



GreenCoin

A decentralized digital currency.

A Computer Science Final Project

By Michael Kuperfish Steinberg

I.D. 214288912

HaKfar HaYarok 2021

Advisor: Yooda Or

Student Contact Information:

Name: Michael Kuperfish Steinberg (מיכאל קופרפיש שטיינברג)

I.D.: 214288912

Date of Birth: 01/06/2003

Address: Oppenheimer 6, Tel Aviv

Tel: +972-58-676-2020

Email: m.kuper.steinberg@gmail.com

General Information:

School: הכפר הירוק ע"ש לוי אשכול

School Tel: 03-645-5666

Field: Computer Science

Study Units: 5 Units

Advisor Contact Information:

Advisor: Yooda Or (יהודה אור)

I.D.: 023098007

Tel: +972-50-734-4457

Email: yooda@gmail.com

Address: HaKerem 3, Tel Aviv

Academic Degree: MA Engineer, Technion Certified Engineer, Microsoft Certified,
Academic transfer to Computer Science on behalf of the country since the year 2000.

Workplaces: Weizmann Institute of Technology, John Bryce College, HaKfar HaYarok,
Youth Engineering College of Computer Science.

Table of contents:

Introduction	5
Theoretical Background	10
Cryptography	12
Hash Functions	13
SHA-256 - Secure Hash Algorithm - 256	14
Code	14
SHA_OPs.h + SHA_OPs.c	14
SHA_Rotate.h + SHA_Rotate.c	15
SHA256.h + SHA256.c	16
DSA - Digital Signature Algorithm	25
Domain Parameters	26
Signature	27
Generation	27
Verification	28
Bitwise Math	29
Code	31
BNMath.h + BNMath.c	31
Code	65
DSA.h + DSA.c	65
BlockChain	82
Transactions	83
Signing & Verification	85
Signing	86
Verification	87
Code	88
Transaction.h + Transaction.c	88
Blocks	101
Mining	101
Wallets	101
Base-64 Encoding	101
Code	101
Base64.h + Base64.c	101
Network	106
TCP	106
P2P	106
Works Cited	107

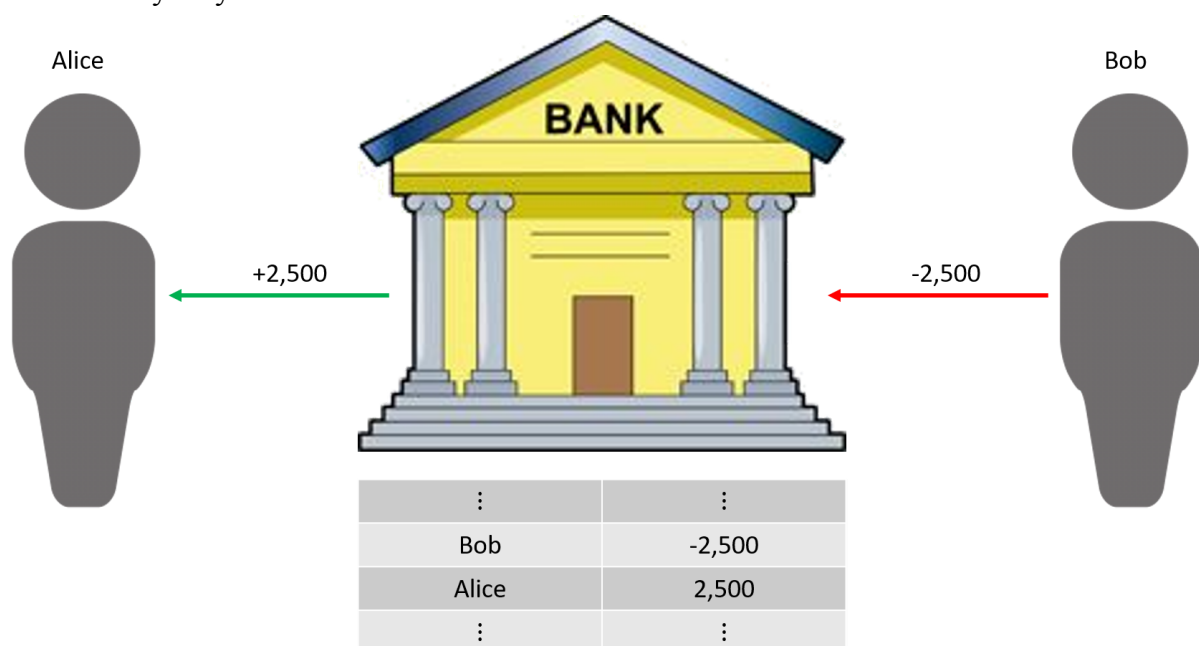
All code can be found on GitHub: <https://github.com/Michael-K-Stein/GreenCoin>

Introduction

Digital Currencies are a complete parallel to fiat currencies (government issued money). Each digital currency creates its own economy which hosts transactions using that currency, similarly to how John could use his salary, which was paid in United States Dollars (USD), and dine at a local restaurant, paying for the meal in USD, John could have been paid in GreenCoins and have done the exact same (assuming the restaurant accepts GreenCoins).

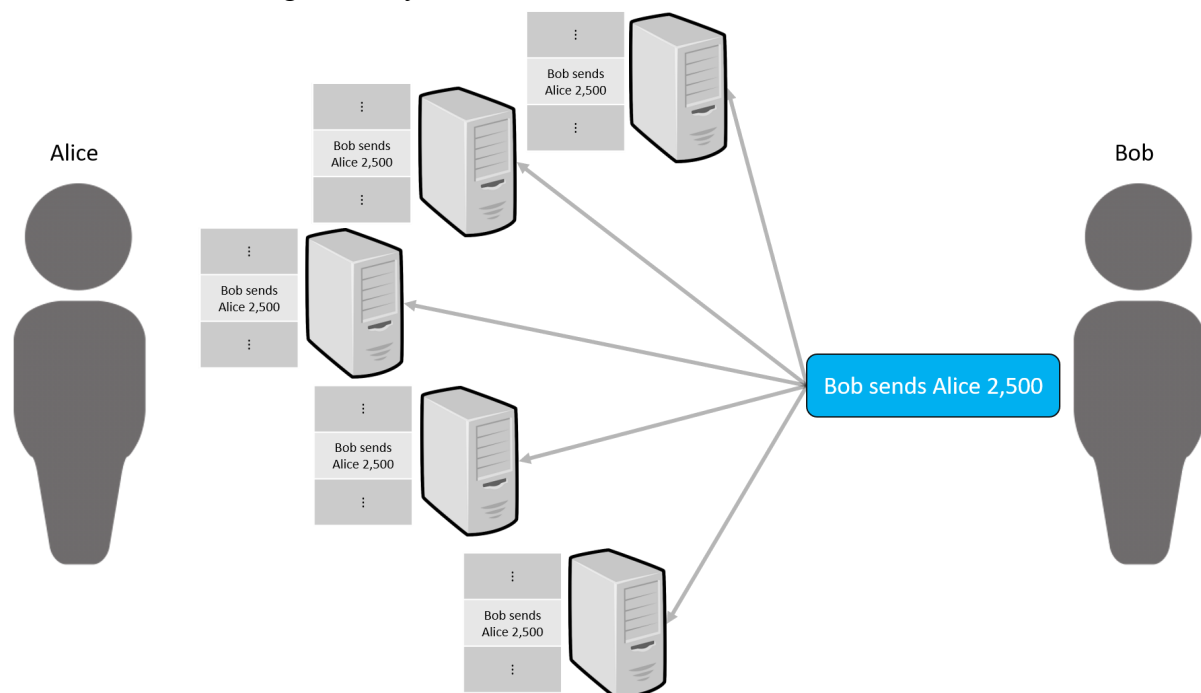
However, unlike traditional currency and banking systems, digital currencies are generally minted and controlled very differently. In traditional economies, a globally trusted financial institution such as a bank would record and store transactions between accounts, and thus conclude how much money each account has and can spend.

Assume Bob sends Alice \$2,500. Bob would send the transaction to the bank who would verify that Bob has the balance / credit to pay the transaction. Then the bank would deduce the 2,500 from Bob's account and credit Alice's account the same amount. This transaction would then be recorded on a ledger which the bank keeps. When either Alice or Bob go to the bank to withdraw their balance the bank can use the ledger to calculate how much money they each have.



The same transaction using a decentralized digital currency such as GreenCoin would not go through a central bank. Instead, Bob would broadcast that he wishes to send Alice 2,500GC. Mining nodes on the network would look at the ledger history to verify that Bob has enough coins and that the transaction is not fraudulent, and if they deem the transaction

valid, add it to the ledger history.



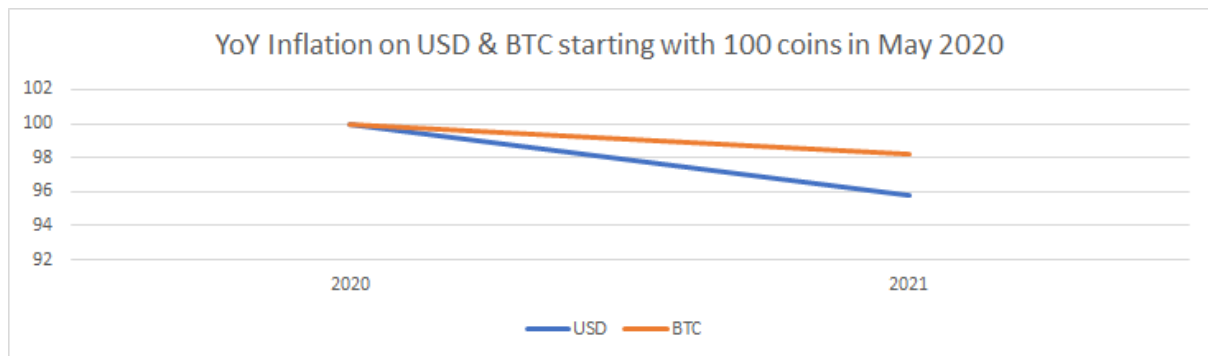
Bob and Alice are now not dependent on the bank, since assuming there are many nodes connected to the network, plenty of servers can confirm that the transaction was made, and Alice can now use the 2,500GC freely. One of the main reasons Bitcoin, the first digital currency, was made was due to the 2008 financial crisis when many banks and other financial institutions had to be bailed out by governments, on the taxpayers dime. When a bank becomes insolvent, meaning it has more liabilities than assets (net negative), it will not pay back its depositors (at least not in full).

Assume it is 2008 and Alice's bank, which processed Bob's transaction (see above), declares bankruptcy. Alice will rush to the bank to try to withdraw her money, but will unfortunately only receive pennies on the dollar. Alice had absolutely no control over the bank and the financial sector's reckless actions which led to the crisis, but she will nevertheless lose the \$2,500 which she had in the bank.

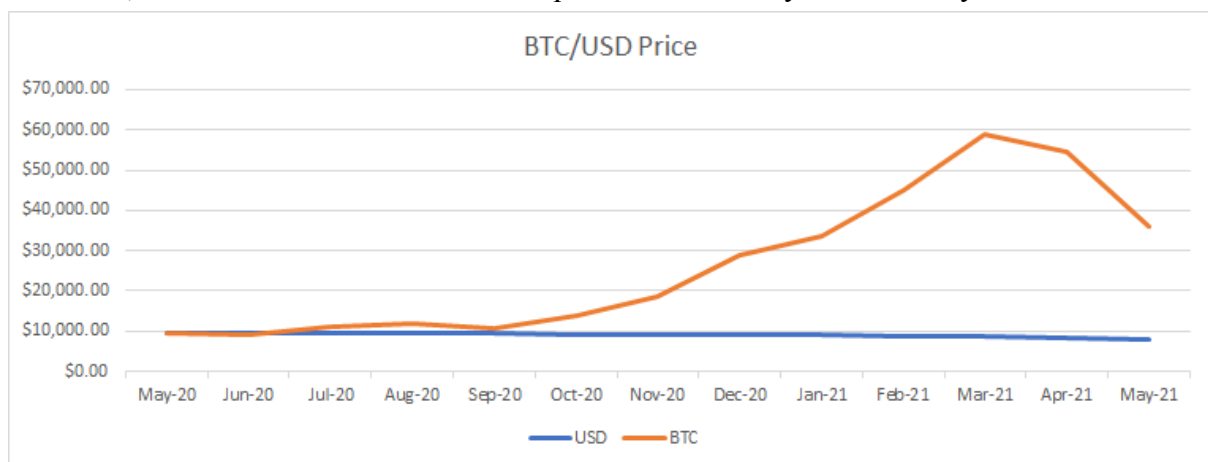
On the contrary, had Alice and Bob done the transaction through a digital currency, no single entity could make the system "crash". No matter how deep in debt a single validator (miner) is, as long as over 50% of validators hold a correct ledger history, no money will ever be lost. Moreover, in most digital currencies, an account cannot hold a negative value, and therefore, there is never any debt.

Furthermore, while the USD is solely controlled by the United States Government, no-one controls a decentralized digital currency. Therefore, hyperinflation is not a hazard. There is no government to wake up and decide to pass a trillion dollar military contracts bill, which will increase inflation. Digital currencies, either have no inflation, meaning all coins are minted at genesis and then distributed, or have a completely predictable inflation rate, which is generally capped (such as Bitcoin's 21,000,000 coin capacity). In situations where there is little trust in the government, digital currencies can be a clear solution. For a real world example, this is a graph assuming you held 100USD and 100BTC, how much would

you have after inflation (from May 2020 until May 2021):

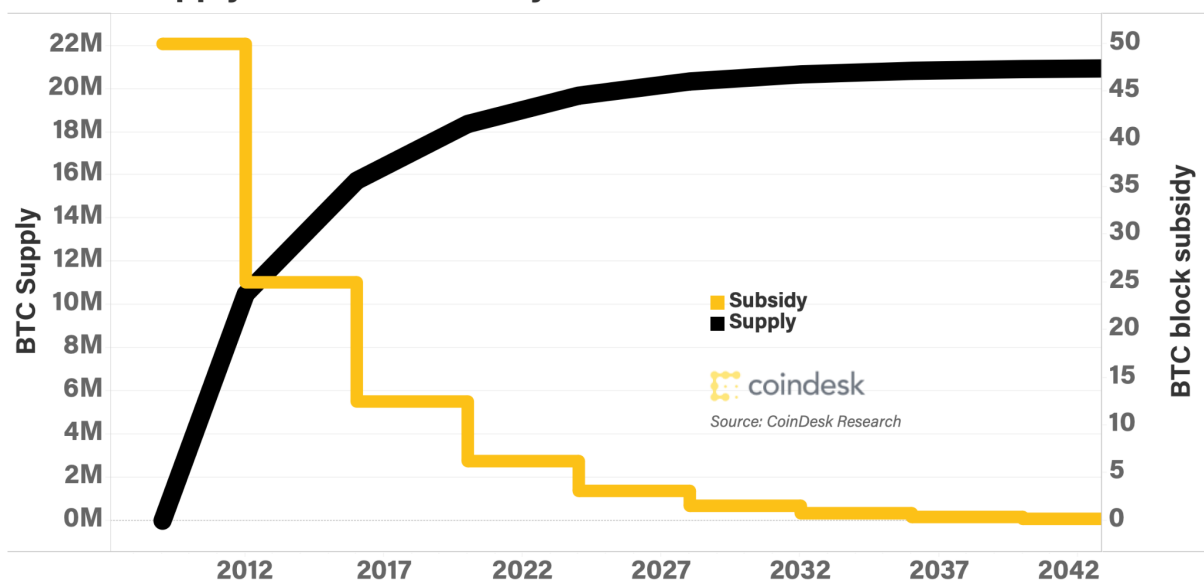


It is clear that holding BTC would be advantageous in this time frame, as less value would be lost over the year. Specifically with Bitcoin, assuming you had 9,688USD, enough for 1BTC, this would be the dollar value equivalent over the year from May 2020:



From this, it is clear that while the USD depreciates in value, BTC increases significantly in value. This is partially due to the fact that over time more and more USD is minted and printed by the government, while BTC follows a steady rule which creates a geometric limit of the total supply:

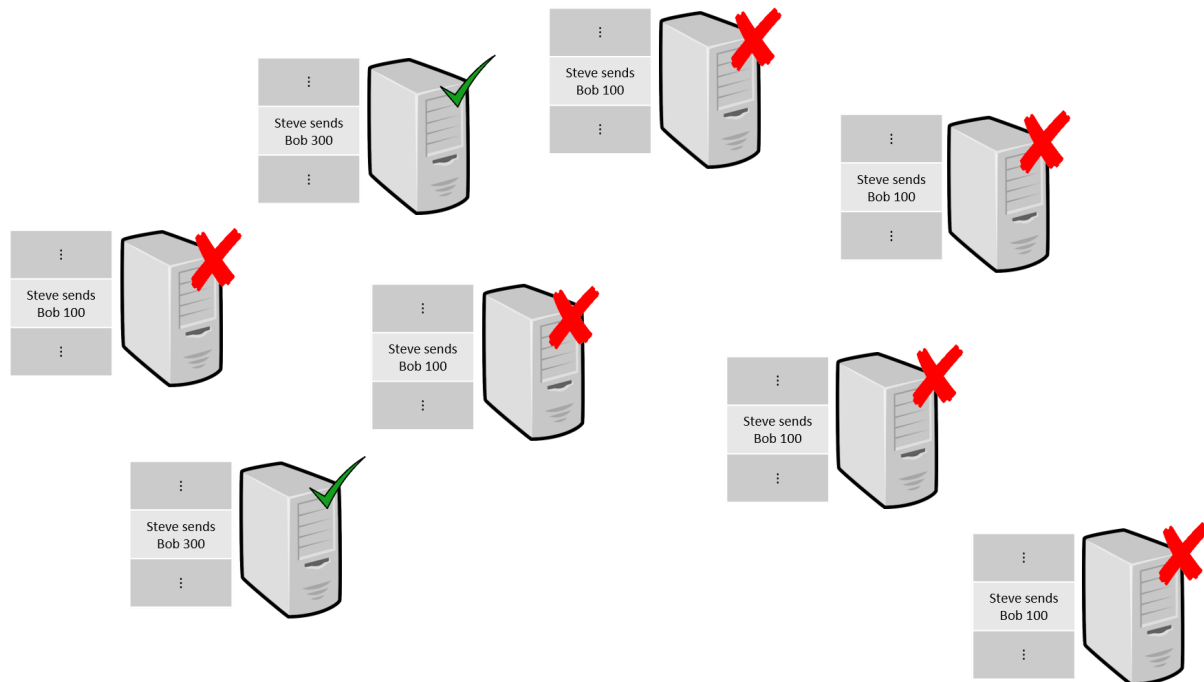
Bitcoin supply and block subsidy over time



The concept of not having a central governing authority seems very appealing. However, there is still one critical issue: How do you host an economy without any central trusted authority? How can Alice be sure that Bob will pay her? How can society be sure that Bob did not send himself fraudulent transactions from fake accounts crediting himself millions? How does Alice protect her account funds from being stolen? Who decides what is “real”?

