# Secure Communication Protocols

**Abstract**

In this assignment, we implement a secure communication protocol consisting of three consecutive steps. First, we use Diffie-Hellman[1][2] key exchange to craft a shared private key between two actors using public key cryptography. We then use an implementation of the Blum Blum Shub[3] algorithm to generate a cryptographically strong pseudo-random shared private key, using the previously exchanged key as seed. Finally, we use the generated random key to encrypt and decrypt messages between the two actors using AES[4] ciphers.

## 1 Introduction

Our main goal is to be able to communicate secret data over an unsecure communication channel. To do so, we have to encrypt our data before sending it, and decrypt it after recieving it, so that it only appears encrypted on the communication channel. To perform these encryptions, we want to use a symmetric cipher allowing us to efficiently compute ciphertext from plaintext and vice versa. Before using symmetric encryption, we have to solve the key exchange problem, because we still need to communicate the private key between the two actors over an unsecure channel. One answer to this first problem is to use public-key cryptography, and more specifically the Diffie-Hellman key exchange protocol.

### 1.1 Diffie-Hellman

This key exchange scheme allows us to create a shared private key by only communicating public data. It relies on the computational difficulty of computing discrete logarithms. It has been formulated in 1976 by Whitfield Diffie and Martin Hellman[1][2]. We use it to generate a first shared private key between two actors, Alice and Bob.

### 1.2 Blum Blum Shub

To increase the strength of the exchanged key, we use it as a seed to a cryptographically strong pseudo-random number generator. The chosen stream generator is Blum Blum Shub[3], which outputs the least significant bit of $x_{n+1}$ at each step of the following sequence.

$$x_{n+1} = x_n^2 \bmod M \qquad \text{with } x_0 = \text{seed}, M \text{ the product of two large primes } p \text{ and } q$$

### 1.3 Advanced Encryption Standard

To encrypt their messages, Alice and Bob can now use a symmetric cipher with their shared private key. Here, we use AES in Electronic Code Book mode. Now that Alice and Bob exchanged their keys, they could use any symmetric cipher and mode of operation.

Using these cryptographic primitves, we implement a secure communication protocol allowing Alice and Bob to communicate data privately.

## 2 Design and Implementation

### 2.1 Diffie-Hellman

The python implementation of the Diffie-Hellman key exchange is fairly straightforward. There are two main steps in this scheme : creating a public key from our private key, and combining the other party's public key with our private key to generate the shared private key. These two steps are represented by the two main modes of operation of `keygen.py` : generate and merge, to be passed to the `--mode` argument. When running the script in `--mode generate`, the main function used is the following.

```
def pubkeygen(prime, root, secret):
    assert(is_prime(prime))
    assert(is_primitive_root(root, prime))

    return pow(root, secret, prime)
```

And when used in `--mode merge`, the main computation is done in this function.

```
def shared_secret_key(secret, other_public_key, prime):
    return pow(other_public_key, secret, prime)
```

Note that in both functions, we use the `pow()` method with three arguments rather than exponentiating and then taking the modulo. This is because on large numbers, the `pow()` method is significantly faster at computing the modulo than the raw computation of exponentiation followed by division. This is demonstrated in Figure 2. This program is designed to accept decimal or hexadecimal representation of integers for the `-{}-secret`, `--prime`, `--root`, and `--public` arguments.

The rest of the script is mostly dedicated to argument parsing, parameters loading and tests. The tests are run when using the script in `--mode test`. The test data comes from the RFC 5114[5] memo which describes standard Diffie-Hellman groups and their associated test data. Results of the tests can be found in Table 1. The default group used by the script if no `--prime` and `--root` are passed is the 2048-bit MODP Group with 256-bit Prime Order Subgroup which can be found in the memo.

Further specification for the other command line arguments can be found in the `README.pdf` file.

### 2.2 Blum Blum Shub

The main computation of the BBS algorithm is also quite simple, as bits are generated incrementally by taking the least significant bit of each modular exponentiation. The generated bits are then merged together and interpreted as a binary number.

```
def generate_random(seed, size):

    bits = []

    for _ in range(size):
        seed = pow(seed, 2, M)
        bits.append(bin(seed)[-1])

    return int(''.join(bits), 2)
```

Although testing for statisical randomness usually involves a series of tests[6], we can visually check for pseudo-randomness by looking at the bitmap created by the generated bits. If we do not identify any regular pattern, we can assume that the randomness is reasonable for the key to be secure enough. This bitmap test can be seen in Figure 1.

