties want to communicate, they each perform only one modular exponentiation. Thus, this can be an efficient way to set up shared keys.

Here is a summary of Diffie-Hellman key exchange:

- **System parameters:** a 2048-bit prime p, a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.
- **Key agreement protocol:** Alice randomly picks a in the range $0 \ldots p-2$ and sends $A = g^a \bmod p$ to Bob. Bob randomly picks b in the range $0 \ldots p-2$ and sends $B = g^b \bmod p$ to Alice. Alice computes $K = B^a \bmod p$. Bob computes $K = A^b \bmod p$. Alice and Bob both end up with the same secret key K, yet as far as we know no eavesdropper can recover K in any reasonable amount of time.

3 Caveat: Don't try this at home!

A brief warning is in order here. You've now seen the conceptual basis underlying public-key algorithms that are widely used in practice. However, if you should need a public-key encryption algorithm, *don't implement your own based on the description here.* The discussion has omitted some nitty-gritty implementation details that are not all that relevant at the conceptual level, but are essential for robust security. Instead of implementing these algorithms yourself, you should just use a well-tested cryptographic library or protocol, such as TLS or PGP.

4 What's the catch?

This all sounds great—almost too good to be true. We have a way for a pair of strangers who have never met each other in person to communicate securely with each other. Unfortunately, it is indeed too good to be true. There is a slight catch. The catch is that if Alice and Bob want to communicate securely using these public-key methods, they need some way to securely learn each others' public key. The algorithms presented here don't help Alice figure out what is Bob's public key; she's on her own for that.

You might think all Bob needs to do is broadcast his public key, for Alice's benefit. However, that's not secure against *active attacks.* Attila the attacker could broadcast his own public key, pretending to be Bob: he could send a spoofed broadcast message that appears to be from Bob, but that contains a public key that Attila generated. If Alice trustingly uses that

public key to encrypt messages to Bob, then Attila will be able to intercept Alice's encrypted messages and decrypt them using the private key Attila chose.

What this illustrates is that Alice needs a way to obtain Bob's public key through some channel that she is confident cannot be tampered with. That channel does not need to protect the *confidentiality* of Bob's public key, but it does need to ensure the *integrity* of Bob's public key. It's a bit tricky to achieve this.

One possibility is for Alice and Bob to meet in person, in advance, and exchange public keys. Some computer security conferences have "key-signing parties" where like-minded security folks do just that. In a similar vein, some cryptographers print their public key on their business cards. However, this still requires Alice and Bob to meet in person in advance. Can we do any better? We'll soon see some methods that help somewhat with that problem.