Bob can claim that Steve sent him 300GC, and show the transaction on his copy of the ledger. If there is no central authority which everyone trusts to say “We never saw this”, how can society know if Steve sent Bob 300GC? Assume Steve also has a copy of a ledger, and his does not have the transaction. Now there is a dispute, Bob claims Steve paid him 300GC, while Steve claims this never happened. What is the “truth”? Did the transaction happen? Who is qualified to decide? There are no bank records or courts to settle the dispute. Therefore, digital currencies had to create consensus in a trustless environment. No single entity can be trusted, yet a common understanding of the “truth” must be had. Hence, blockchain came into existence. Blockchain provides consensus in a trustless environment. Each transaction is recorded on a block, and each block is verified and signed by someone on the network. The verifier of a block has to do some intense computation as a kind of “insurance” that they verified the block’s transactions. Using this system, if Bob fraudulently claims that Steve paid him 300GC, Bob would also have to sign any future block wherein he uses those 300GC, since, hopefully, the other verifiers on the network would have rejected Bob’s claimed transaction and would not credit his account. Due to this, Bob would have to beat all other verifiers to sign the next block, which would mean Bob theoretically needs over 50% of the computing power of the entire network - virtually unfeasible in a large enough network. The “truth” in the blockchain would be the chain of blocks with the highest consensus - the chain which appears in the most nodes’ history. Simply, the “truth” is what the majority agree upon.

See below a situation where Steve paid Bob 100GC, and Bob fraudulently claims it was 300. Then Bob tries to use those 300GC to pay Alice.



Bob's payment to Alice will only be valid if one of the nodes with a green checkmark verifies the block, as they believe that Steve did send Bob 300GC (they are probably Bob's servers). However, since most nodes have the transaction saved as being only worth 100 coins, they will reject Bob's payment to Alice. Therefore, the payment will only go through if the two servers which believe Bob produce the next blocks faster than all other nodes combined

The incentive of a node to verify transactions correctly and verify blocks is economic. A block verifier receives two rewards: a mining fee and transaction fees. The mining fee is actually how new coins are minted; as a reward for a node verifying transactions the system credits them with some fresh coins. The transaction fees are simply the cumulative fees tacked on each transaction on the block - each transaction has a fee which is awarded to whoever verifies the transaction - an incentive for block verifiers to include the transaction on their block. If a verifier does a sloppy job and does not verify the transactions on their block, other nodes will reject the block and not credit the verifier their reward. This would be economically unworthwhile since most of the computational power of verifying a block is not in verifying the transactions. Therefore, it would be a waste of time, power, and money to validate a block without verifying its transactions.

All this allows the network to reach consensus in a trustless environment, where all the rules are predefined, and no one-entity controls the economy.

Theoretical Background

BlockChain:

BlockChain technology is a concept by which data, specifically digital transactions and contracts, are stored on “blocks” and each block is dependent on the previous block. This dependency secures the “chain” of blocks from a nefarious block being inserted in retrospect or previous blocks being tampered with. BlockChain forces each block to be created with consideration for the previous block, and by induction, every block before it. Thus, each block is entangled with all the previous blocks, which creates an unbreakable chain.

Digital Signature Algorithm:

_____The Digital Signature Algorithm (DSA) is a public-key cryptography algorithm which can verify the authenticity of a sender as the actual source of a message, and tamper-check the data, using publicly available knowledge.

Hash Functions:

_____Hash functions transform data of an arbitrary size into a fixed length digest. This digest cannot be reversed in order to retrieve the original data, yet calculating the hash of the data is fairly simple. Hash functions are one-way functions which can quickly check whether two chunks of data match exactly.

Packets, Sockets and Ports:

Computers communicate with a network by sending Packets through Sockets and Ports.

A **packet** is the actual content of the data which is sent over the network. A packet is made up of the protocol metadata, usually IPs and ports, and actual data the applications want to send.

A **socket** is the actual application endpoint. An application can create a socket and through it communicate with other computers. The communications of a socket pass through a port.

Computers have many **ports**, $65,535 (2^{16} - 1)$ to be exact, which allow them to communicate with remote endpoints. A socket will use a port to send or receive packets from other computers.

Practical situation: An application opens a socket and binds it to a port, then creates a packet with information it wants to send and tells the socket to send it. The socket then sends the packet through the binded port to a remote endpoint - a remote socket.

Internet Protocols - TCP:

Communications over the internet are implemented using protocols. Each protocol has a set of rules which make it ideal for certain situations, and award it advantages as well as disadvantages. One of the most widely used protocols is Transmission Control Protocol (TCP), which is favored thanks to its reliability.

TCP requires a connection between two endpoints, which begins with a three-way-handshake where the initiator sends a “Synchronize” (SYN) packet to remote peer, the remote peer returns an “Acknowledgement” (ACK) and another “Synchronize” (SYN), and finally the initiator returns an “Acknowledgement” (ACK) to the peer. Due to TCP’s reliable connection and various packet verification methods, two endpoints can know if a packet was received or lost and whether the packet was received correctly. While all this validation creates a reliable protocol, the redundancy heavily weighs on the performance time. Therefore, TCP is used when packet sending frequency is low, and accuracy must be high.

Decentralized Networks - P2P:

A decentralized network is a set of connections between endpoints which communicate with each other, without the communication passing through a central server. In order to achieve a server-less communication net, each endpoint acts as both a server and a client. This will be referred to as a peer-to-peer connection (**P2P**), as network peers are connected to one another directly rather than through a server. Ideally, each peer on the network would host a server, and simultaneously connect to all other peers’ servers.

Cryptography

Cryptography is the mathematical field and study of obscuring, hiding, and securing communication. The classical definition of cryptography is encrypting communication data in order to prevent third-parties from intercepting messages. This technique was used throughout history, mostly by militaries, and was infamously used by the Nazi regime with the “Enigma” machine. Moreover, cryptography in the form of ciphers have been used for around 4,000 years, the earliest use being on a stone carving in Egypt. Ancient Greece initiated the dependency of militaries on cryptography.

However, since the advent of computer technology, classical methods of encryption have been rendered utterly useless. Arguably, the first ever computer, Alan Turing’s Enigma cryptanalysis machine, was actually specifically designed to crack a cryptographic cipher. What took a brilliant team of mathematicians several months to decrypt would now take anyone with an internet connection minutes. Therefore, much more advanced encryption methods have been developed and implemented, both in military and civilian use.

A commonly used type of cryptography is Cryptographic Hash Functions. These functions are used to convert data into a digest / hash which cannot be reversed. This is further explained below in [Hash Functions](#).

Another type of cryptography is Public-Key Cryptography, which has two major use cases: Encrypting communication between two parties, and verifying the authenticity of a message. The former, which is widely used on the web, recognizable by the ‘S’ in ‘HTTPS’, is a method of protecting communication between two parties without them needing to share a common “key”. Bob could use Alice’s public key to encrypt some data, and only Alice, who has the corresponding private key, would be able to understand the message. The first practical implementation of Public-Key Cryptography, is the RSA algorithm. The latter of the use cases is what transactions in Crypto-currencies use. An algorithm such as DSA can confirm the authenticity of a message using the signer’s public key, which matches the private key they used to sign the message. This form of public-key encryption does not encrypt a message in any way, rather proves that the message received is accurate to what the sender meant to send, and proves that said sender is really the originator of the message.

Hash Functions

A hash function is a function which maps data of an arbitrary length to a hash (data) of a fixed and predefined length.

The main properties of hash functions used in this project are the defined length and determinism. All hashing will be done using SHA256 (see below) which produces a fixed length output hash of 256 bits (32 bytes). This property of defined length allows the placement of a hash inside of a block in our blockchain (later explained in [BlockChain](#)) which is arguably the core of blockchain technology. The second property, determinism, creates certainty that data hashed today will produce the exact same hash tomorrow, and the steps by which this hash is produced are all mathematical, and therefore, deterministic. This means that a chunk of data hashed by Bob on Sunday in Tel Aviv, will produce the exact same hash for Alice on Tuesday in California. Since this is an absolute certainty, if Alice's hash is different from Bob's, either the data or the hash we tampered with (possibly by signal interference), or either Bob or Alice are lying.

A key feature and use of hashes is in data protection, specifically passwords. Hashes are one-way functions, meaning that a value for an input is easily computed, however, inverting a value to an input is practically impossible. Therefore, a password, say "qwerty", can be saved to a publicly accessible file as a hash, say a SHA256 hash "65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5", and it would be impossible to compute from this hash that "qwerty" is the password. This can be used in a function such as:

```
int SuperSecureLogin(char * password_input){

    char * correct_password_hash =
"65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5";

    char * input_as_hash = Hash(password_input);

    if(strcmp(correct_password_hash, input_as_hash) == 0) {
        // correct_password_hash == input_as_hash
        printf("Login successful!\n");
        printf("Welcome!\n");
    } else {
        // correct_password_hash != input_as_hash
        printf("Incorrect password!\n");
    }
}
```

A hacker could now access the source code for the login, say it was client-side javascript on a website, and even with the code, and the password being hardcoded, not be able to guess the password (Note: "qwerty" is a very weak and common password. Therefore,

the hash of qwerty can be found in hash directories and crackers such as JohnTheRipper or [hashtoolkit.com](https://github.com/hash-toolkit/hashtoolkit)).

Due to hash functions' one-way functionality, deriving data from a hash is not possible. This feature will be exploited in [BlockChain Mining](#).

SHA-256 - Secure Hash Algorithm - 256

SHA, the most common hashing algorithm family, was first published by the National Institute of Standards and Technology (NIST). The original, SHA-0, was developed by the National Security Agency for [DSA](#) use, though SHA-0 was withdrawn due to a significant flaw.

SHA-256 is part of the SHA-2 family, published in 2001, and is 256 bits in size. SHA-256 features 64 rounds of And, XOr, Rotation, Add, Or, Shift Right operations. As of June 2021, no collisions have been found for SHA-256, meaning it is secure and safe to use.

SHA-256, being a strong hash algorithm, produces completely different hashes for very similar inputs. For example:

Input	Hash
Hello World	a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
Hello, World	03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5
Hello world	64ec88ca00b268e5ba1a35678a1b5316d212f4f366b2477232534a8aeca37f3c

This will be used in [BlockChain Mining](#).

Code

The code, in C, to hash data of arbitrary length:

SHA_OPs.h + SHA_OPs.c

```
#pragma once

#include <stdint.h>
#include <string.h>

uint32_t SHA256_op_ch(uint32_t x, uint32_t y, uint32_t z);

uint32_t SHA256_op_maj(uint32_t x, uint32_t y, uint32_t z);

uint32_t SHA256_op_a(uint32_t x);

uint32_t SHA256_op_b(uint32_t x);
```

```
uint32_t SHA256_op_c(uint32_t x);

uint32_t SHA256_op_d(uint32_t x);

#include "SHA_OPs.h"

uint32_t SHA256_op_ch(uint32_t x, uint32_t y, uint32_t z)
{
    return ((x & y) ^ (~x & z));
}

uint32_t SHA256_op_maj(uint32_t x, uint32_t y, uint32_t z)
{
    return ((x & y) ^ (x & z) ^ (y & z));
}

uint32_t SHA256_op_a(uint32_t x)
{
    return (bit_rotate_right(x, 2) ^ bit_rotate_right(x, 13) ^ bit_rotate_right(x, 22));
}

uint32_t SHA256_op_b(uint32_t x)
{
    return (bit_rotate_right(x, 6) ^ bit_rotate_right(x, 11) ^ bit_rotate_right(x, 25));
}

uint32_t SHA256_op_c(uint32_t x)
{
    return (bit_rotate_right(x, 7) ^ bit_rotate_right(x, 18) ^ (x >> 3));
}

uint32_t SHA256_op_d(uint32_t x)
{
    return (bit_rotate_right(x, 17) ^ bit_rotate_right(x, 19) ^ (x >> 10));
}
```

SHA_Rotate.h + SHA_Rotate.c

```
#pragma once
```

```
#include <stdint.h>
#include <string.h>

uint32_t bit_rotate_left(uint32_t x, char rotations);
uint32_t bit_rotate_right(uint32_t x, char rotations);
uint64_t bit_rotate_right_64(uint64_t x, char rotations);
```

```
#include "SHA_Rotate.h"

uint32_t      bit_rotate_left(uint32_t x, char rotations)
{
    return ((x << rotations) | (x >> (32 - rotations)));
}

uint32_t      bit_rotate_right(uint32_t x, char rotations)
{
    return ((x >> rotations) | (x << (32 - rotations)));
}

uint64_t      bit_rotate_right_64(uint64_t x, char rotations)
{
    return ((x >> rotations) | (x << (64 - rotations)));
}
```

SHA256.h + SHA256.c

```
#pragma once

#ifndef __SHA256_H
#define __SHA256_H

#include <stdint.h>
#include <string.h>

#include "SHA_Rotate/SHA_Rotate.h"
#include "SHA_OPs/SHA_OPs.h"

typedef uint32_t      t_4_uint32[4];
typedef uint32_t      t_8_uint32[8];
typedef uint64_t      t_8_uint64[8];
```



```
typedef uint32_t    t_16_uint32[16];
typedef uint32_t    t_64_uint32[64];
typedef uint64_t    t_80_uint32[80];
typedef uint64_t    t_80_uint64[80];

#define DEC(x) (x-1)

// Chunks of 512 bits
#define SHA256_CHUNK_SIZE 64

#define SHA256_CHUNKS_SIZE(len) ((len + 1 + 8 + DEC(SHA256_CHUNK_SIZE)) &
-DEC(SHA256_CHUNK_SIZE))
#define SHA256_CHUNK_COUNT(len) (SHA256_CHUNKS_SIZE(len) /
SHA256_CHUNK_SIZE)

const uint32_t g_SHA256_k[64];

const uint32_t g_SHA256_default_buffers[8];

// Format a message into a standard 512bit block
size_t format_message(char * message, size_t message_len, unsigned char **
output_message_ptr);

uint32_t bit_swap_32(uint32_t x);

uint64_t bit_swap_64(uint64_t x);

char * SHA256_strncpy(char *dst, const char *src, size_t n);

void uint32_arr_assign_add(uint32_t *dst, const uint32_t *src, size_t len);
void uint32_arr_cpy(uint32_t *dst, const uint32_t *src, size_t len);
void uint64_arr_assign_add(uint64_t *dst, const uint64_t *src, size_t len);
void uint64_arr_cpy(uint64_t *dst, const uint64_t *src, size_t len);


char * uitoa_base(uintmax_t nb, intmax_t base, char letter);
char * uitoa_base_len(uintmax_t nb, intmax_t base, char letter, size_t len);

void SHA256_init_w_array(t_64_uint32 w_array, unsigned char *formatted_msg);

void SHA256_shuffle_buffers(t_8_uint32 buffers, t_64_uint32 w_array);
void SHA256_run_ops(t_8_uint32 buffers, unsigned char *formatted_msg, size_t
```

```
msg_len);

char * build_hash(uint32_t *buffers, size_t buffer_count);
// The SHA-256 hash algorithm
char * Hash_SHA256(char * message, size_t message_len);

#endif // !__SHA256_H

#include "SHA256.h"

const uint32_t g_SHA256_k[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
    0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174, 0xe49b69c1, 0xefbe4786,
    0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
    0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85, 0xa2bfe8a1, 0xa81a664b,
    0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
    0x5b9cca4f, 0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
};

const uint32_t g_SHA256_default_buffers[8] = {
    0x6a09e667,
    0xbb67ae85,
    0x3c6ef372,
    0xa54ff53a,
    0x510e527f,
    0x9b05688c,
    0x1f83d9ab,
    0x5be0cd19
};

// Format a message into a standard 512bit block
size_t format_message(char * message, size_t message_len, unsigned char **
output_message_ptr) {

    size_t output_message_len = message_len + (512 - (message_len % 512));
```

```
    size_t pos;

    *output_message_ptr = (unsigned char *)malloc(output_message_len);

    unsigned char * output_message = *output_message_ptr;

    // Copy the message into the beginning of the formatted message
    memcpy(output_message, message, message_len);

    // Set bit after message to 1, and the rest to 0.
    output_message[message_len] = 0b10000000;

    pos = message_len + 1;

    while (pos < output_message_len) {
        output_message[pos++] = 0;
    }

    output_message[output_message_len - 9] = (uint64_t)message_len;

    return output_message_len;
}

uint32_t bit_swap_32(uint32_t x)
{
    x = ((x << 8) & 0xFF00FF00) | ((x >> 8) & 0xFF00FF);
    return (x << 16) | (x >> 16);
}

uint64_t bit_swap_64(uint64_t x)
{
    x = ((x << 8) & 0xFF00FF00FF00FF00ULL)
        | ((x >> 8) & 0x00FF00FF00FF00FFULL);
    x = ((x << 16) & 0xFFFF0000FFFF0000ULL)
        | ((x >> 16) & 0x0000FFFF0000FFFFULL);
    return (x << 32) | (x >> 32);
}

char *SHA256_strncpy(char *dst, const char *src, size_t n)
{
    size_t i;
```

```
    i = 0;
    while (src[i] != '\0' && i < n)
    {
        dst[i] = src[i];
        i++;
    }
    while (i < n)
    {
        dst[i] = '\0';
        i++;
    }
    return (dst);
}

void uint32_arr_assign_add(uint32_t *dst, const uint32_t *src, size_t len)
{
    size_t i;

    i = 0;
    while (i < len)
    {
        dst[i] += src[i];
        i++;
    }
}

void uint32_arr_cpy(uint32_t *dst, const uint32_t *src, size_t len)
{
    size_t i;

    i = 0;
    while (i < len)
    {
        dst[i] = src[i];
        i++;
    }
}

void uint64_arr_assign_add(uint64_t *dst, const uint64_t *src, size_t len)
{
    size_t i;

    i = 0;
```

```
    while (i < len)
    {
        dst[i] += src[i];
        i++;
    }
}

void uint64_arr_cpy(uint64_t *dst, const uint64_t *src, size_t len)
{
    size_t i;

    i = 0;
    while (i < len)
    {
        dst[i] = src[i];
        i++;
    }
}

char *uitoa_base(uintmax_t nb, intmax_t base, char letter)
{
    uintmax_t    temp;
    int           power;
    char         *str;

    temp = nb;
    power = 1;
    while (temp /= base)
        power++;
    if (!(str = (char *)calloc(power + 1, sizeof(char)))) {
        return (NULL);
    }
    while (power--)
    {
        if (nb % base >= 10)
            str[power] = nb % base - 10 + letter;
        else
            str[power] = nb % base + '0';
        nb /= base;
    }
}
```

```
    }
    return (str);
}

char *uitoa_base_len(uintmax_t nb, intmax_t base, char letter, size_t len)
{
    int i;
    int diff;
    char *str;
    char *new_str;

    i = 0;
    str = uitoa_base(nb, base, letter);
    diff = len - strlen(str);
    if (diff > 0)
    {
        if (!(new_str = (char*)calloc(len + 1, sizeof(char))))
            return (NULL);
        while (i < diff)
            new_str[i++] = '0';
        SHA256_strncpy(new_str + i, str, len - diff);
        free(str);
        return (new_str);
    }
    return (str);
}

void SHA256_init_w_array(t_64_uint32 w_array, unsigned char *formatted_msg)
{
    int i;

    i = 0;
    while (i < 64)
    {
        if (i < 16) {
            w_array[i] = bit_swap_32(((uint32_t *)formatted_msg)[i]);
        }
        else {
            w_array[i] = SHA256_op_d(w_array[i - 2]) + w_array[i - 7] +
            SHA256_op_c(w_array[i - 15]) + w_array[i - 16];
        }
        i++;
    }
}
```

```
void SHA256_shuffle_buffers(t_8_uint32 buffers, t_64_uint32 w_array)
{
    int i = 0;
    uint32_t temp_a;
    uint32_t temp_b;

    while (i < 64)
    {
        temp_a = buffers[7] + SHA256_op_b(buffers[4]) + SHA256_op_ch(buffers[4],
buffers[5], buffers[6]) + g_SHA256_k[i] + w_array[i];
        temp_b = SHA256_op_a(buffers[0]) + SHA256_op_maj(buffers[0], buffers[1],
buffers[2]);
        buffers[7] = buffers[6];
        buffers[6] = buffers[5];
        buffers[5] = buffers[4];
        buffers[4] = buffers[3] + temp_a;
        buffers[3] = buffers[2];
        buffers[2] = buffers[1];
        buffers[1] = buffers[0];
        buffers[0] = temp_a + temp_b;
        i++;
    }
}

void SHA256_run_ops(t_8_uint32 buffers,
    unsigned char *formatted_msg, size_t msg_len)
{
    size_t chunk_i;
    t_64_uint32 w_array;
    t_8_uint32 internal_buffers;

    chunk_i = 0;
    while (chunk_i < SHA256_CHUNK_COUNT(msg_len))
    {
        SHA256_init_w_array(w_array, formatted_msg + chunk_i *
SHA256_CHUNK_SIZE);
        uint32_arr_cpy(internal_buffers, buffers, 8);
        SHA256_shuffle_buffers(internal_buffers, w_array);
        uint32_arr_assign_add(buffers, internal_buffers, 8);
        chunk_i++;
    }
}
```

```
char GLOBAL_HASH_BUFFER;
char * build_hash(uint32_t *buffers, size_t buffer_count)
{
    char * hash;
    char * hash_tmp;
    size_t buffer_i;
    uint32_t buffer;

    buffer_i = 0;
    if (!(hash = (char*)calloc((buffer_count * 8) + 1, sizeof(char)))) {
        return NULL;
    }

    while (buffer_i < buffer_count)
    {
        buffer = 0 ? bit_swap_32(buffers[buffer_i]) : buffers[buffer_i];
        if (!(hash_tmp = uitoa_base_len(buffer, 16, 'a', 8))) {
            return (NULL);
        }
        SHA256_strncpy(hash + (buffer_i * 8), hash_tmp, 8);
        free(hash_tmp);
        buffer_i++;
    }
    return hash;
}

// The SHA-256 hash algorithm
char * Hash_SHA256(char * message, size_t message_len) {
    unsigned char *formatted_msg;
    t_8_uint32 buffers;

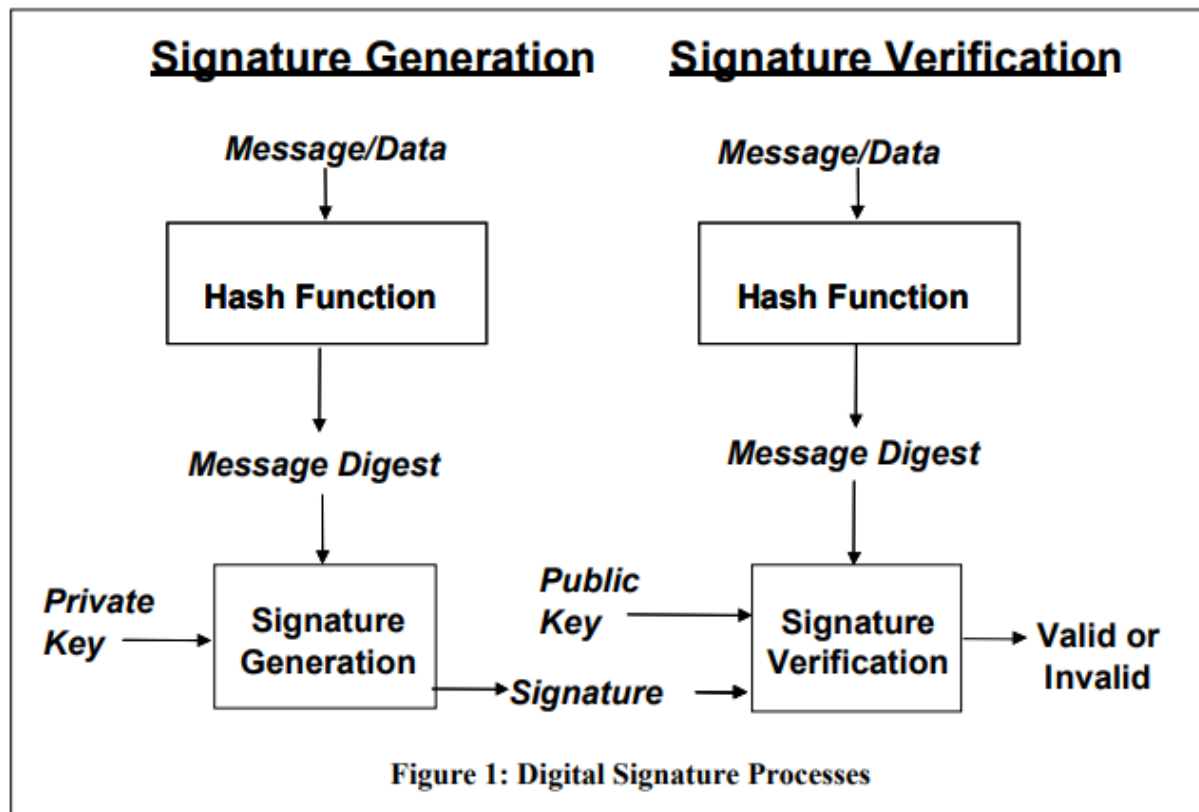
    size_t formatted_size = format_message(message, message_len, &formatted_msg);
    if (formatted_size != SHA256_CHUNK_COUNT(message_len) *
SHA256_CHUNK_SIZE) {
        //printf("Size mismatch!\n");
    }

    uint32_arr_cpy(buffers, g_SHA256_default_buffers, 8);
    SHA256_run_ops(buffers, formatted_msg, message_len);
    free(formatted_msg);
    return build_hash(buffers, 8);
}
```