## 2.3 Advanced Encryption Standard

The implementation of a symmetric cipher is not part of this assignment as it has already been done in Assignment 1. We choose AES for its ingenuity and subsequent popularity, and we use the Electronic Code Book mode of operation so that we do not have to transfer initialization vectors or nonces with the message. In a real-world application, we would prefer to use another mode as ECB is semantically insecure. Given a plaintext, ECB will always produce the same ciphertext each time. For a small amount of short text messages, this mode of operation is secure enough for our protocol.

```python
def encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    if len(plaintext) % 16 != 0:
        plaintext = pad(plaintext, 16)
    ciphertext = cipher.encrypt(plaintext)
    return ciphertext
```

For encryption, we have to pad the plaintext if it does not fit the block size (16 bytes, 128 bits) as AES is a block cipher. During the decryption process, we unpad the decrypted ciphertext in the similar way.

```python
def decrypt(ciphertext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    plaintext = cipher.decrypt(ciphertext)
    if len(plaintext) % 16 != 0:
        plaintext = unpad(plaintext, 16)
    return plaintext
```

Although most of the work here is done by an external library[7], we can still test the global implementation by using NIST test data[8]. This is done in Table 2.

## 2.4 Putting it all together

Now that we have seen how the different primitives are implemented, we can combine them to create the secure communication protocol we were seeking. To demonstrate how this can be done, a shell script describing a usecase is given. The file `demo.sh` serves as an example of how these primitives can be used in a modular way. Users may chose to run the programs quietly, or in verbose mode to display the parameters used. They may chose to redirect output to files. They may chose to change the parameters of the different primitives. This demo shell script is POSIX-compliant and should run on any reasonable shell.

The other shell script provided, `test.sh`, allows to run the implementation tests of the different primitives.

# 3 Test Results

Table 1 shows the equivalence of test data from RFC 5114[5] and keys generated by our own implementation. The test data can be found in the `files/2048-bit MODP Group` folder.

Figure 2 shows the speed comparison of modular exponentiation methods. We observe exponential time complexity as the exponent gets larger using naive computation, whereas `pow()` remains nearly constant time.

| Initiator | Public key equivalence | Shared private key equivalence |
|---|---|---|
| A | ✓ | ✓ |
| B | ✓ | ✓ |

Table 1: Comparing IETF test data to our implementation

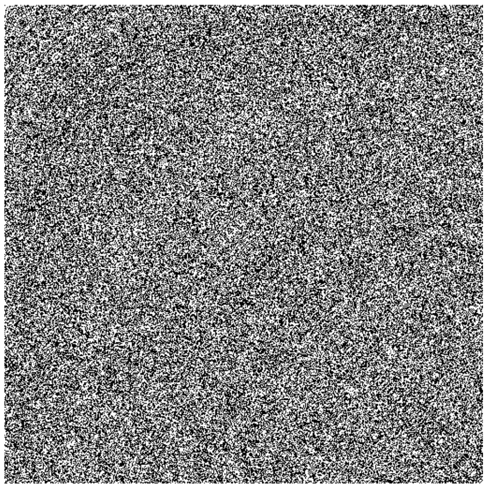

Figure 1: Bitmap representation of a generated pseudo-random number

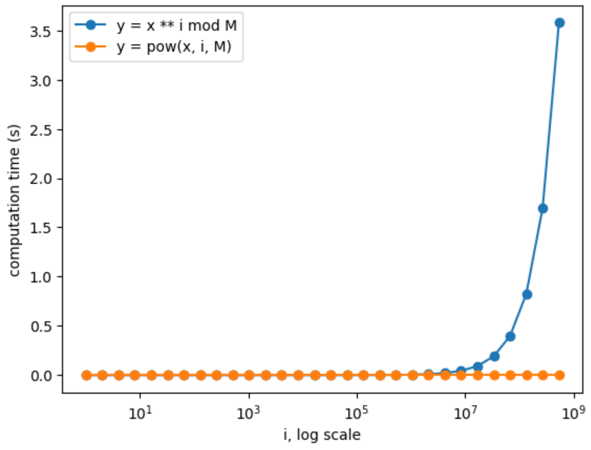

Figure 2: Modular exponentiation speed comparison

Table 2 shows the equivalence between NIST test data[8] and data generated with our implementation using pycryptodome's AES cipher. The keys, ciphertexts and plaintexts used can be found in the `files/AES_test_data` folder.

