Fall 2021

Section 1: TTh 3:30pm - 4:45pm - JFSB B092 (changed from MARB 130)

Project 4: Diffie-Hellman

Objectives

- Gain an understanding of modular arithmetic and modular exponentiation.
- Learn how to use a BIGNUM library.
- Learn how asymmetric cryptography can be used to establish secure communications over an insecure channel

Background

Diffie-Hellman key exchange and RSA public-key encryption both rely on functions like this:

$$(g^a \bmod p)^b \bmod p$$

The Diffie-Hellman key-exchange algorithm allows two people, Alice and Bob, to agree upon a number (to use as a key for encryption) without Eve knowing what it is, even though she sees everything that passes between Alice and Bob. It relies on the fact that:

$$(g^a)^b = (g^b)^a = g^{ab} = g^{ab}$$

The first equation above has the same property. Here's how the key exchange works:

Alice picks a generator g, a prime p, and a large random number a. She sends g, p, and the resulting value of $g^a \bmod p$ to Bob, but not a by itself.

Bob picks a large random value b, and sends $g^b \bmod p$ back to Alice. Eve, watching each exchange, now has g, p, $g^b \bmod p$ and $g^a \bmod p$, but she can't get a or b from these. We hope. It would require that Eve know how to efficiently compute discrete logarithms, and the security of this algorithm (as well as that of RSA and most other public key algorithms) depends on nobody figuring out how to do that.

Bob takes the $g^a \bmod p$ that Alice sent him and raises it to b, then takes $\bmod p$, giving him $(g^a \bmod p)^b \bmod p$. Alice does the same thing and ends up with $(g^b \bmod p)^a \bmod$ - which is the same number. Alice and Bob can now use this number as a password for encrypting future communications.

Since a, b and p are usually very large numbers (hundreds of digits long), we need a fast way to compute $g^a \bmod p$. That's where **modular exponentiation** comes in.

You can use the Linux desktop calculator, dc, to do modular exponentiation for checking your work. For example:

```
% dc
2 8 255 |p
1
q
%
```

This computes $2^8 \bmod 255$. You can search for "man dc" for more information on dc.

Modular Exponentiation

The term modular exponentiation refers to the function:

$$g^a \bmod p$$

We need to compute this function efficiently to use public key cryptography of the Diffie-Hellman or RSA variety.

If we calculate g^a as:

$$g^a = g \times g \times g \times g \times \ldots \times g$$

we'll spend a lot of time multiplying if a is large (which it usually is; often it's greater than the number of particles in the universe!). Here's how to get there faster.

Since we're dealing with computers here, let's assume we have a binary representation of a. If $a = 42$, we'd write it as

$$42 = 101010 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

With $a = 42$, this means that:

$$g^a = g^{2^5+2^3+2^1} = g^{32} * g^8 * g^2$$

Now we just need g^{32}, g^8 and g^2. But check this out:

$$g^2 * g^2 = g^4$$

$$g^4 * g^4 = g^8$$

$$g^8 * g^8 = g^{16}$$

$$g^{16} * g^{16} = g^{32}$$

So by progressively multiplying the products by themselves, we can get g^{2^n} with just $n - 1$ multiplications. Since there are only about $log_2(a)$ bits in a, we can find g^a with at most $log_2(a) - 1 + log_2(a)$ multiplication operations. (The second $log_2(a)$ comes from where we multiplied $g^{32} \times g^8 \times g^2$ above).

So how does the modulus play into all of this? Well, it turns out that you can distribute the operation into the multiplication without changing the result, leaving you with smaller numbers to multiply, like this:

$$((g^{32} \bmod p) \times (g^8 \bmod p) \times (g^2 \bmod p)) \bmod p = g^{42} \bmod p$$

You can do the same thing when computing g^{2^n}. Why does it work?

Here's an example:

$$(3 \times 4 \times 5) \bmod 7 =?(((3 \times 4) \bmod 7) \times 5) \bmod 7$$

Let's derive it:

$$\begin{aligned}
(3 \times 4 \times 5) \bmod 7 &= (12 \times 5) \bmod 7 \\
(3 \times 4 \times 5) \bmod 7 &= ((7 + 5) \times 5) \bmod 7 \\
(3 \times 4 \times 5) \bmod 7 &= (7 \times 5 + 5 \times 5) \bmod 7 \\
(3 \times 4 \times 5) \bmod 7 &= (5 \times 5) \bmod 7 \\
(3 \times 4 \times 5) \bmod 7 &= ((12) \bmod 7) \times 5) \bmod 7 \\
(3 \times 4 \times 5) \bmod 7 &= ((3 \times 4) \bmod 7) \times 5) \bmod 7
\end{aligned}$$

Requirements

In this lab you will use Diffie-Hellman key exchange to establish a secret key to use for communication with a server. Specifically:

1. Select a and p. Both must be at least 500 bits long. To be cryptographically strong, p must be a randomly generated prime, and $(p - 1)/2$ should also be prime (although certain other combinations of p and the generator are also secure). a should be a cryptographically safe random number (i.e., not the result of a plain-vanilla rand() function). Reading from /dev/urandom on a Linux machine should be adequate to produce a. Other methods of generating random numbers will also be acceptable. Generating primes is a little more complicated. You may use a library (such as OpenSSL) to generate p.
2. Write a routine to compute $g^a \bmod p$, for $g = 5$. You may use a `bignum` library's multiply and modulus functions, but no composite functions like modular exponentiation or modular multiplication.
3. Using the website at https://grader.cs465.byu.edu/diffie-hellman select Debug Mode, and input p and $g^a \bmod p$.

where both values are expressed in decimal. In Debug mode, you will be able to see the server choose a value for b and compute its value of $g^b \bmod p$. It will also compute g^{ab}. You can hard-code this value of g^b and compute g^{ab} to be sure they match. You can also hard-code b to be sure you can compute g^b the same as the server.

Resources

Python ints automatically grow into bigints as needed with no user intervention.

If you are coding in Java then you will find the BigInteger object very useful.

OpenSSL includes a great bignum library, including generating primes. Note that you must link in the openssl libraries using `-lcrypto` when compiling. For example to compile your c program named `dh.c` with openssl, type `gcc -o dh dh.c -lcrypto`.

Here is some sample OpenSSL BIGNUM code.

OpenSSL can be difficult to install on Windows machines for VC++. Here are some helpful hints on installing the OpenSSL library on your home computer.

Passing

1. Using your code, choose a value for p and a and then hard-code these into your code. Compute $g^a \bmod p$.
2. Go to the the following web site: https://grader.cs465.byu.edu/diffie-hellman and enter your values for p and $g^a \bmod p$. Select Passoff Mode and Submit.
3. On the following screen, you will see a value for the server's $g^b \bmod p$ (the server calls it $g^t \bmod p$). Hard code this value into your code, and compute g^{ba} using your hard-coded a. The value of g^{ab} is your computed secret key.
4. Copy the file shown into a file called `foo`. Use the openssl command given on the site to decrypt the file and supply your secret key as the decryption password. This will create a file called `bar`.
5. Copy and paste the contents of `bar` into the form and submit.

If you do this correctly, you will get a message that says

```
Good Job! You've completed the lab.
```

Submission

Submit a zip or gzip file that contains:

1. Your source code.
2. Instructions for compiling (if needed) and running your code.
3. A screenshot of your successful submission to the website.