DSA - Digital Signature Algorithm

The Digital Signature Algorithm, developed by the National Institute of Standards and Technology (NIST), is a method of verifying that a certain chunk of data was approved and signed by the relevant parties. DSA bases verification on the discrete logarithm problem, meaning that DSA can be trusted under the assumption that calculating the discrete logarithm of a very large number is obstinate.

Similarly to RSA, DSA uses a private and public key pair, the private key is used to sign the message, and the public key is used to verify the signature.



(Figure taken from FIPS 186-3, Digital Signature Standard)

Assuming the private key is kept secret, it can be confirmed with absolute certainty that a message with a valid signature was signed in its current form by the proper signatory. Should the message have been tampered with, say a malicious party altered the transaction value from 100 to 500, the hash function will return a different digest (as explained in [Hash Functions](#)) and the signature will no longer match. Reversing the private key from the public key is also intractable due to the discrete logarithm problem, and fixing the signature to match the altered data is just as difficult. Therefore, a DSA signature verifies that:

1. The message / data have not been tampered with.
2. The message was approved by the signatory (who's public key was used in verification).

Assume that Alice is a spy in a foreign country, and she wants to send Bob, another spy, secret information about the location of a warehouse he must visit. However, Bob is very

sceptical, and believes that the message is not really from Alice, rather that the local authorities are trying to set him up and send him into a trap. Alice can assure Bob with absolute certainty that she indeed sent the message by attaching some extra bits of information to this message. Alice will add the following:

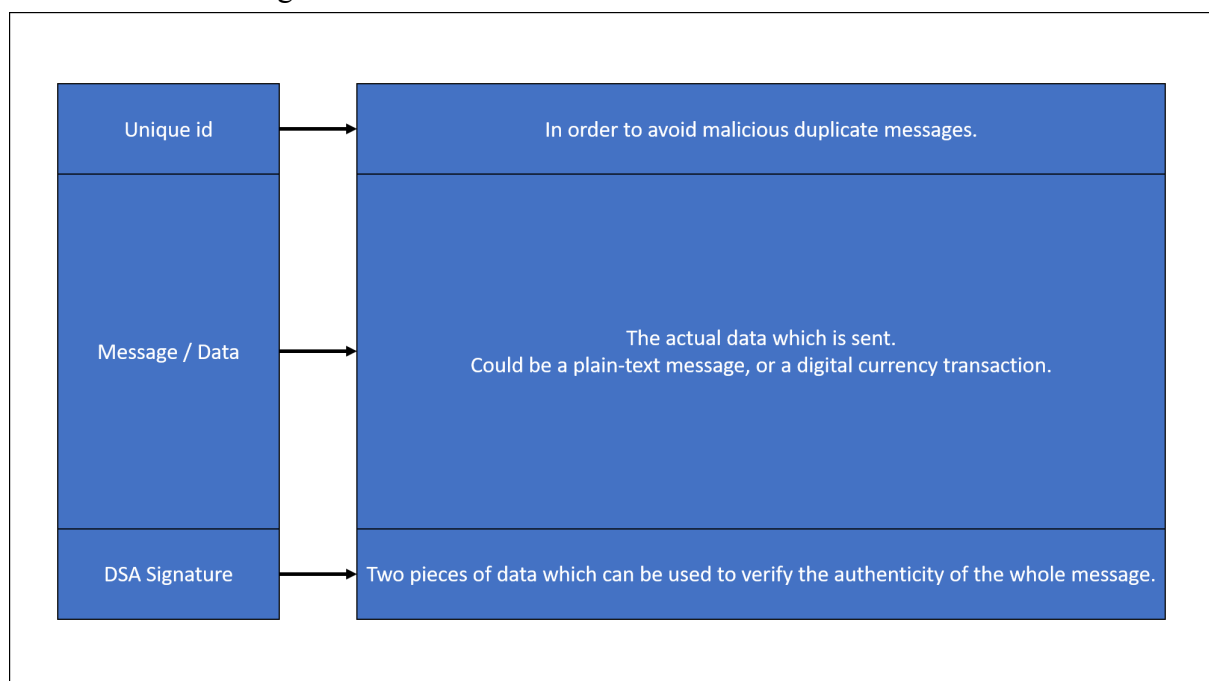
1. A unique identifier to the message which will never be reused. Like the current time.
2. A DSA signature.

Assuming that before being deployed, Bob was given Alice's public key, Bob can now use this key and the signature attached to the message to verify that Alice was indeed the source. The unique identifier is used in order to avoid a situation such as the following:

A few days after Alice sent Bob the message and he verified it and went to the warehouse, the local authorities, which managed to sniff the packet as Alice was sending it to Bob, set up a trap at the same warehouse, and without needing to do any hacking, resend the message as-is to Bob. Bob verifies the message, and sees absolutely no problem - the signature is valid. Bob goes to the warehouse again and is caught.

This could be avoided by having Alice attach a timestamp, the current time, to the original message. Now if Bob receives it again he can see that it is old news and ignore it, and for the local authorities to make it seem new, they would have to re-sign the whole message which would mean cracking the hash function - virtually impossible - or finding Alice's private key - which should also be impossible if Alice is responsible.

Alice's message structure would look like this:



Domain Parameters

Before signing and verifying messages, however, some public and general information should be agreed upon:

- p - A prime number which defines the Galois Field over which all operations are made, $GF(p)$.
- q - A prime factor of $p - 1$.

- g - The generator, of order q , of the cyclic group of $GF(p)^*$. That is, g is of order q , and is in the multiplicative group $GF(p)$.

Generating g :

Generate_ $g(p, q)$ {

1. $e = \frac{p-1}{q}$
2. $h \in \mathbb{N}. 1 < h < (p - 1)$
3. $g = h^e \bmod p$
4. if ($g = 1$), choose a different h , and go to step 2.
5. Return g .

}

Signature

The actual signature part of a message is quite short and simple. The algorithm produces two numbers, called 'r' and 's', or '(r, s)' as a whole. The pair (r, s) can then be used to verify the authenticity of the signature and the message. 'r' and 's' both satisfy $0 < r < q$, $0 < s < q$ respectively. Thus, a DSA signature is twice the bit length of q .

Generation

The algorithm to produce a signature corresponding to a message is as follows:

- I. z represents the hash of the message. $z =$ the min(N , outlen) leftmost bits of the hash of the message (where N is the bit length of q , and outlen is the bit length of the hash digest).
- II. Choose a random k such that $0 < k < q$, and then calculate k^{-1} such that $1 = (k \times k^{-1}) \bmod q$.
 - In order to keep the private key a secret, a new and different k should be used for each signature.
- III. $r = (g^k \bmod p) \bmod q$
- IV. $s = (k^{-1} (z + x \cdot r)) \bmod q$
- V. If ($r = 0$) \vee ($s = 0$) choose a different k and repeat from step II.

Let's demonstrate this signature generation using small values: Let:

- $p = 11$.
- $q = 5$ (Notice that 5 is a prime factor of $11-1$).
- Generate g :
 - $e = \frac{p-1}{q} = \frac{11-1}{5} = \frac{10}{5} = 2$.
 - Choose random $h = 3$.
 - $g = h^e \bmod p = 3^2 \bmod 11 = 9 \bmod 11 = 9$.

- $g = 9$.
- And let's give ourselves the private key $x = 4$.
- Assume our message was "Hello World":
 - The SHA-256 hash of "Hello World" is:
 "a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e" =
 "1010010110010001101001101101010000001011111010000100000
 010000000100101000000001000101110011001111001111011011110
 110001100100001101011000101100011001011011111000010111100
 11011010001100101011010101110110010011101111011001101011
 01100111110001010001101110" in binary.
 - The bit length of q (N) is 3, since $5 = 0b101$.
 - $\min(3, 256) = 3$.
 - $z = 0b101 = 5$.
- Choose random $k = 4$.
 - Calculating the inverse: $k^{-1} = 4$, since
 $(4 \times 4) \bmod 5 = 16 \bmod 5 = 1$.
- $r = (g^k \bmod p) \bmod q = (9^4 \bmod 11) \bmod 5 = (6561 \bmod 11) \bmod 5 = 5 \bmod 5 = 0$
 - This is an invalid r , and therefore we will choose a different k and repeat.
- Choose new random $k = 3$.
 - Calculating the inverse: $k^{-1} = 2$, since
 $(3 \times 2) \bmod 5 = 6 \bmod 5 = 1$.
- $r = (g^k \bmod p) \bmod q = (9^3 \bmod 11) \bmod 5 = (729 \bmod 11) \bmod 5 = 3 \bmod 5 = 3$
- $s = (k^{-1}(z + xr)) \bmod q = (2(5 + 4 \cdot 3)) \bmod 5 = (2(5 + 12)) \bmod 5 = (2 \cdot 17) \bmod 5 = 4$
- Signature: $(r, s) = (3, 4)$.

Verification

In order to verify a signature, the following steps are done. However, let's first calculate our public key 'y':

$$y = g^x \bmod p = 9^4 \bmod 11 = 5.$$

- I. z represents the hash of the message. $z =$ the $\min(N, \text{outlen})$ leftmost bits of the hash of the message (where N is the bit length of q , and outlen is the bit length of the hash digest).
- II. Check that $0 < r < q \wedge 0 < s < q$.
- III. $w = s^{-1} \bmod q$
- IV. $u_1 = (z \cdot w) \bmod q$
- V. $u_2 = (r \cdot w) \bmod q$
- VI. $v = ((g^{u_1} \cdot y^{u_2}) \bmod p) \bmod q$
- VII. If $(v = r)$, the signature is valid.

If either r or s are not in the range $(0, q)$, the signature is invalid. If at the end of the calculation, $v = r$, the signature is valid. Any other outcome means either the message has been tampered with, or the signature is fraudulent.

Let's continue with the demonstration now that we have the signature (3, 4) on the message "Hello World":

- $w = s^{-1} \bmod q = 4^{-1} \bmod 5 = 4.$
- $u_1 = (z \cdot w) \bmod q = (5 \cdot 4) \bmod 5 = 0.$
- $u_2 = (r \cdot w) \bmod q = (3 \cdot 4) \bmod 5 = 2.$
- $v = ((g^{u_1} \cdot y^{u_2}) \bmod p) \bmod q = ((9^0 \cdot 5^2) \bmod 11) \bmod 5 = (25 \bmod 11) \bmod 5 = 3.$
- $v = r \equiv 3 = 3.$

And voilà, $v = r$, meaning the signature is valid.

Bitwise Math

However, using small numbers like in the demonstration is extremely ill advised. A hacker could easily brute-force a signature by just guessing all the values as there are not many. Specifically, in our demonstration, the simplest attack with zero cryptographic skill would only have to guess only 16 numbers, something that would take any computer less than a second to calculate, and could even be done by hand in about a minute. To overcome this, using very large numbers is mandatory. Instead of defining p and q to be small 3-4 bit numbers, we choose extremely large values consisting of thousands of bits. The specific

value GreenCoin uses would require $(2^{160} - 1)^2$ guesses, meaning the chance of guessing correctly is around 2^{-320} which is less than the chance of randomly selecting a star in the universe and then flying blindly until you land on a random star, and then landing on the desired star four times in a row. Or more accurately, the chance is one in this number:

2,135,987,035,920,910,082,395,021,706,169,552,114,602,704,
,522,353,729,766,672,379,801,985,812,356,115,207,983,983,
650,221,850,625.

The catch is, in order to represent numbers this large, we need lots of bits - specifically, up to 2,048 bits for p . This is absurdly large considering we could not even fit the *bit length* of this number into some integer types!

Computers have many data types, the most common ones for storing numbers are:

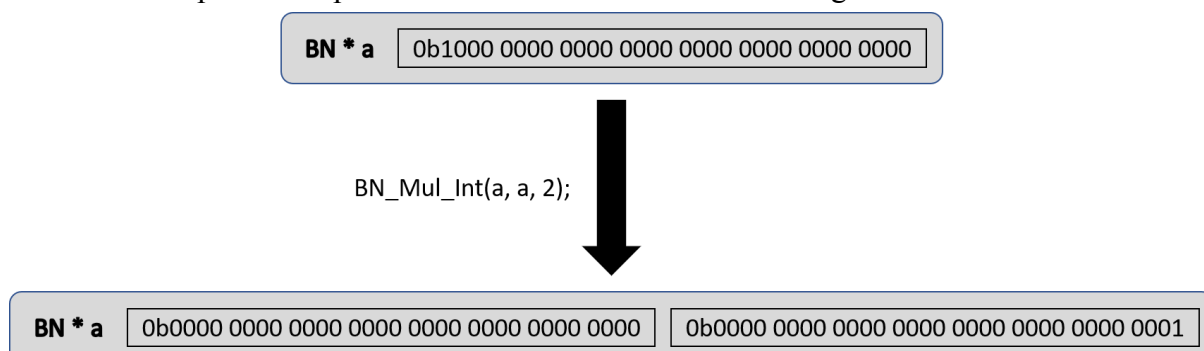
Byte Size	Type Name	Range	Name
1	Byte / Char	-128 -> 127	int8_t
		0 -> 255	uint8_t
2	Word	-32,768 -> 32,767	int16_t
		0 -> 65,535	uint16_t
4	Double Word	-2,147,483,648 -> 2,147,483,647	int32_t
		0 -> 4,294,967,295	uint32_t
		3.4E +/- 38 (7 digits)	float
8	Quad Word	-9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807	int64_t
		0 -> 18,446,744,073,709,551,615	uint64_t
		1.7E +/- 308 (15 digits)	double

Technically, some computers will also support SSE instructions which use 128 bit registers. While theoretically we could use these registers to perform 128 bit operations, this is not trivial and simply not worth the effort as it would not even work on plenty of computers.

Therefore, a whole new data type must be created - one which can hold arbitrarily large values. Notice, though, that we only actually need to deal with whole integers, specifically positive integers: $x \in \mathbb{N}^+$.

For this purpose we create a new library: BNMath (Big/Bit Number Math), which will allow us to manipulate arbitrarily large integers in a way similar to how we use normal integers. These BNs are implemented by using dynamically allocated arrays of uint32_t, which store the actual value of the number. Metadata such as sign and size is stored separately.