| Key Length | Plaintext equivalence | Ciphertext equivalence |
|---|---|---|
| 128 bits | ✓ | ✓ |
| 192 bits | ✓ | ✓ |
| 256 bits | ✓ | ✓ |

Table 2: Comparing NIST test data to our implementation

## 4 Discussion

The key generation using Diffie-Hellman with a 2048-bit group is pretty much unbreakable with the current technology as it is estimated that an academic team can break a 768-bit prime and that a nation-state can break a 1024-bit prime[9]. This is arguably the most secure part of our protocol.

With a 2048-bit group, the exponent could be as large as $(2^{2048} - 1) \sim 10^{616}$, which would take roughly $10^{170}$ years to compute with naive exponentiation. The `pow()` method is not only convenient, it is required for the program to complete.

Blum blum shub is slow and not practically secure (https://crypto.stackexchange.com/questions/3454/blum-blum-shub-vs-aes-ctr-or-other-csprngs)

ECB not secure https://blog.filippo.io/the-ecb-penguin/

## 5 Conclusion

## References

[1] Whitfield Diffie and Martin E. Hellman. New directions in cryptography, 1976.

[2] S. L. Graham, R. L. Rivest, and Ralph C. Merkle. Secure communications over insecure channels, 1978.

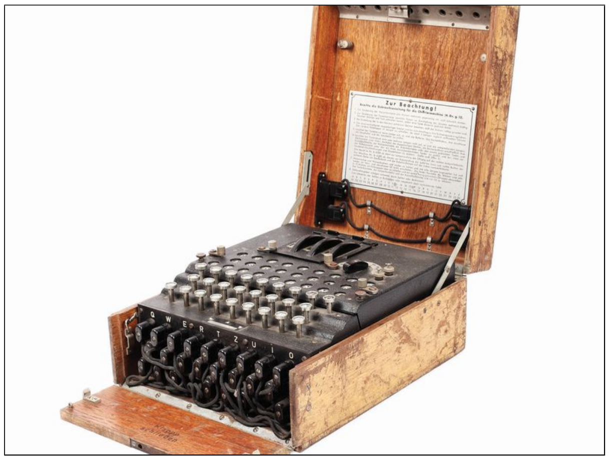

Figure 3: Original image

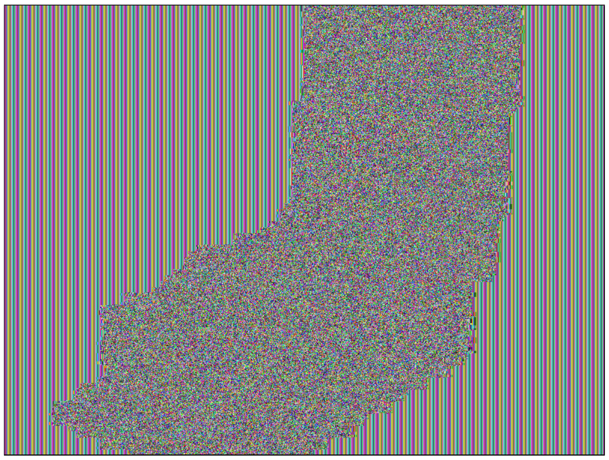

Figure 4: Encrypted image using AES-ECB

[3] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986. doi: 10.1137/0215025. URL https://doi.org/10.1137/0215025.

[4] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.

[5] Matt Lepinski and Stephen Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114, January 2008.

[6] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2001.

[7] Helder Eijs and open source contributors. Pycryptodome's documentation. https://www.pycryptodome.org/en/latest/index.html, 2019.

[8] Computer Security Division, Information Technology Laboratory, National Institute of Standards, Technology, and Department of Commerce. Example values - cryptographic standards and guidelines. https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

[9] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015.