A multiplication operation on a BN would look something like this:



Since the number in 'a' multiplied by 2 was too large for the 32 bit storage, the BN increased the storage size to 2 uint32_t, stored in a semi-little endian format - the lower 32 bits being first. This method essentially uses 32 bit values as if they were bytes, and thus uses an array of them to form numbers.

Code

BNMath.h + BNMath.c

```
#pragma once
#ifndef __BNMath_H
#define __BNMath_H

#include "../General/error.h"
#include <string.h>
#include <math.h>
#include <stdint.h>
#include "stdio.h"

#define BN_INT_SIZE 4
#define NULL_DATA 0

typedef unsigned int uint_t;
typedef int int_t;

typedef enum {
    BN_LITTLE_ENDIAN,
    BN_BIG_ENDIAN
} BN_ENDIAN_FORMAT;

typedef struct {
    int_t sign;    // Positive or negative
    uint_t size;  // Bit size of the number - as multiples of uint_t
    uint_t * data; // The actual bit value of the number
} BN;

static const signed char HEX_REVERSE_SEQUENCE[256];

#define arraysize(a) (sizeof(a) / sizeof(a[0]))
#define BN_Is_Even(a)    !BN_Get_Bit_Value(a, 0)
#define BN_Is_Odd(a)     BN_Get_Bit_Value(a, 0)

int_t MAX(int_t a, int_t b);
int_t MIN(int_t a, int_t b);

void BN_Init(BN ** r); // Initialize a bit number;
```

```

void BN_Init_Stack(BN * r); // Initialize a bit number onto the stack. No memory
allocation
void BN_Free(BN * r); // Free a bit number;

error_t BN_Resize_Decrease(BN * r, uint_t size);
error_t BN_Resize(BN * n, uint_t size); // Resize the memory capacity of an existing bit
number;

uint_t BN_Get_Length(const BN * r);
uint_t BN_Get_Byte_Length(const BN * r);
uint_t BN_Get_Bit_Length(const BN * r);

error_t BN_Set_Bit_Value(BN * r, uint_t index, uint_t value);
uint_t BN_Get_Bit_Value(const BN * r, uint_t index);

int_t BN_Compare(const BN * a, const BN * b);
int_t BN_Compare_Int(const BN * a, int_t b);
int_t BN_Compare_Abs(const BN * a, const BN * b);

error_t BN_Copy(BN * r, const BN * a);
error_t BN_Set_Value(BN * r, long long int a);

error_t BN_Randomize(BN * r, uint_t length);

error_t BN_Import(BN * r, const uint8_t * data, uint_t length, BN_ENDIAN_FORMAT
format);
error_t BN_Import_Hex_String(BN * r, char * data, uint_t length,
BN_ENDIAN_FORMAT format);

error_t BN_Add(BN * r, const BN * a, const BN * b); // Adds a and b into r. r
= a + b
error_t BN_Add_Int(BN * r, const BN * a, int_t b); // Addition of Bit
number and a normal number
error_t BN_Add_Absolute(BN * r, const BN * a, const BN * b); // Addition assuming
both a and b are positive

error_t BN_Sub(BN * r, const BN * a, const BN * b); // Subtracts b from a
and returns into r. r = a - b
error_t BN_Sub_Int(BN * r, const BN * a, int_t b);
error_t BN_Sub_Absolute(BN * r, const BN * a, const BN * b);

error_t BN_Shift_Left(BN * r, uint_t l);
error_t BN_Shift_Right(BN * r, uint_t l);

```



```

error_t BN_Mul(BN * r, const BN * a, const BN * b);
error_t BN_Mul_Int(BN * r, const BN * a, int_t b);

error_t BN_Div(BN * q, BN * r, const BN * a, const BN * b);
error_t BN_Div_Int(BN * q, BN * r, const BN * a, int_t b);

error_t BN_Mod(BN * r, const BN * a, const BN * p);           // r = a % p
error_t BN_Add_Mod(BN * r, const BN * a, const BN * b, const BN * p); // r = (a + b) % p
error_t BN_Sub_Mod(BN * r, const BN * a, const BN * b, const BN * p); // r = (a - b) % p
error_t BN_Mul_Mod(BN * r, const BN * a, const BN * b, const BN * p); // r = (a * b) % p
error_t BN_Exp_Mod(BN * r, const BN * a, const BN * e, const BN * p); // r = pow(a,e) % p
error_t BN_Inv_Mod(BN * r, const BN * a, const BN * p);      // Returns (into r) the multiplicative inverse of a over the field p

error_t BN_Montgomery_Mul(BN * r, const BN * a, const BN * b, uint_t k, const BN * p, BN * t);
error_t BN_Montgomery_Red(BN * r, const BN * a, uint_t k, const BN * p, BN * t);

void BN_Mul_Core(uint_t *r, const uint_t *a, int_t m, const uint_t b);

void BN_Dump(FILE * stream, const char * prepend, const BN * a);

int BN_Is_Prime(BN * r);

#endif // !__BNMath_H

```

```

#include "BNMath.h"

```

```

static const signed char HEX_REVERSE_SEQUENCE[256] = {
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, -1, -1, -1, -1,
-1, 10, 11, 12, 13, 14, 15, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, 10, 11, 12, 13, 14, 15, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,

```

```
void BN_Free(BN * r) {
    if (r->data != NULL_DATA) {
        memset(r->data, 0, r->size * BN_INT_SIZE);
        free(r->data);
    }

    r->size = 0;
}
```

```
    r->data = NULL_DATA;

    free(r);
}

error_t BN_Resize_Decrease(BN * r, uint_t size) {
    uint_t * data;

    size = (uint_t)MAX(size, 1); // Override if size < 1

    data = calloc(size * BN_INT_SIZE);
    if (data == 0) { /* Could not allocate memory! */ return ERROR_FAILED; }

    if (r->size > 0) {
        memcpy(data, r->data, size * BN_INT_SIZE);
        /*printf("[0] 0x%.8X 0x%.8X\n", r->data[0], data[0]);
        printf("[1] 0x%.8X 0x%.8X\n", r->data[1], data[1]);
        printf("[2] 0x%.8X\n", r->data[2]);*/
        free(r->data);
    }

    r->size = size;
    r->data = data;

    return ERROR_NONE;
}

error_t BN_Resize(BN * r, uint_t size) {
    uint_t * data;

    size = (uint_t)MAX(size, 1); // Override if size < 1

    if (r->size == size) { return ERROR_NONE; }
    else if (r->size > size) {
        //return BN_Resize_Decrease(r, size);
        return ERROR_NONE;
    }

    data = calloc(size * BN_INT_SIZE, sizeof(char));
    if (data == 0) { /* Could not allocate memory! */ return ERROR_FAILED; }

    if (r->size > 0) {
```

```
    memcpy(data, r->data, r->size * BN_INT_SIZE);
    free(r->data);
}

r->size = size;
r->data = data;

return ERROR_NONE;
}
```

```
uint_t BN_Get_Length(const BN * r) {
    if (r->size == 0) { return 0; }

    uint_t i = 0;
    for (i = r->size - 1; i >= 0; i--) {
        if (r->data[i] != 0) {
            break;
        }
    }

    return i + 1;
}
```

```
uint_t BN_Get_Byte_Length(const BN * r) {
    uint_t n = 0;
    uint32_t m;

    if (r->size == 0) { return 0; }

    for (n = r->size - 1; n > 0; n--) {
        if (r->data[n] != 0) {
            break;
        }
    }

    m = r->data[n];
    n *= BN_INT_SIZE;

    for (; m != 0; m >>= 8) { n++; }

    return n;
}
```

```
uint_t BN_Get_Bit_Length(const BN * r) {
    uint_t n;
```

```
uint32_t m;

if (r->size == 0) { return 0; }

for (n = r->size - 1; n > 0; n--) {
    if (r->data[n] != 0) { break; }
}

m = r->data[n];
n *= BN_INT_SIZE * 8;

// Final bit count
for (; m != 0; m >>= 1) {
    n++;
}

return n;
}

error_t BN_Set_Bit_Value(BN * r, uint_t index, uint_t value) {
    uint_t quot;
    uint_t rema;

    quot = index / (BN_INT_SIZE * 8);
    rema = index % (BN_INT_SIZE * 8);

    error_t error;
    error = BN_Resize(r, quot + 1);
    if (error) { return error; }

    if (value) { // 1
        r->data[quot] |= (1 << rema); // Or
    }
    else { // 0
        r->data[quot] &= ~(1 << rema); // And
    }

    return ERROR_NONE;
}

uint_t BN_Get_Bit_Value(const BN * r, uint_t index) {
    uint_t quot = index / (BN_INT_SIZE * 8);
    uint_t rema = index % (BN_INT_SIZE * 8);
```

```
    if (quot >= r->size) { // Index out of range
        return 0;
    }

    return (r->data[quot] >> rema) & 0x01;
}

int_t BN_Compare(const BN * a, const BN * b) {
    uint_t m = BN_Get_Length(a);
    uint_t n = BN_Get_Length(b);

    if (!m && !n) {
        return 0;
    }
    else if (m > n) {
        return a->sign;
    } else if (m < n) {
        return -b->sign;
    }

    if (a->sign > 0 && b->sign < 0) {
        return 1;
    } else if (a->sign < 0 && b->sign > 0) {
        return -1;
    }

    while (n--)
    {
        if (a->data[n] > b->data[n]) {
            return a->sign;
        } else if (a->data[n] < b->data[n]) {
            return -a->sign;
        }
    }

    return 0;
}

int_t BN_Compare_Int(const BN * a, int_t b) {
    uint_t value = (b >= 0) ? b : -b;

    // Demo BN from b
    BN t;
```

```
t.sign = (b >= 0) ? 1 : -1;
t.size = 1;
t.data = &value;

return BN_Compare(a, &t);
}
int_t BN_Compare_Abs(const BN * a, const BN * b) {
    uint_t m = BN_Get_Length(a);
    uint_t n = BN_Get_Length(b);

    if (!m && !n) {
        return 0; // Numbers are both of size 0 -> empty
    } else if (m > n) {
        return 1;
    } else if (m < n) {
        return -1;
    }

    while (n--)
    {
        if (a->data[n] > b->data[n]) {
            return 1;
        } else if (a->data[n] < b->data[n]) {
            return -1;
        }
    }

    return 0;
}

error_t BN_Copy(BN * r, const BN * a) {
    if (r == a) { // are r and a the exact same
        return ERROR_NONE;
    }

    uint_t n = BN_Get_Length(a);

    //Adjust the size of the destination operand
    error_t error = BN_Resize(r, n);
    //Any error to report?
    if (error) {
        return error;
    }
}
```

```
    }

    memset(r->data, 0, r->size * BN_INT_SIZE);

    memcpy(r->data, a->data, n * BN_INT_SIZE);

    r->sign = a->sign;

    return ERROR_NONE;
}

error_t BN_Set_Value(BN * r, long long int a) {
    error_t error = BN_Resize(r, sizeof(a) / BN_INT_SIZE);
    if (error) {
        return error;
    }

    memset(r->data, 0, r->size * BN_INT_SIZE);

    r->data[0] = (a >= 0) ? a : -a;
    r->data[1] = (a >= 0) ? (a >> 8 * BN_INT_SIZE) : -(a >> 8 * BN_INT_SIZE);

    r->sign = (a >= 0) ? 1 : -1;

    return ERROR_NONE;
}

error_t BN_Randomize(BN * r, uint_t length) {
    uint_t n = (length + (BN_INT_SIZE * 8) - 1) / (BN_INT_SIZE * 8);
    uint_t m = length % (BN_INT_SIZE * 8);

    error_t error = BN_Resize(r, n);

    if (error) {
        return error;
    }

    memset(r->data, 0, r->size * BN_INT_SIZE);
    r->sign = 1;

    //Generate a random pattern
    for (int i = 0; i < length; i++) {
        error |= BN_Set_Bit_Value(r, i, rand()%2);
    }
}
```



```

    }
    if (error) {
        return error;
    }

    if (n > 0 && m > 0)
    {
        r->data[n - 1] &= (1 << m) - 1;
    }

    return ERROR_NONE;
}

error_t BN_Add(BN * r, const BN * a, const BN * b) {
    error_t error = 0;

    int_t sign = a->sign;

    if (a->sign == b->sign) {
        error = BN_Add_Absolute(r, a, b);
        r->sign = sign;
    } else {
        if (BN_Compare_Abs(a, b) >= 0) {
            //Perform subtraction
            error = BN_Sub_Absolute(r, a, b);
            r->sign = sign;
        } else {
            error = BN_Sub_Absolute(r, b, a);
            r->sign = -sign;
        }
    }

    return error;
}

error_t BN_Add_Int(BN * r, const BN * a, int_t b) {
    uint_t value;

    // Create demo BN from b
    BN t;
    value = (b >= 0) ? b : -b;
    t.sign = (b >= 0) ? 1 : -1;
    t.size = 1;

```

```
t.data = &value;

return BN_Add(r, a, &t);
}
error_t BN_Add_Absolute(BN * r, const BN * a, const BN * b) {
    error_t error = 0;
    uint_t i;
    uint_t n;
    uint_t c;
    uint_t d;

    // If b and r are the same, swap a and b
    if (r == b)
    {
        const BN * t = a;
        a = b;
        b = t;
    } else if (r != a)
    {
        BN_Copy(r, a); // Copy a to r
    }

    n = BN_Get_Length(b);
    BN_Resize(r, n);

    r->sign = 1;

    c = 0; // Carry bit

    // Add operands
    for (i = 0; i < n; i++)
    {
        // Add carry bit
        d = r->data[i] + c;
        // Update carry bit
        if (d != 0) c = 0;
        // Perform addition
        d += b->data[i];
        // Update carry bit
        if (d < b->data[i]) c = 1;
        // Save result
        r->data[i] = d;
    }
}
```

```
// Loop as long as the carry bit is set
for (i = n; c && i < r->size; i++)
{
    // Add carry bit
    r->data[i] += c;
    // Update carry bit
    if (r->data[i] != 0) c = 0;
}

// Check the final carry bit
if (c && n >= r->size)
{
    // Extend the size of the destination register
    BN_Resize(r, n + 1);
    // Add carry bit
    r->data[n] = 1;
}

return error;
}

error_t BN_Sub(BN * r, const BN * a, const BN * b) {
    error_t error = 0;
    int_t sign = a->sign;

    if (a->sign == b->sign)
    {
        if (BN_Compare_Abs(a, b) >= 0)
        {
            error = BN_Sub_Absolute(r, a, b);
            r->sign = sign;
        }
        else
        {
            error = BN_Sub_Absolute(r, b, a);
            r->sign = -sign;
        }
    }
    else {
        error = BN_Add_Absolute(r, a, b);
        r->sign = sign;
    }
}
```

```
        return error;
    }
error_t BN_Sub_Int(BN * r, const BN * a, int_t b) {
    uint_t value;

    // Create BN from b
    BN t;
    value = (b >= 0) ? b : -b;
    t.sign = (b >= 0) ? 1 : -1;
    t.size = 1;
    t.data = &value;

    return BN_Sub(r, a, &t);
}
error_t BN_Sub_Absolute(BN * r, const BN * a, const BN * b) {
    error_t error = 0;
    uint_t c;
    uint_t d;
    uint_t i;
    uint_t m;
    uint_t n;

    if (BN_Compare_Abs(a, b) < 0)
    {
        const BN * t = b;
        a = b;
        b = t;
    }

    m = BN_Get_Length(a);
    n = BN_Get_Length(b);

    BN_Resize(r, m);

    r->sign = 1;

    c = 0; // Carry bit

    for (i = 0; i < n; i++)
    {
        d = a->data[i];

        //Check the carry bit
```

```
    if (c)
    {
        //Update carry bit
        if (d != 0) c = 0;
        //Propagate carry bit
        d -= 1;
    }

    //Update carry bit
    if (d < b->data[i]) c = 1;
    //Perform subtraction
    r->data[i] = d - b->data[i];
}

//Loop as long as the carry bit is set
for (i = n; c && i < m; i++)
{
    //Update carry bit
    if (a->data[i] != 0) c = 0;
    //Propagate carry bit
    r->data[i] = a->data[i] - 1;
}

//R and A are not the same instance?
if (r != a)
{
    //Copy the remaining words
    for (; i < m; i++)
    {
        r->data[i] = a->data[i];
    }

    //Zero the upper part of R
    for (; i < r->size; i++)
    {
        r->data[i] = 0;
    }
}

return error;
}
```

```
error_t BN_Shift_Left(BN * r, uint_t l) {
    error_t error = 0;
    uint_t i;

    //Number of 32-bit words to shift
    uint_t n1 = l / (BN_INT_SIZE * 8);
    //Number of bits to shift
    uint_t n2 = l % (BN_INT_SIZE * 8);

    if (!r->size || !l) {
        return ERROR_NONE;
    }

    error = BN_Resize(r, r->size + (l + 31) / 32);
    if (error) {
        return error;
    }

    //First, shift words
    if (n1 > 0)
    {
        //Process the most significant words
        for (i = r->size - 1; i >= n1; i--)
        {
            r->data[i] = r->data[i - n1];
        }

        //Fill the rest with zeroes
        for (i = 0; i < n1; i++)
        {
            r->data[i] = 0;
        }
    }

    //Then shift bits
    if (n2 > 0)
    {
        //Process the most significant words
        for (i = r->size - 1; i >= 1; i--)
        {
            r->data[i] = (r->data[i] << n2) | (r->data[i - 1] >> (32 - n2));
        }
    }
}
```

```
//The least significant word requires a special handling
r->data[0] <=<= n2;
}

return ERROR_NONE;
}
error_t BN_Shift_Right(BN * r, uint_t l) {
    uint_t i;
    uint_t m;

    //Number of 32-bit words to shift
    uint_t n1 = l / (BN_INT_SIZE * 8);
    //Number of bits to shift
    uint_t n2 = l % (BN_INT_SIZE * 8);

    //Check parameters
    if (n1 >= r->size)
    {
        // If we are moving more bits than there are, then reset all the data.
        memset(r->data, 0, r->size * BN_INT_SIZE);
        return ERROR_NONE;
    }

    //First, shift words
    if (n1 > 0)
    {
        //Process the least significant words
        for (m = r->size - n1, i = 0; i < m; i++)
        {
            r->data[i] = r->data[i + n1];
        }

        //Fill the rest with zeroes
        for (i = m; i < r->size; i++)
        {
            r->data[i] = 0;
        }
    }

    //Then shift bits
    if (n2 > 0)
    {
        //Process the least significant words
        for (m = r->size - n1 - 1, i = 0; i < m; i++)
```

```

    {
        r->data[i] = (r->data[i] >> n2) | (r->data[i + 1] << (32 - n2));
    }

    //The most significant word requires a special handling
    r->data[m] >>= n2;
}

// Check if we need to resize
uint_t length_whole;
uint_t length_bits;
uint_t total_length;

total_length = BN_Get_Bit_Length(r);
length_whole = total_length / (BN_INT_SIZE * 8);
length_bits = total_length % (BN_INT_SIZE * 8);

uint_t actual_length;
actual_length = ceil_div(((length_whole * BN_INT_SIZE * 8) + length_bits) ,
(BN_INT_SIZE * 8));

if (actual_length != r->size && actual_length > 0) {
    // Resize
    BN_Resize(r, actual_length);
}

return ERROR_NONE;
}

error_t BN_Mul(BN * r, const BN * a, const BN * b) {
    error_t error = 0;
    int_t i;
    int_t m;
    int_t n;
    BN ta;
    BN tb;

    //Initialize multiple precision integers
    BN_Init_Stack(&ta);
    BN_Init_Stack(&tb);

    if (r == a)

```



```
{
//Copy A to TA
BN_Copy(&ta, a);
a = &ta;
}

if (r == b)
{
//Copy B to TB
BN_Copy(&tb, b);
b = &tb;
}

m = BN_Get_Length(a);
n = BN_Get_Length(b);

//Adjust the size of R
BN_Resize(r, m + n);

r->sign = (a->sign == b->sign) ? 1 : -1;

//Clear the contents of the destination integer
memset(r->data, 0, r->size * BN_INT_SIZE);

//Perform multiplication
if (m < n)
{
for (i = 0; i < m; i++)
{
    BN_Mul_Core(&r->data[i], b->data, n, a->data[i]);
}
}
else
{
for (i = 0; i < n; i++)
{
    BN_Mul_Core(&r->data[i], a->data, m, b->data[i]);
}
}

free(ta.data);
free(tb.data);
```

```
        return error;
    }
error_t BN_Mul_Int(BN * r, const BN * a, int_t b) {
    uint_t value;

    // Create BN from b
    BN t;
    value = (b >= 0) ? b : -b;
    t.sign = (b >= 0) ? 1 : -1;
    t.size = 1;
    t.data = &value;

    return BN_Mul(r, a, &t);
}

error_t BN_Div(BN * q, BN * r, const BN * a, const BN * b) {
    error_t error = 0;
    uint_t m;
    uint_t n;
    BN c;
    BN d;
    BN e;

    //Check whether the divisor is zero
    if (!BN_Compare_Int(b, 0)) {
        return ERROR_FAILED;
    }

    BN_Init_Stack(&c);
    BN_Init_Stack(&d);
    BN_Init_Stack(&e);

    BN_Copy(&c, a);
    BN_Copy(&d, b);
    BN_Set_Value(&e, 0);

    m = BN_Get_Bit_Length(&c);
    n = BN_Get_Bit_Length(&d);

    if (m > n) {
        BN_Shift_Left(&d, m - n);
    }
}
```

```
while (n++ <= m)
{
    BN_Shift_Left(&e, 1);

    int_t cmp = BN_Compare(&c, &d);

    if (cmp >= 0)
    {
        BN_Set_Bit_Value(&e, 0, 1);
        BN_Sub(&c, &c, &d);
    }

    BN_Shift_Right(&d, 1);
}

if (q != NULL) {
    BN_Copy(q, &e);
}

if (r != NULL) {
    BN_Copy(r, &c);
}

free(c.data);
free(d.data);
free(e.data);

return error;
}

error_t BN_Div_Int(BN * q, BN * r, const BN * a, int_t b) {
    uint_t value;

    // Create BN from b
    BN t;
    value = (b >= 0) ? b : -b;
    t.sign = (b >= 0) ? 1 : -1;
    t.size = 1;
    t.data = &value;

    return BN_Div(q, r, a, &t);
}
```

```
error_t BN_Mod(BN * r, const BN * a, const BN * p) {
    error_t error = 0;
    int_t sign;
    uint_t m;
    uint_t n;
    BN c;

    if (BN_Compare_Int(p, 0) <= 0) {
        return ERROR_FAILED;
    }

    BN_Init_Stack(&c);

    sign = a->sign;
    m = BN_Get_Bit_Length(a);
    n = BN_Get_Bit_Length(p);

    BN_Copy(r, a);

    if (m >= n)
    {
        BN_Copy(&c, p);
        BN_Shift_Left(&c, m - n);

        while (BN_Compare_Abs(r, p) >= 0)
        {
            if (BN_Compare_Abs(r, &c) >= 0)
            {
                BN_Sub_Absolute(r, r, &c);
            }

            BN_Shift_Right(&c, 1);
        }
    }

    if (sign < 0)
    {
        BN_Sub_Absolute(r, p, r);
    }

    free(c.data);
}
```

```
        return error;
    }

error_t BN_Add_Mod(BN * r, const BN * a, const BN * b, const BN * p) {
    error_t error = 0;

    BN_Add(r, a, b);
    BN_Mod(r, r, p);

    return error;
}

error_t BN_Sub_Mod(BN * r, const BN * a, const BN * b, const BN * p) {
    error_t error = 0;

    BN_Sub(r, a, b);
    BN_Mod(r, r, p);

    return error;
}

error_t BN_Mul_Mod(BN * r, const BN * a, const BN * b, const BN * p) {
    error_t error = 0;

    BN_Mul(r, a, b);
    BN_Mod(r, r, p);

    return error;
}

error_t BN_Exp_Mod(BN * r, const BN * a, const BN * e, const BN * p) {
    error_t error = 0;
    int_t i;
    int_t j;
    int_t n;
    uint_t d;
    uint_t k;
    uint_t u;
    BN b;
    BN c2;
    BN t;
    BN s[8];
```

```
//Initialize
BN_Init_Stack(&b);
BN_Init_Stack(&c2);
BN_Init_Stack(&t);

//Initialize precomputed values
for (i = 0; i < arraysize(s); i++)
{
    BN_Init_Stack(&s[i]);
}

//Very small exponents are often selected with low Hamming weight.
//The sliding window mechanism should be disabled in that case
d = (BN_Get_Bit_Length(e) <= 32) ? 1 : 4;

if (BN_Is_Even(p))
{
    //Let B = A^2
    BN_Mul_Mod(&b, a, a, p);
    BN_Copy(&s[0], a);

    //Precompute S[i] = A^(2 * i + 1)
    for (i = 1; i < (1 << (d - 1)); i++)
    {
        BN_Mul_Mod(&s[i], &s[i - 1], &b, p);
    }

    //Let R = 1
    BN_Set_Value(r, 1);

    //The exponent is processed in a left-to-right fashion
    i = BN_Get_Bit_Length(e) - 1;

    //Perform sliding window exponentiation
    while (i >= 0)
    {
        //The sliding window exponentiation algorithm decomposes E
        //into zero and nonzero windows
        if (!BN_Get_Bit_Value(e, i))
        {
            //Compute R = R^2
            BN_Mul_Mod(r, r, r, p);
            //Next bit to be processed
        }
    }
}
```

```

        i--;
    }
    else
    {
        //Find the longest window
        n = MAX(i - d + 1, 0);

        //The least significant bit of the window must be equal to 1
        while (!BN_Get_Bit_Value(e, n)) n++;

        //The algorithm processes more than one bit per iteration
        for (u = 0, j = i; j >= n; j--)
        {
            //Compute  $R = R^2$ 
            BN_Mul_Mod(r, r, r, p);
            //Compute the relevant index to be used in the precomputed table
            u = (u << 1) | BN_Get_Bit_Value(e, j);
        }

        //Perform a single multiplication per iteration
        BN_Mul_Mod(r, r, &s[u >> 1], p);
        //Next bit to be processed
        i = n - 1;
    }
}
}
else
{
    //Compute the smaller  $C = (2^{32})^k$  such as  $C > P$ 
    k = BN_Get_Length(p);

    //Compute  $C^2 \bmod P$ 
    BN_Set_Value(&c2, 1);
    BN_Shift_Left(&c2, 2 * k * (BN_INT_SIZE * 8));
    BN_Mod(&c2, &c2, p);

    //Let  $B = A * C \bmod P$ 
    if (BN_Compare(a, p) >= 0)
    {
        BN_Mod(&b, a, p);
        BN_Montgomery_Mul(&b, &b, &c2, k, p, &t);
    }
    else

```

```

{
    BN_Montgomery_Mul(&b, a, &c2, k, p, &t);
}

//Let  $R = B^2 * C^{-1} \bmod P$ 
BN_Montgomery_Mul(r, &b, &b, k, p, &t);
//Let  $S[0] = B$ 
BN_Copy(&s[0], &b);

//Precompute  $S[i] = B^{(2 * i + 1)} * C^{-1} \bmod P$ 
for (i = 1; i < (1 << (d - 1)); i++)
{
    BN_Montgomery_Mul(&s[i], &s[i - 1], r, k, p, &t);
}

//Let  $R = C \bmod P$ 
BN_Copy(r, &c2);
BN_Montgomery_Red(r, r, k, p, &t);

//The exponent is processed in a left-to-right fashion
i = BN_Get_Bit_Length(e) - 1;

//Perform sliding window exponentiation
while (i >= 0)
{
    //The sliding window exponentiation algorithm decomposes E
    //into zero and nonzero windows
    if (!BN_Get_Bit_Value(e, i))
    {
        //Compute  $R = R^2 * C^{-1} \bmod P$ 
        BN_Montgomery_Mul(r, r, r, k, p, &t);
        //Next bit to be processed
        i--;
    }
    else
    {
        //Find the longest window
        n = MAX(i - d + 1, 0);

        //The least significant bit of the window must be equal to 1
        while (!BN_Get_Bit_Value(e, n)) n++;

        //The algorithm processes more than one bit per iteration
    }
}

```



```

        for (u = 0, j = i; j >= n; j--)
        {
            //Compute  $R = R^2 * C^{-1} \bmod P$ 
            BN_Montgomery_Mul(r, r, r, k, p, &t);
            //Compute the relevant index to be used in the precomputed table
            u = (u << 1) | BN_Get_Bit_Value(e, j);
        }

        //Compute  $R = R * T[u/2] * C^{-1} \bmod P$ 
        BN_Montgomery_Mul(r, r, &s[u >> 1], k, p, &t);
        //Next bit to be processed
        i = n - 1;
    }

    //Compute  $R = R * C^{-1} \bmod P$ 
    BN_Montgomery_Red(r, r, k, p, &t);
}

free(b.data);
free(c2.data);
free(t.data);

for (i = 0; i < arraysize(s); i++)
{
    free(s[i].data);
}

return error;
}

error_t BN_Inv_Mod(BN * r, const BN * a, const BN * p) {
    error_t error = 0;
    BN b;
    BN c;
    BN q0;
    BN r0;
    BN t;
    BN u;
    BN v;

    //Initialize multiple precision integers
    BN_Init_Stack(&b);

```

```
BN_Init_Stack(&c);
BN_Init_Stack(&q0);
BN_Init_Stack(&r0);
BN_Init_Stack(&t);
BN_Init_Stack(&u);
BN_Init_Stack(&v);

BN_Copy(&b, p);
BN_Copy(&c, a);
BN_Set_Value(&u, 0);
BN_Set_Value(&v, 1);

while (BN_Compare_Int(&c, 0) > 0)
{
    BN_Div(&q0, &r0, &b, &c);

    BN_Copy(&b, &c);
    BN_Copy(&c, &r0);

    BN_Copy(&t, &v);
    BN_Mul(&q0, &q0, &v);
    BN_Sub(&v, &u, &q0);
    BN_Copy(&u, &t);
}

if (BN_Compare_Int(&b, 1))
{
    //MPI_CHECK(ERROR_FAILURE);
}

if (BN_Compare_Int(&u, 0) > 0)
{
    BN_Copy(r, &u);
}
else
{
    BN_Add(r, &u, p);
}

free(b.data);
free(c.data);
free(q0.data);
free(r0.data);
```

```

    free(t.data);
    free(u.data);
    free(v.data);

    return error;
}

error_t BN_Montgomery_Mul(BN * r, const BN * a, const BN * b, uint_t k, const BN * p,
BN * t)
{
    error_t error = 0;
    uint_t i;
    uint_t m;
    uint_t n;
    uint_t q;

    //Use Newton's method to compute the inverse of P[0] mod 2^32
    for (m = 2 - p->data[0], i = 0; i < 4; i++)
    {
        m = m * (2 - m * p->data[0]);
    }

    //Precompute -1/P[0] mod 2^32;
    m = ~m + 1;

    //We assume that B is always less than 2^k
    n = MIN(b->size, k);

    //Make sure T is large enough
    BN_Resize(t, 2 * k + 1);
    //Let T = 0
    BN_Set_Value(t, 0);

    //Perform Montgomery multiplication
    for (i = 0; i < k; i++)
    {
        //Check current index
        if (i < a->size)
        {
            //Compute q = ((T[i] + A[i] * B[0]) * m) mod 2^32
            q = (t->data[i] + a->data[i] * b->data[0]) * m;
            //Compute T = T + A[i] * B
            BN_Mul_Core(t->data + i, b->data, n, a->data[i]);
        }
    }
}

```

```

    }
    else
    {
        //Compute  $q = (T[i] * m) \bmod 2^{32}$ 
        q = t->data[i] * m;
    }

    //Compute  $T = T + q * P$ 
    BN_Mul_Core(t->data + i, p->data, k, q);
}

//Compute  $R = T / 2^{(32 * k)}$ 
BN_Shift_Right(t, k * (BN_INT_SIZE * 8));
BN_Copy(r, t);

//A final subtraction is required
if (BN_Compare(r, p) >= 0)
{
    BN_Sub(r, r, p);
}

return error;
}
error_t BN_Montgomery_Red(BN * r, const BN * a, uint_t k, const BN * p, BN * t)
{
    uint_t value;

    //Let  $B = 1$ 
    BN b;
    value = 1;
    b.sign = 1;
    b.size = 1;
    b.data = &value;

    //Compute  $R = A / 2^k \bmod P$ 
    return BN_Montgomery_Mul(r, a, &b, k, p, t);
}

void BN_Mul_Core(uint_t *r, const uint_t *a, int_t m, const uint_t b)
{
    int_t i;
    uint32_t c;

```

```
uint32_t u;
uint32_t v;
uint64_t p;

//Clear variables
c = 0;
u = 0;
v = 0;

//Perform multiplication
for (i = 0; i < m; i++)
{
    p = (uint64_t)a[i] * b;
    u = (uint32_t)p;
    v = (uint32_t)(p >> 32);

    u += c;
    if (u < c) v++;

    u += r[i];
    if (u < r[i]) v++;

    r[i] = u;
    c = v;
}

//Propagate carry
for (; c != 0; i++)
{
    r[i] += c;
    c = (r[i] < c);
}
}

void BN_Dump(FILE * stream, const char * prepend, const BN * a)
{
    uint_t i;

    //Process each word
    for (i = 0; i < a->size; i++)
    {
```

```
//Beginning of a new line?
if (i == 0 || ((a->size - i - 1) % 8) == 7)
    fprintf(stream, "%s", prepend);

//Display current data
fprintf(stream, "%08X ", a->data[a->size - 1 - i]);

//End of current line?
if (((a->size - i - 1) % 8) == 0 || i == (a->size - 1))
    fprintf(stream, "\r\n");
}
}

error_t BN_Import(BN * r, const uint8_t * data, uint_t length, BN_ENDIAN_FORMAT
format) {
    error_t error;
    uint_t i;

    //Check input format
    if (format == BN_LITTLE_ENDIAN)
    {
        //Skip trailing zeroes
        while (length > 0 && data[length - 1] == 0)
        {
            length--;
        }

        error = BN_Resize(r, (length + BN_INT_SIZE - 1) / BN_INT_SIZE);

        if (!error)
        {
            memset(r->data, 0, r->size * BN_INT_SIZE);
            r->sign = 1;

            //Import data
            for (i = 0; i < length; i++, data++)
            {
                r->data[i / BN_INT_SIZE] |= *data << ((i % BN_INT_SIZE) * 8);
            }
        }
    }
    else if (format == BN_BIG_ENDIAN)
    {

```

```

//Skip leading zeroes
while (length > 1 && *data == 0)
{
    data++;
    length--;
}

error = BN_Resize(r, (length + BN_INT_SIZE - 1) / BN_INT_SIZE);

if (!error)
{
    memset(r->data, 0, r->size * BN_INT_SIZE);
    r->sign = 1;

    //Start from the least significant byte
    data += length - 1;

    //Import data
    for (i = 0; i < length; i++, data--)
    {
        r->data[i / BN_INT_SIZE] |= *data << ((i % BN_INT_SIZE) * 8);
    }
}
else
{
    error = ERROR_FAILED;
}

return error;
}

error_t BN_Import_Hex_String(BN * r, char * data, uint_t length,
BN_ENDIAN_FORMAT format) {
    error_t error;
    uint_t i;

    //Skip leading zeroes
    /*while (length > 1 && *data == 0)
    {
        data++;
        length--;
    }*/

```

```
error = BN_Resize(r, (length + (2*BN_INT_SIZE) - 1) / (2*BN_INT_SIZE));

if (!error)
{
    memset(r->data, 0, r->size * BN_INT_SIZE);
    r->sign = 1;

    //Start from the least significant byte
    data += length - 1;

    //Import data
    for (i = 0; i < length/(BN_INT_SIZE*2); i++)
    {
        for (int a = 0; a < (2 * BN_INT_SIZE); a++) {
            r->data[i] |= HEX_REVERSE_SEQUENCE [*data] << (a*4);
            data--;
        }
    }
}

return error;
}

int BN_Is_Prime(BN * r) {

    uint_t order = BN_Get_Bit_Length(r);
    uint_t sqrt_order = ceil_div(order, 2);

    BN sqr;
    BN_Init_Stack(&sqr);

    for (int i = 0; i < sqrt_order; i++) {
        BN_Set_Bit_Value(&sqr, i, 1);
    }

    BN c;
    BN_Init_Stack(&c);
    BN_Set_Value(&c, 2);

    BN q;
    BN rem;
    BN_Init_Stack(&q);
    BN_Init_Stack(&rem);
```



```
while (BN_Compare(&c, &sqr) <= 0) {
    BN_Div(&q, &rem, r, &c);

    if (BN_Compare_Int(&rem, 0) == 0) {
        free(sqr.data);
        free(c.data);
        free(q.data);
        free(rem.data);
        return 0;
    }

    BN_Add_Int(&c, &c, 1);
}

free(sqr.data);
free(c.data);
free(q.data);
free(rem.data);

return 1;
}
```

Code

DSA.h + DSA.c

```
#pragma once

#ifndef __DSA_H
#define __DSA_H

#include "../BNMath/BNMath.h"
#include "../General/debug.h"
```

```
typedef BN* uint;

typedef BN DSA_Private_Key;
typedef BN DSA_Public_Key;

typedef struct {
    uint p; // Prime modulus
    uint q; // Order of the group
    uint G; // Group generator
} DSA_Domain_Parameters;

DSA_Domain_Parameters * params;

/*typedef struct {
    uint p;
    uint q;
    uint G;
    uint x; // Secret part of private key
} DSA_Private_Key;

typedef struct {
    uint p;
    uint q;
    uint G;
```

```
    uint y;
} DSA_Public_Key;*/

typedef struct {
    uint r;
    uint s;
} DSA_Signature;

typedef enum {
    SIGNATURE_VALID,
    SIGNATURE_INVALID
} SIGNATURE_VALID_STATE;

void DSA_Init_Public_Key(DSA_Public_Key * key);
void DSA_Free_Public_Key(DSA_Public_Key * key);
//void DSA_Load_Public_Key(DSA_Public_Key * key, uint p, uint q, uint G);
//void DSA_Load_Public_Key_From_Params(DSA_Public_Key ** key,
DSA_Domain_Parameters * params);
//void DSA_Load_Public_Key_Domain_Params(DSA_Public_Key * key,
DSA_Domain_Parameters * params);

void DSA_Init_Private_Key(DSA_Private_Key * key);
void DSA_Free_Private_Key(DSA_Private_Key * key);
//void DSA_Load_Private_Key(DSA_Private_Key * key, uint p, uint q, uint G);
```

```
error_t DSA_Create_Keys(DSA_Private_Key * priv_key, DSA_Public_Key * pub_key);
```

```
void DSA_Init_Signature(DSA_Signature ** signature);
```

```
void DSA_Free_Signature(DSA_Signature * signature);
```

```
void DSA_Generate_P(uint p, uint q, uint_t L);
```

```
void DSA_Generate_G(uint G, uint p, uint q, uint h);
```

```
error_t DSA_Generate_Signature(const DSA_Private_Key * key, const uint8_t *  
message_digest, size_t message_len, DSA_Signature * signature);
```

```
SIGNATURE_VALID_STATE DSA_Verify_Signature(const DSA_Public_Key * key,  
const uint8_t * message_digest, size_t message_len, DSA_Signature * signature);
```

```
#endif // !__DSA_H
```

```
#include "DSA.h"
```

```
void DSA_Init_Public_Key(DSA_Public_Key ** key) {
```

```
    BN_Init(key);
```

```
    /*key = (DSA_Public_Key*)malloc(sizeof(DSA_Public_Key));
```

```
    DSA_Public_Key * key_addr = *key;
```

```
    /*BN_Init(&(key_addr->p));
```

```
    BN_Init(&(key_addr->q));
```

```
        BN_Init(&(key_addr->G));

        BN_Init(&(key_addr->y));*/
    }

/*void DSA_Load_Public_Key_From_Params(DSA_Public_Key ** key,
DSA_Domain_Parameters * params) {

    *key = (DSA_Public_Key*)malloc(sizeof(DSA_Public_Key));

    DSA_Public_Key * key_addr = *key;

    key_addr->p = params->p;

    key_addr->q = params->q;

    key_addr->G = params->G;

    BN_Init(&(key_addr->y));

}

void DSA_Load_Public_Key_Domain_Params(DSA_Public_Key * key,
DSA_Domain_Parameters * params) {

    if (key->p != NULL) { memcpy(key->p, params->p, params->p->size + 2); }

    else {

        key->p = params->p;

    }

    if (key->q != NULL) { memcpy(key->q, params->q, params->q->size + 2); }

    else {

        key->q = params->q;

    }

    if (key->G != NULL) { memcpy(key->G, params->G, params->G->size + 2); }

    else {

        key->G = params->G;
```

```
    }  
}*/  
  
void DSA_Free_Public_Key(DSA_Public_Key * key) {  
    /*BN_Free(&(key->p));  
    BN_Free(&(key->q));  
    BN_Free(&(key->G));  
    BN_Free(&(key->y));*/  
    BN_Free(key);  
}  
  
/*void DSA_Load_Public_Key(DSA_Public_Key * key, uint p, uint q, uint G) {  
    key->p = p;  
    key->q = q;  
    key->G = G;  
}*/  
  
void DSA_Init_Private_Key(DSA_Private_Key ** key) {  
    BN_Init(key);  
    /*  
    *key = (DSA_Private_Key*)malloc(sizeof(DSA_Private_Key));  
    DSA_Private_Key * key_addr = *key;  
    BN_Init(&(key_addr->p));  
    BN_Init(&(key_addr->q));  
    BN_Init(&(key_addr->G));  
    BN_Init(&(key_addr->x));*/
```

```
}

void DSA_Free_Private_Key(DSA_Private_Key * key) {

    /*BN_Free(&(key->p));

    BN_Free(&(key->q));

    BN_Free(&(key->G));

    BN_Free(&(key->x));*/

    BN_Free(key);

}

/*void DSA_Load_Private_Key(DSA_Private_Key * key, uint p, uint q, uint G) {

    key->p = p;

    key->q = q;

    key->G = G;

}*/

error_t DSA_Create_Keys(DSA_Private_Key * priv_key, DSA_Public_Key * pub_key) {

    BN * p;

    BN * q;

    BN * G;

    /*if (BN_Compare(priv_key->p, pub_key->p) != 0) { return ERROR_FAILED; }

    if (BN_Compare(priv_key->q, pub_key->q) != 0) { return ERROR_FAILED; }

    if (BN_Compare(priv_key->G, pub_key->G) != 0) { return ERROR_FAILED; }*/

    p = params->p;

    q = params->q;
```

```
G = params->G;

uint_t N;
uint_t L;

N = BN_Get_Bit_Length(q);
L = BN_Get_Bit_Length(p);

BN c1;
BN c;

BN_Init_Stack(&c1);
BN_Init_Stack(&c);

BN_Randomize(&c1, N + 64);

BN qt;

BN_Init_Stack(&qt);
BN_Sub_Int(&qt, q, 1);

BN_Mod(&c, &c1, &qt);

BN_Add_Int(priv_key, &c, 1);

BN_Exp_Mod(pub_key, G, priv_key, p);
```



```
    free(c1.data);

    free(c.data);

    return ERROR_NONE;
}

void DSA_Init_Signature(DSA_Signature ** signature) {
    *signature = (DSA_Signature*)malloc(sizeof(signature));

    DSA_Signature * sig_addr = *signature;

    BN_Init(&(sig_addr->r));

    BN_Init(&(sig_addr->s));
}

void DSA_Free_Signature(DSA_Signature * signature) {
    BN_Free((signature->r));

    BN_Free((signature->s));

    //free(signature);
}

void DSA_Generate_P(uint p, uint q, uint_t L) {
    BN_Add_Int(p, q, 1);

    int_t m = 2;

    while (!BN_Is_Prime(p)) {
```

```
        BN_Mul_Int(p, q, m);

        BN_Add_Int(p, p, 1);

        m++;
    }
}

void DSA_Generate_G(uint G, uint p, uint q, uint h) {

    BN * e;

    BN_Init(&e);

    BN * r;

    BN_Init(&r);

    BN * t; BN_Init(&t);

    BN_Sub_Int(t, p, 1);

    BN_Div(e, r, t, q);

    do {

        BN_Exp_Mod(G, h, e, p);

        BN_Add_Int(h, h, 1);

    } while (!BN_Compare_Int(G, 1));

    BN_Free(e);

    BN_Free(t);
}
```

```
BN_Free(r);
}

error_t DSA_Generate_Signature(const DSA_Private_Key * key, const uint8_t *
message_digest, size_t message_len, DSA_Signature * signature) {

    error_t error = 0;

    uint_t n;

    BN k;

    BN z;

    //Check parameters

    if (key == NULL || message_digest == NULL || signature == NULL) {

        return ERROR_FAILED;

    }

    //Debug message

    TRACE_DEBUG("DSA signature generation...\r\n");

    TRACE_DEBUG(" p:\r\n");

    TRACE_DEBUG_MPI("    ", (key->p));

    TRACE_DEBUG(" q:\r\n");

    TRACE_DEBUG_MPI("    ", (key->q));

    TRACE_DEBUG(" g:\r\n");

    TRACE_DEBUG_MPI("    ", (key->G));

    TRACE_DEBUG(" x:\r\n");

    TRACE_DEBUG_MPI("    ", (key->x));
```

```
TRACE_DEBUG(" digest:\r\n");

TRACE_DEBUG_ARRAY("", message_digest, message_len);


//Initialize multiple precision integers

BN_Init_Stack(&k);

BN_Init_Stack(&z);


//Let N be the bit length of q

n = BN_Get_Bit_Length((params->q));


//Compute N = MIN(N, outlen)

n = min(n, message_len * 4);


//Convert the digest to a multiple precision integer

BN_Import_Hex_String(&z, message_digest, (n + 3) / 4, BN_BIG_ENDIAN);


//Keep the leftmost N bits of the hash value

if ((n % 8) != 0)
{
    BN_Shift_Right(&z, 8 - (n % 8));
}


//Debug message

TRACE_DEBUG(" z:\r\n");
```

```
TRACE_DEBUG_MPI("    ", &z);

do {

    //Generated a pseudorandom number
    BN_Randomize(&k, n);

    //Make sure that  $0 < k < q$ 
    if (BN_Compare(&k, (params->q)) >= 0)
        BN_Shift_Right(&k, 1);

    //Compute  $r = (g^k \bmod p) \bmod q$ 
    BN_Exp_Mod((signature->r), (params->G), &k, (params->p));
    BN_Mod((signature->r), (signature->r), (params->q));

    //Compute  $k^{-1} \bmod q$ 
    BN_Inv_Mod(&k, &k, params->q);

    //Compute  $s = k^{-1} * (z + x * r) \bmod q$ 
    BN_Mul((signature->s), (key), (signature->r));
    BN_Add((signature->s), (signature->s), &z);
    //BN_Mod((signature->s), (signature->s), (key->q));

    BN_Mul_Mod((signature->s), (signature->s), &k, (params->q));
```

```
    } while ((BN_Compare_Int(signature->r, 0) == 0 || BN_Compare_Int(signature->s,
0) == 0));

    //Dump DSA signature

    TRACE_DEBUG(" r:\r\n");

    TRACE_DEBUG_MPI("    ", (signature->r));

    TRACE_DEBUG(" s:\r\n");

    TRACE_DEBUG_MPI("    ", (signature->s));

    free(k.data);

    free(z.data);

    //Clean up side effects if necessary

    if (error)

    {

        //Release (R, S) integer pair

        BN_Free((signature->r));

        BN_Free((signature->s));

    }

    return error;

}

error_t DSA_Verify_Signature(const DSA_Public_Key * key, const uint8_t *
message_digest, size_t message_len, DSA_Signature * signature) {
```

```
BN * r = signature->r;

BN * s = signature->s;


BN * p = params->p;

BN * q = params->q;

BN * G = params->G;

BN * y = key;


if (!(BN_Compare_Int(r, 0) > 0 && BN_Compare(r, q) < 0)) { return
SIGNATURE_INVALID; }

if (!(BN_Compare_Int(s, 0) > 0 && BN_Compare(s, q) < 0)) { return
SIGNATURE_INVALID; }


BN w;

BN z;

BN u1;

BN u2;

BN v;

BN_Init_Stack(&w);

BN_Init_Stack(&z);

BN_Init_Stack(&u1);

BN_Init_Stack(&u2);

BN_Init_Stack(&v);


BN_Inv_Mod(&w, s, q);
```

```
uint_t n = BN_Get_Bit_Length(q);

n = min(n, message_len * 4);

BN_Import_Hex_String(&z, message_digest, (n + 3) / 4, BN_BIG_ENDIAN);

if ((n % 8) != 0)
{
    BN_Shift_Right(&z, 8 - (n % 8));
}

BN_Mul_Mod(&u1, &z, &w, q);
BN_Mul_Mod(&u2, r, &w, q);

BN tmp1;
BN tmp2;

BN_Init_Stack(&tmp1);
BN_Init_Stack(&tmp2);

BN_Exp_Mod(&tmp1, G, &u1, p);
BN_Exp_Mod(&tmp2, y, &u2, p);

BN_Mul_Mod(&v, &tmp1, &tmp2, p);

BN_Mod(&v, &v, q);
```



```
    free(tmp1.data);

    free(tmp2.data);

    TRACE_DEBUG(" v:\r\n");
    TRACE_DEBUG_MPI("    ", &v);

    int valid = (BN_Compare(&v, r) == 0);

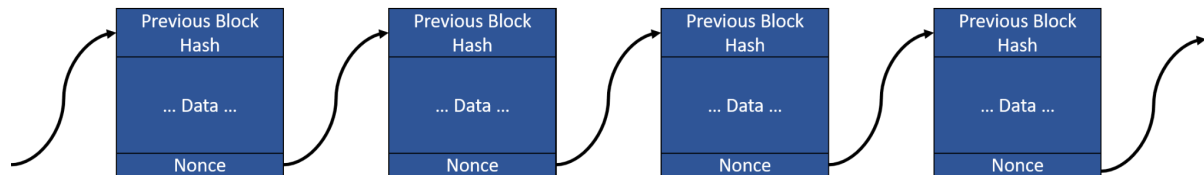
    free(w.data);
    free(z.data);
    free(u1.data);
    free(u2.data);
    free(v.data);

    if (valid) {
        return SIGNATURE_VALID;
    }

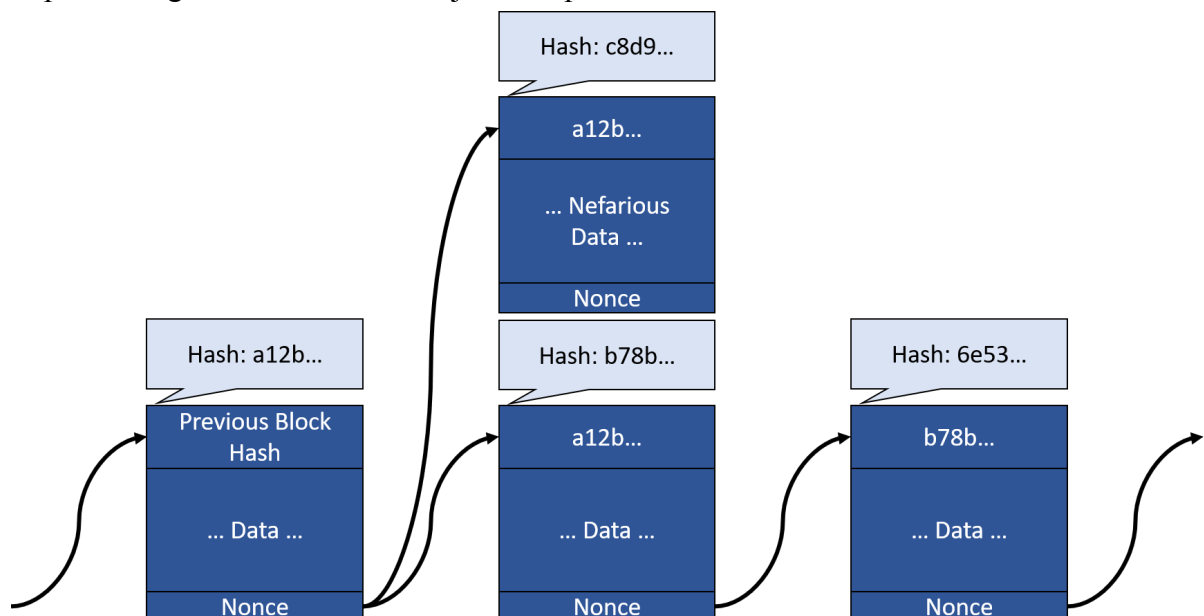
    return SIGNATURE_INVALID;
}
```

BlockChain

Blockchain technology is the backbone of all cryptocurrencies. A blockchain is a growing set of blocks linked together using hash algorithms. The purpose of this linkage is to avoid the situation where a malicious party injects a fraudulent block into the history of blocks.



Now assume a hacker did try and fake a block. Adding the block at the head of the chain would be practically impossible as all connected nodes will reject the block and not add it. However, if the hacker can insert the block into the middle of the chain, some nodes might skip checking the transactions and just accept the block if the hash chain matches.



Here it is clear that while a hacker can use the previous block hash, it would be virtually impossible to match the hash of the nefarious block's hash to the real block's hash. Therefore, this new block will create an entire new chain, which will necessarily be shorter, or simply have less computational value, and thus be less valuable than the real blockchain. Hence, the community will easily be able to differentiate between the real chain and the fake one, and uphold the authenticity of the blockchain.

BlockChain is useful since it essentially splits a very complicated problem into smaller easier problems; instead of the problem of verifying that every transaction ever made is valid, just verify this set of transactions, and then have a notary sign on the set as a whole and call it a block. Now instead of verifying each transaction, an entire set can be verified at once as a block.

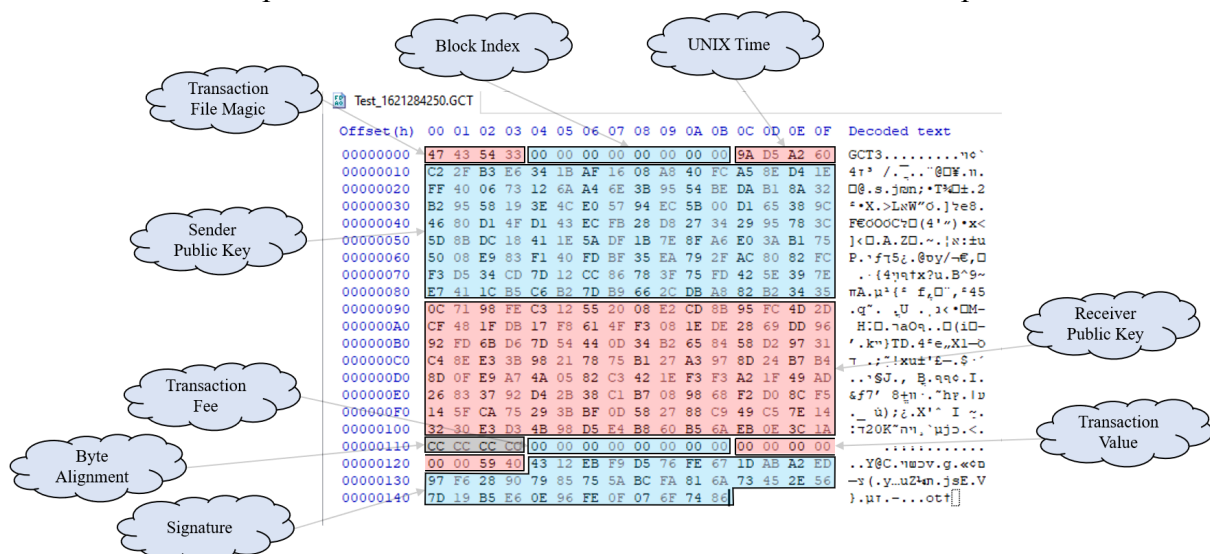
Transactions

Transactions are the actual register of the exchange of value on the blockchain. Each transaction will hold the information about the sender, receiver, the value of the transaction, some metadata, and a signature - similar to a common receipt. A group of transactions, in the case of GreenCoin - 64, will be written onto a block at which point a notary will verify the whole set and sign them off. Each transaction is completely unique as the transaction must include the time it was signed. This facilitates easy counterfeiting handling, as the second occurrence of the exact same transaction will necessarily be fraudulent. To make this verification even faster, the transaction metadata also includes the block it is on, and therefore a transaction can only be duplicated on the same block. Thus, a notary will verify that there are no duplicate transactions on a block, and if there are, will take only one.

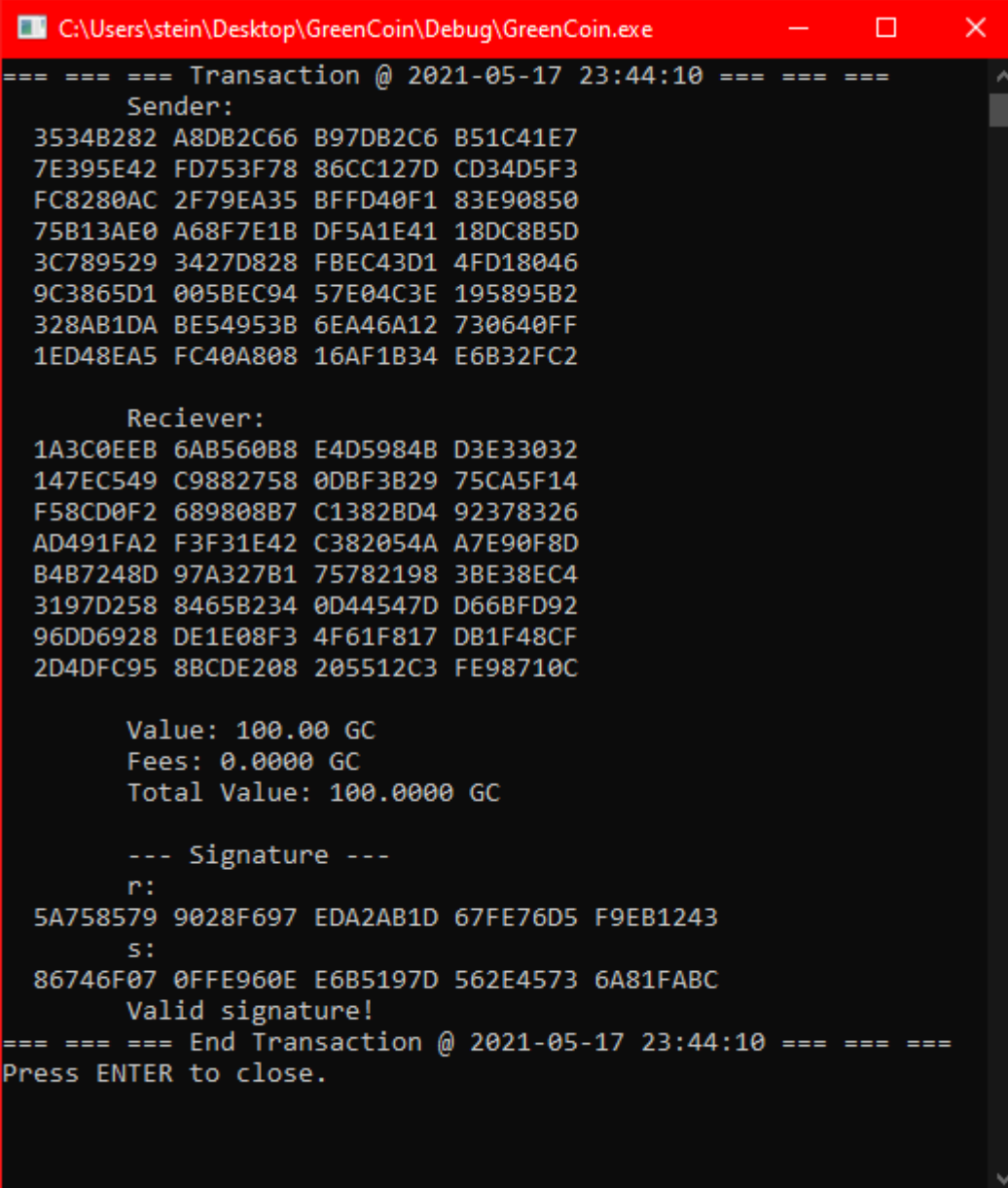
A transaction, including its metadata, has the following structure:

Byte Size	Data Name	Description
8	Block Index	The index of the block the transaction is destined for.
4	Time (UNIX)	The time the transaction was signed.
128	Sender	Public key of the sender.
128	Receiver	Public key of the receiver.
8	Fee	The fee to be collected by the notary.
8	Value	The value to be passed to the receiver. Note: Receiver += Value; Sender -= (Value + Fee);
40	Signature	The DSA signature (r, s).

An actual exported transaction will look like this in hexadecimal representation:



And the actual GreenCoin program will show a preview of the transaction in the form of a receipt like this:



```

C:\Users\stein\Desktop\GreenCoin\Debug\GreenCoin.exe
=== === Transaction @ 2021-05-17 23:44:10 === === ===
Sender:
3534B282 A8DB2C66 B97DB2C6 B51C41E7
7E395E42 FD753F78 86CC127D CD34D5F3
FC8280AC 2F79EA35 BFFD40F1 83E90850
75B13AE0 A68F7E1B DF5A1E41 18DC8B5D
3C789529 3427D828 FBEC43D1 4FD18046
9C3865D1 005BEC94 57E04C3E 195895B2
328AB1DA BE54953B 6EA46A12 730640FF
1ED48EA5 FC40A808 16AF1B34 E6B32FC2

Reciever:
1A3C0EEB 6AB560B8 E4D5984B D3E33032
147EC549 C9882758 0DBF3B29 75CA5F14
F58CD0F2 689808B7 C1382BD4 92378326
AD491FA2 F3F31E42 C382054A A7E90F8D
B4B7248D 97A327B1 75782198 3BE38EC4
3197D258 8465B234 0D44547D D66BFD92
96DD6928 DE1E08F3 4F61F817 DB1F48CF
2D4DFC95 8BCDE208 205512C3 FE98710C

Value: 100.00 GC
Fees: 0.0000 GC
Total Value: 100.0000 GC

--- Signature ---
r:
5A758579 9028F697 EDA2AB1D 67FE76D5 F9EB1243
s:
86746F07 0FFE960E E6B5197D 562E4573 6A81FABC
Valid signature!
=== === End Transaction @ 2021-05-17 23:44:10 === === ===
Press ENTER to close.

```

A transaction will take up exactly 328 bytes of data (4 of which are alignment bytes), and thus the cumulative size of all transactions on a block should be $328 \cdot 64 = 20,992$ bytes. An exported transaction, meaning a transaction saved to a file or broadcasted on the network, has 4 added bytes which are the magic. The magic allows the program to differentiate between data type stored in files, or on the network, by having the first two bytes be the ASCII values of “GC” (GreenCoin), then one byte about the type itself - a transaction having the code “T” (ASCII 84 \ 0x54) - and a final reserved byte which can be used in the future as a version indicator, though it is currently the char “3” (ASCII 51 \ 0x33).

Signing & Verification

The core piece of each transaction is its signature. The signature allows anyone who stumbles upon the transaction to verify its authenticity. The signature is of course a DSA signature as described [here](#). The signature is quickly checked each time a transaction is previewed, and a corresponding message will appear after the (r, s) signature pair at the bottom of the transaction.

In order to match and verify a signature, the public key of the sender must match the private key used to sign the transaction. This means that the person who's coins are being transferred from them has to also be the one to sign the transaction. The purpose of this is obviously to avoid coin theft.

From the user's side, signing is simply entering the private key which matches the public key.

Here is an example of a transaction being created, which will send 100GC from Alice's wallet

(yQXNOtIfvBL+mJFNzZJZGGgtqdTtsexoZazVfMMYHIsuR9gyQYIzn759pL14+qyeGdooUdewGczSYyugHI1FQu9AWgUKt6RGdfnUBkWP4DXhqFvac5ZnJpDeHPOpTpBsN2riMtJxKLZffW+VYhIARaOkqpbTud13sLPUQz1+jQ0=) to Bob's wallet (3EE08LXVh8lqibPY8hu0oEGhuBENLV8TyBC4698vy9D591DWD1FLi84ySBDttTbQzX2RMWTF6/IP1IRTacJAudXjGBOWniU9g1BRjkGTLPGY+HrZly5FHWcrkpO5WrRef+8LbrOW74/jfMoXJe42cqBS36oIquLGhDuyOgHckis=). Generating these two wallets right now to demonstrate this, as one should never publish or share their private key:

```

C:\Users\stein\Desktop\GreenCoin\Debug\GreenCoin.exe
> wallet
Now in wallet command line.
Type 'exit' to return to general command line.
> generate
x:
662E9852 52C04F1F EB6C00E7 BD7F5C47 B39342F9
X B64: '+UKTs0dcf73nAGzrH0/AUIKYLmY='
y:
0D8D7E3D 43D4B3B0 77DDB9D3 96AAA4A3 45001262 956F155F B62871D2 32E26A37
6C904EA9 F31CDE90 26679673 5A55A8E1 35E0A945 06D4F075 46A4B70A 055A40EF
42458D1C A02B63D2 CC19B0D7 5128DA19 9EACFA78 BDA47DBE 9F338241 32D8472E
8B1C18C3 7CD5AC65 68EC81ED D4A92D68 185992CD 4D9198FE 12BC1FD2 3ACD05C9
Y B64: 'yQXNOtIfvBL+mJFNzZJZGGgtqdTtsexoZazVfMMYHIsuR9gyQYIzn759pL14+qyeGdooUdewGczSYyugHI1FQu9AWgUKt6RGdfnUBkWP4DXhqFvac5ZnJpDeHPOpTpBsN2riMtJxKLZffW+VYhIARaOkqpbTud13sLPUQz1+jQ0='
> generate
x:
1BAD08D6 DE4FC886 8C0415A0 3F6778EA 43A38C49
X B64: 'SYyjQ+p4Zz+gFQSMhshP3tYrRs='
y:
2B92DC01 3AB23B84 C6E2AA08 AADF52A0 7236EE25 17CA7CE3 8FEF96B3 6E0BEF7F
5EB45AB9 93922B67 1D452E97 D97AF898 F12C9341 8E515083 3D259E96 1318E3D5
5140C269 5384D44F F9EBC564 31917DCD A036B5ED 104832CE 8B4B510F D650F7F9
D0CB2FDF EBB810C8 135F2D0D 11B8A141 A0B41BF2 D8B3896A C987D5B5 F00E41DC
Y B64: '3EE08LXVh8lqibPY8hu0oEGhuBENLV8TyBC4698vy9D591DWD1FLi84ySBDttTbQzX2RMWTF6/IP1IRTacJAudXjGBOWniU9g1BRjkGTLPGY+HrZly5FHWcrkpO5WrRef+8LbrOW74/jfMoXJe42cqBS36oIquLGhDuyOgHckis='
>

```

So Alice's private key is "+UKTs0dcf73nAGzrH0/AUIKYLmY=".

```

C:\Users\stein\Desktop\GreenCoin\Debug\GreenCoin.exe
5140C269 5384D44F F9EBC564 31917DCD D036B5ED 104832CE 8B4B510F D650F7F9
D0CB2FDF EBB810C8 135F2D0D 11B8A141 A0B41BF2 D8B3896A C987D5B5 F00E41DC
Y B64: '3EE08LXVh81qibPY8hu0oEGhuBENLV8TyBC4698vy9D591DWD1FLi84ySBDttTbQzX2RMWTF6/1P1IRTacJAUDXjGB0WniU9g1BRjkGTLPGY+HrZ
ly5FHWcrkp05WrRef+8Lbr0W74/jfMoXJe42cqBS36oIqulGhDuyOgHckis='
> exit
Exiting wallet command line.
> transact
=== Create Transaction ===
Block #?
2000
Please enter your public key (as base64):
yQXN0tIfvBL+mJFNzZJZGGgtqdTtsexoZazVfMMYHIsuR9gyQYIzn759pL14+qyeGdooUdewGczSYyugHI1FQu9AwgUKt6RGdfnUBKwp4DXhqFvac5ZnJpDe
HP0pTpBsN2riMtJxKLZfFW+VYhIARaOkqbTud13sLPUQz1+jQ0=
Please enter the reciever's public key (as base64):
3EE08LXVh81qibPY8hu0oEGhuBENLV8TyBC4698vy9D591DWD1FLi84ySBDttTbQzX2RMWTF6/1P1IRTacJAUDXjGB0WniU9g1BRjkGTLPGY+HrZly5FHWcr
kp05WrRef+8Lbr0W74/jfMoXJe42cqBS36oIqulGhDuyOgHckis=
How much would you like to send?
100
Please enter your private key (as base64) in order to sign this transaction:
+UKTs0dcf73nAGzrH0/AULKYLmY=
Transaction signed!

Transaction summary:
=== === Transaction @ 2021-06-09 02:24:25 === === ===
Sender:
0D8D7E3D 43D483B0 77DDB9D3 96AAA4A3
45001262 956F155F B62871D2 32E26A37
6C904EA9 F31CDE90 26679673 5A55A8E1
35E0A945 06D4F975 46A4B70A 055A40EF
42458D1C A02B63D2 CC19B0D7 5128DA19
9EACFA78 BDA47DBE 9F338241 32D8472E
8B1C18C3 7CD5AC65 68ECB1ED D4A92D68
185992CD 4D9198FE 12BC1FD2 3ACD05C9

Receiver:
2B92DC01 3AB23B84 C6E2AA08 AADF52A0
7236EE25 17CA7CE3 8FEF96B3 6E0BEF7F
5EB45A89 93922B67 1D452E97 D97AF898
F12C9341 8E515083 3D259E96 1318E3D5
5140C269 5384D44F F9EBC564 31917DCD
D036B5ED 104832CE 8B4B510F D650F7F9
D0CB2FDF EBB810C8 135F2D0D 11B8A141
A0B41BF2 D8B3896A C987D5B5 F00E41DC

Value: 100.00 GC
Fees: 0.0000 GC
Total Value: 100.0000 GC

--- Signature ---
r:
78E19960 52C19B99 6ADB77E 64A26810 27E322AD
s:
AD53CF54 148AEEBD 922760B9 1F1B9F71 5C020883
Valid signature!
=== === End Transaction @ 2021-06-09 02:24:25 === === ===

Type 'EXECUTE' to continue, otherwise cancel the transaction.

```

Signing

Signing the transaction simply means taking the data of the transaction, not including the signature:

```

char * transaction_info = (char*)malloc(sizeof(_Transaction) - sizeof(_Signature));

memcpy(transaction_info, transaction, sizeof(_Transaction) - sizeof(_Signature));

```

And producing a hash digest of this data. Doing so will entangle the signature to the transaction's actual data, which protects the signature from being tacked onto other transactions which might be fraudulent.

```
char * message_digest = Hash_SHA256(transaction_info, sizeof(_Transaction) -
sizeof(_Signature));
```

Now we simply use our [DSA signing function](#) to produce the (r, s) signature pair and tack it onto the transaction.

Verification

Verifying a transaction is even easier. All we need is the transaction data, including the signature. Then we perform the same operations, calculate the hash digest of the transaction data (not including the signature). This time, since we are verifying, we also deduce the public key of the signer from the transaction - this is simply the wallet address of the sender. We also extract the DSA signature pair (r, s) from the transaction and send everything off to the [DSA signature verification function](#), which will return a SIGNATURE_VALID_STATE of either valid or invalid.

Notice that verifying a transaction only checks its authenticity, meaning it only checks that the sender signed the transaction. Verification regarding funds is done by notaries and is part of block [verification](#) - the [blockchain](#).

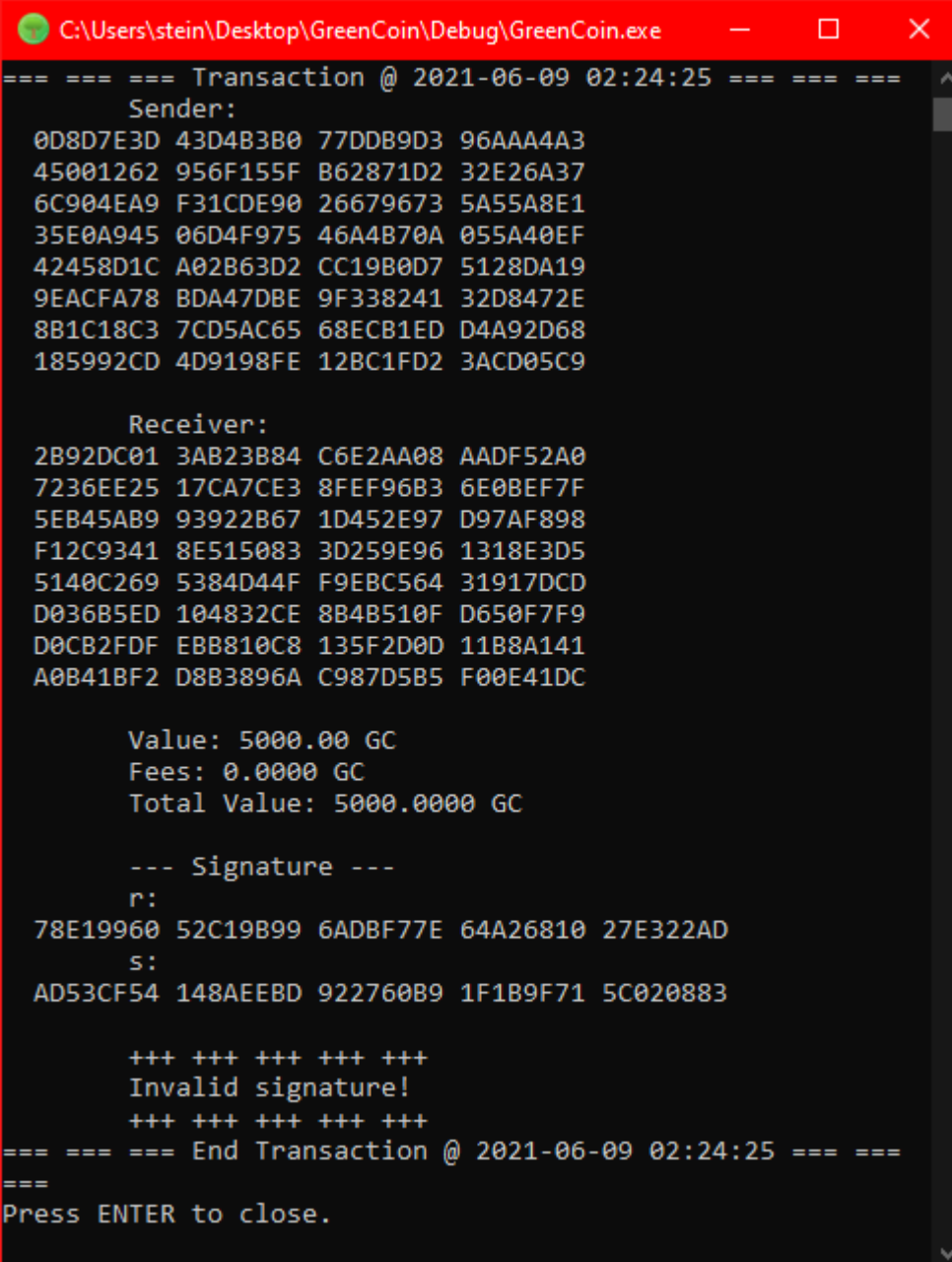
If a transaction was tampered with, the verification will catch the tamper. Assume after Alice sends the transaction, Bob tries to mess with it and changes the value of 100GC to 5,000GC.

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text	Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text
00000000	17 43 54 33 D0 07 00 00 00 00 00 29 FC BF 60	GCT13.....)Qz`	00000000	47 43 54 33 D0 07 00 00 00 00 00 29 FC BF 60	GCT13.....)Qz`
00000010	C9 05 CD 3A D2 1F BC 12 FE 98 91 4D CD 92 59 18	0 : : .4. . "M 'Y.	00000010	C9 05 CD 3A D2 1F BC 12 FE 98 91 4D CD 92 59 18	0 : : .4. . "M 'Y.
00000020	68 2D A9 D4 ED B1 EC 68 65 AC D5 7C C3 18 1C 8B	h-@7z0nhe- [n...<	00000020	68 2D A9 D4 ED B1 EC 68 65 AC D5 7C C3 18 1C 8B	h-@7z0nhe- [n...<
00000030	2E 47 D8 32 41 82 33 9F BE 7D A4 BD 78 FA AC 9E	.G2-A,3,4)shxn-	00000030	2E 47 D8 32 41 82 33 9F BE 7D A4 BD 78 FA AC 9E	.G2-A,3,4)shxn-
00000040	19 DA 28 51 D7 B0 19 CC D2 63 2B A0 1C 8D 45 42	.Q(Q 0.?'<+ ..EB	00000040	19 DA 28 51 D7 B0 19 CC D2 63 2B A0 1C 8D 45 42	.Q(Q 0.?'<+ ..EB
00000050	EF 40 5A 05 0A B7 A4 46 75 F9 D4 06 45 A9 E0 35	182...mFune.E05x	00000050	EF 40 5A 05 0A B7 A4 46 75 F9 D4 06 45 A9 E0 35	182...mFune.E05x
00000060	E1 A9 55 5A 73 96 67 26 90 DE 1C F3 A9 4E 90 6C	3"UZs-g4.Q.q8N.1	00000060	E1 A9 55 5A 73 96 67 26 90 DE 1C F3 A9 4E 90 6C	3"UZs-g4.Q.q8N.1
00000070	37 6A E2 32 D2 71 28 B6 5F 15 6F 95 62 12 00 45	7j021q(1...o'b..E	00000070	37 6A E2 32 D2 71 28 B6 5F 15 6F 95 62 12 00 45	7j021q(1...o'b..E
00000080	A3 A4 AA 96 D3 B9 DD 77 B0 B3 D4 43 3D 7E 8D 0D	80x-':Qwn*?Cm-..	00000080	A3 A4 AA 96 D3 B9 DD 77 B0 B3 D4 43 3D 7E 8D 0D	80x-':Qwn*?Cm-..
00000090	DC 41 0E F0 B5 D5 87 C9 6A 89 B3 D8 F2 1B B4 A0	QA.µµ *µjµ'h'.<	00000090	DC 41 0E F0 B5 D5 87 C9 6A 89 B3 D8 F2 1B B4 A0	QA.µµ *µjµ'h'.<
000000A0	41 A1 B8 11 0D 2D 5F 13 C8 10 B8 EB DF 2F CB D0	Al...-..>..D/ /	000000A0	41 A1 B8 11 0D 2D 5F 13 C8 10 B8 EB DF 2F CB D0	Al...-..>..D/ /
000000B0	F9 F7 50 D6 0F 51 4B 8B CE 32 48 10 ED B5 36 D0	p8P...QK(2-H.µµ16	000000B0	F9 F7 50 D6 0F 51 4B 8B CE 32 48 10 ED B5 36 D0	p8P...QK(2-H.µµ16
000000C0	CD 7D 91 31 64 C5 EB F9 4F D4 84 53 69 C2 40 51	1' {dpg 0µ.S1 @Q	000000C0	CD 7D 91 31 64 C5 EB F9 4F D4 84 53 69 C2 40 51	1' {dpg 0µ.S1 @Q
000000D0	D5 E3 18 13 96 9E 25 3D 83 50 51 8E 41 93 2C F1	µµ...µ=fPQ.A",µ	000000D0	D5 E3 18 13 96 9E 25 3D 83 50 51 8E 41 93 2C F1	µµ...µ=fPQ.A",µ
000000E0	98 F8 7A D9 97 2E 45 1D 67 2B 92 93 B9 5A B4 5E	"µµ-µ.g+µ"Z'^	000000E0	98 F8 7A D9 97 2E 45 1D 67 2B 92 93 B9 5A B4 5E	"µµ-µ.g+µ"Z'^
000000F0	7F EF 0B 6E B3 96 EF 8F E3 7C CA 17 25 EE 36 72	.1.n'-6n.0 µ.1x	000000F0	7F EF 0B 6E B3 96 EF 8F E3 7C CA 17 25 EE 36 72	.1.n'-6n.0 µ.1x
00000100	A0 52 DF AA 08 AA E2 C6 84 3B B2 3A 01 DC 92 2B	RDµ.*µ;µ µ:µ'+	00000100	A0 52 DF AA 08 AA E2 C6 84 3B B2 3A 01 DC 92 2B	RDµ.*µ;µ µ:µ'+
00000110	CC CC CC CC 00 00 00 00 00 00 00 00 00 00 00 00µµ...	00000110	CC CC CC CC 00 00 00 00 00 00 00 00 00 00 00 00µµ...
00000120	00 00 59 40 AD 22 E3 27 10 68 A2 64 7E F7 DB 6A	..Y8."µ'.hed-µQj	00000120	00 88 B3 40 AD 22 E3 27 10 68 A2 64 7E F7 DB 6A	..Y8."µ'.hed-µQj
00000130	99 9B C1 52 60 99 E1 78 83 08 02 5C 71 9F 1B 1F	=> µ'µµxµf..µq...	00000130	99 9B C1 52 60 99 E1 78 83 08 02 5C 71 9F 1B 1F	=> µ'µµxµf..µq...
00000140	B9 60 27 92 BD EE 8A 14 54 CF 53 AD	µ'µµsn..T S.	00000140	B9 60 27 92 BD EE 8A 14 54 CF 53 AD	µ'µµsn..T S.

00 00 00 00 00 00 59 40 = 100

00 00 00 00 00 88 B3 40 = 5,000

The signature itself was really generated by Alice and matches her private-public key pair. However, since the signature is dependant on the [hash](#) of the transaction, the following occurs:



```

C:\Users\stein\Desktop\GreenCoin\Debug\GreenCoin.exe
=== === === Transaction @ 2021-06-09 02:24:25 === === ===
Sender:
0D8D7E3D 43D4B3B0 77DDB9D3 96AAA4A3
45001262 956F155F B62871D2 32E26A37
6C904EA9 F31CDE90 26679673 5A55A8E1
35E0A945 06D4F975 46A4B70A 055A40EF
42458D1C A02B63D2 CC19B0D7 5128DA19
9EACFA78 BDA47DBE 9F338241 32D8472E
8B1C18C3 7CD5AC65 68ECB1ED D4A92D68
185992CD 4D9198FE 12BC1FD2 3ACD05C9

Receiver:
2B92DC01 3AB23B84 C6E2AA08 AADF52A0
7236EE25 17CA7CE3 8FEF96B3 6E0BEF7F
5EB45AB9 93922B67 1D452E97 D97AF898
F12C9341 8E515083 3D259E96 1318E3D5
5140C269 5384D44F F9EBC564 31917DCD
D036B5ED 104832CE 8B4B510F D650F7F9
D0CB2FDF EBB810C8 135F2D0D 11B8A141
A0B41BF2 D8B3896A C987D5B5 F00E41DC

Value: 5000.00 GC
Fees: 0.0000 GC
Total Value: 5000.0000 GC

--- Signature ---
r:
78E19960 52C19B99 6ADBF77E 64A26810 27E322AD
s:
AD53CF54 148AEEBD 922760B9 1F1B9F71 5C020883

+++ +++ +++ +++ +++
Invalid signature!
+++ +++ +++ +++ +++
=== === === End Transaction @ 2021-06-09 02:24:25 === ===
===
Press ENTER to close.

```

The GreenCoin program recognizes that the transaction is fraudulent and rejects the signature.

This is due to the fact that even such a small change, only two bytes, changed the hash from

“3575ce08858deb9c469e3b34cb2fa5e95247827aae650d0fca770e8642cc7584” to

“79dae68885019d77bf26784c91d1fe130bd9a7da0138b07db9e96f0474cb6f99”.

Code

Transaction.h + Transaction.c


```
#pragma once

#ifndef __TRANSACTION_H
#define __TRANSACTION_H

#include "../Wallet/Wallet.h"

typedef struct _Signature {

    uint_t r[5];

    uint_t s[5];

} _Signature;

typedef struct _Transaction {

    // Index of the block in the chain

    uint64_t Block_Index;

    /* DEPRECATED

    // Index of the transaction on the ledger

    uint32_t Index;

    */

    // The time this transaction was signed.

    uint32_t Time;
```

```
/*  
    The combination of the block_index and the index are completely unique in the  
chain,  
    and therefore protect the signature from being copied either again on the same  
block  
    or at the same position on a different block.  
*/  
  
// Address of the sender  
_Wallet_Address Sender;  
  
// Address of the receiver  
_Wallet_Address Receiver;  
  
// Fees - This can currently be disregarded  
double Fee;  
  
// Value of the transaction in GreenCoins  
double Value;  
  
// Signature in order to validate sender  
_Signature Signature;  
} _Transaction;  
  
void Print_Transaction_Signature_Part(FILE * fstream, uint_t * q);  
void Print_Transaction_Wallet_Address(FILE * fstream, uint_t * q);
```

```
void Print_Transaction(FILE * fstream, _Transaction * transaction);

void Transaction_Export(FILE * fstream, _Transaction * transaction);

void Transaction_Export_To_File(char * file_path, _Transaction * transaction);

void Sign_Transaction(_Transaction * transaction, BN * priv_key);

// Returns the value of the transaction with regards to a wallet.
// If the wallet was the sender, deduce the TOTAL value of the transaction.
// If the wallet was the receiver, add the value of the transaction.

double Calculate_Transaction_Change_To_Wallet(_Transaction * transaction,
_Wallet_Address pk);

SIGNATURE_VALID_STATE Verify_Transaction(_Transaction * transaction);

void Transaction_Demo(void * wsadata, void * socket);

#endif // !__TRANSACTION_H
```

```
#include "Transaction.h"
```

```
#include "../General/Print/PrettyPrint.h"
```

```
void Print_Transaction_Signature_Part(FILE * fstream, uint_t * q) {

    for (int i = 4; i >= 0; i--) {

        fprintf(fstream, "%.8X ", q[i]);

    }

    fprintf(fstream, "\n");

}

void Print_Transaction_Wallet_Address(FILE * fstream, uint_t * q) {

    fprintf(fstream, " ");

    for (int i = 31; i >= 0; i--) {

        fprintf(fstream, "%.8X ", q[i]);

        if (i % 4 == 0) { fprintf(fstream, "\n "); }

    }

    fprintf(fstream, "\n");

}

void Print_Transaction(FILE * fstream, _Transaction * transaction) {

    fprintf(fstream, "==== Transaction @ %s ====\n",
HumanFormatDateTimeInt(transaction->Time));

    fprintf(fstream, "\tSender: \n");

    Print_Transaction_Wallet_Address(fstream, transaction->Sender);

    fprintf(fstream, "\tReciever: \n");

    Print_Transaction_Wallet_Address(fstream, transaction->Receiver);

    fprintf(fstream, "\tValue: %.2f GC\n", transaction->Value);

    fprintf(fstream, "\tFees: %.4f GC\n", transaction->Fee);

    fprintf(fstream, "\tTotal Value: %.4f GC\n", transaction->Value + transaction->Fee);

    fprintf(fstream, "\n\t--- Signature ---\n");

}
```

```
fprintf(fstream, "\\tr:\\n ");

Print_Transaction_Signature_Part(fstream, transaction->Signature.r);

fprintf(fstream, "\\ts:\\n ");

Print_Transaction_Signature_Part(fstream, transaction->Signature.s);


DSA_Public_Key * pub_key=0;

DSA_Init_Public_Key(&pub_key);

//DSA_Load_Public_Key(pub_key, params->p, params->q, params->G);

BN_Resize(pub_key, 32);

memcpy(pub_key->data, transaction->Sender, 32 * sizeof(uint_t));


DSA_Signature * sign;

DSA_Init_Signature(&sign);

BN_Resize(sign->r, 5);

BN_Resize(sign->s, 5);

memcpy(sign->r->data, transaction->Signature.r, 5 * sizeof(uint_t));

memcpy(sign->s->data, transaction->Signature.s, 5 * sizeof(uint_t));


char * digest = Hash_SHA256(transaction, sizeof(_Transaction) -
sizeof(_Signature));


SIGNATURE_VALID_STATE valid = DSA_Verify_Signature(pub_key, digest, 64,
sign);

if (valid == SIGNATURE_VALID) {

fprintf(fstream, "\\tValid signature!\\n");
```

```
    }

    else {

        fprintf(fstream, "\n\t+++ +++ +++ +++ +++\n\tInvalid signature!\n\t+++ +++ +++
+++ +++\n");

    }


    fprintf(fstream, "==== End Transaction @ %s ==== \n",
HumanFormatDateTimeInt(transaction->Time));
}

void Transaction_Export(FILE * fstream, _Transaction * transaction) {

    fwrite(GCT_MAGIC, 1, 4, fstream);

    fwrite(transaction, sizeof(_Transaction), 1, fstream);

}

void Transaction_Export_To_File(char * file_path, _Transaction * transaction) {

    FILE * ft;

    fopen_s(&ft, file_path, "wb");

    // Write magic

    fwrite(GCT_MAGIC, 1, 4, ft);

    // Write transaction data

    fwrite(transaction, sizeof(_Transaction), 1, ft);
```

```
    fclose(ft);
}

void Sign_Transaction(_Transaction * transaction, BN * pk) {

    char * transaction_info = (char*)malloc(sizeof(_Transaction) - sizeof(_Signature));

    memcpy(transaction_info, transaction, sizeof(_Transaction) - sizeof(_Signature));

    //char * message = "abc";

    char * message_digest = Hash_SHA256(transaction_info, sizeof(_Transaction) -
sizeof(_Signature));//ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f2
0015ad";

    DSA_Signature * signature = 0;

    DSA_Init_Signature(&signature);

    DSA_Private_Key * priv_key;

    priv_key = pk;

    DSA_Generate_Signature(priv_key, message_digest, 64, signature);

    _Signature sig;

    memcpy((sig.r), signature->r->data, 5 * sizeof(uint_t));

    memcpy((sig.s), signature->s->data, 5 * sizeof(uint_t));
```

```
memcpy(&(transaction->Signature), &sig, sizeof(_Signature));

free(transaction_info);

//DSA_Free_Private_Key(priv_key);
}

SIGNATURE_VALID_STATE Verify_Transaction(_Transaction * transaction) {

    DSA_Public_Key * pub_key = 0;

    DSA_Init_Public_Key(&pub_key);

    BN_Resize(pub_key, 32);

    memcpy(pub_key->data, transaction->Sender, 32 * sizeof(uint_t));

    DSA_Signature * sign;

    DSA_Init_Signature(&sign);

    BN_Resize(sign->r, 5);

    BN_Resize(sign->s, 5);

    memcpy(sign->r->data, transaction->Signature.r, 5 * sizeof(uint_t));

    memcpy(sign->s->data, transaction->Signature.s, 5 * sizeof(uint_t));

    char * digest = Hash_SHA256(transaction, sizeof(_Transaction) -
sizeof(_Signature));

    SIGNATURE_VALID_STATE valid = DSA_Verify_Signature(pub_key, digest, 64,
sign);
```



```
    DSA_Free_Public_Key(pub_key);

    DSA_Free_Signature(sign);

    return valid;
}

double Calculate_Transaction_Change_To_Wallet(_Transaction * transaction,
_Wallet_Address pk) {

    if (memcmp(transaction->Sender, pk, 32 * sizeof(uint_t)) == 0) {

        return -(transaction->Value + transaction->Fee);

    }

    else if (memcmp(transaction->Receiver, pk, 32 * sizeof(uint_t)) == 0) {

        return transaction->Value;

    }

    else {

        return 0;

    }

    return 0;

}

void Transaction_Demo(void * wsadata, void * socket) {

    printf("=== Create Transaction ===\n");

    _Transaction transaction;
```

```
printf("Block #?\n");

char buffer[64]; fgets(buffer, 64, stdin);

transaction.Block_Index = strtol(buffer, NULL, 10);


/*printf("Transaction #?\n");

fgets(buffer, 64, stdin);

transaction.Index = strtol(buffer, NULL, 10);*/


printf("Please enter your public key (as base64): \n");

char sender_64[180]; fgets(sender_64, 180, stdin);


printf("Please enter the receiver's public key (as base64): \n");

char reciever_64[180]; fgets(reciever_64, 180, stdin);


printf("How much would you like to send?\n");

fgets(buffer, 64, stdin);

double value = strtod(buffer, NULL);

transaction.Value = value;

transaction.Fee = 0;


byte * sender;

byte * receiver;

B64_Decode(sender_64, &sender);

B64_Decode(reciever_64, &receiver);
```

```
memcpy(transaction.Sender, sender, sizeof(_Wallet_Address));  
memcpy(transaction.Receiver, receiver, sizeof(_Wallet_Address));  
  
printf("Please enter your private key (as base64) in order to sign this transaction:  
\n");  
  
char priv_key_64[180]; fgets(priv_key_64, 180, stdin);  
  
byte * priv_k; size_t priv_key_byte_length = B64_Decode(priv_key_64, &priv_k);  
  
BN * pk; BN_Init(&pk);  
  
BN_Resize(pk, 16);  
  
pk->sign = 1;  
  
memcpy(pk->data, priv_k, priv_key_byte_length);  
  
  
transaction.Time = time(NULL);  
  
  
Sign_Transaction(&transaction, pk);  
  
  
printf_Success("Transaction signed!\n");  
  
  
printf("\n\n\n");  
  
printf("Transaction summary: \n");  
  
  
Print_Transaction(stderr, &transaction);  
  
printf("\n\n\n");
```

```
printf("Type 'EXECUTE' to continue, otherwise cancel the transaction.\n");

char * EXECUTE = "EXECUTE";

fgets(buffer, 64, stdin); buffer[strlen(EXECUTE)] = 0x0;

if (strcmp(buffer, EXECUTE) == 0) {
    printf_Success("Transaction executed!\n");

    char export_path[256];
    sprintf_s(export_path, 256, "Demo\\Transaction_%u.GCT", transaction.Time);

    printf_Info("Now exporting to: '%s'\n", export_path);

    Transaction_Export_To_File(export_path, &transaction);

    Network_Broadcast_Transaction(wsadata, socket, &transaction,
sizeof(_Transaction));
}
else {
    printf_Info("Transaction has been cancelled!\n");
}
}
```

Blocks

Mining

Wallets

Base-64 Encoding

Code

Base64.h + Base64.c

```
#pragma once

#ifndef __BASE_64_H
#define __BASE_64_H

#include "../General.h"

const char BASE64_CHAR_SEQUENCE[64];
static const signed char BASE64_REVERSE_SEQUENCE[256];

// Returns the size of something to be converted ~ 4/3 * len
size_t B64_Encoded_Size(size_t len);

// Returns the size of something to be decoded ~ 3/4 * len
size_t B64_Decoded_Size(char * base64_array);

/* Encode bytes to base64 char array
   Returns the length of the base64 char array.
   Allocates buffer. */
size_t B64_Encode(byte * input, size_t input_length, char ** output);

/* Decodes base64 char array to byte array
   Returns the length of the char array.
   Allocates buffer. */
size_t B64_Decode(char * input, byte ** output);

#endif // !__BASE_64
```

```
#include "Base64.h"
```

[illegible]

```
    size_t length = strlen(base64_array);

    if (length < 1) { return 0; }

    int padding_count = (int)(base64_array[length - 1] == '=') +
(int)(base64_array[length - 2] == '=');

    length = ((length / 4) * 3) - padding_count;

    return length;
}

size_t B64_Encode(byte * input, size_t input_length, char ** output)
{
    char * out, * pos;

    byte * end, * in;

    size_t out_len;

    out_len = 4 * ((input_length + 2) / 3);

    if (out_len < input_length) {
        return -1;
    }

    *output = (char*)malloc(out_len + 1);
    out = *output;

    end = input + input_length;
    in = input;
    pos = out;

    while (end - in >= 3) {
        *pos++ = BASE64_CHAR_SEQUENCE[in[0] >> 2];
        *pos++ = BASE64_CHAR_SEQUENCE[((in[0] & 0x03) << 4) | (in[1] >> 4)];
        *pos++ = BASE64_CHAR_SEQUENCE[((in[1] & 0x0f) << 2) | (in[2] >> 6)];
        *pos++ = BASE64_CHAR_SEQUENCE[in[2] & 0x3f];
        in += 3;
    }

    if (end - in) {
        *pos++ = BASE64_CHAR_SEQUENCE[in[0] >> 2];
```

```
        if (end - in == 1) {
            *pos++ = BASE64_CHAR_SEQUENCE[(in[0] & 0x03) << 4];
            *pos++ = '=';
        }
        else {
            *pos++ = BASE64_CHAR_SEQUENCE[((in[0] & 0x03) << 4) |
            (in[1] >> 4)];
            *pos++ = BASE64_CHAR_SEQUENCE[(in[1] & 0x0f) << 2];
        }
        *pos++ = '=';
    }

    *pos = 0x0;

    return out_len;
}

size_t B64_Decode(char * input, byte ** output) {
    size_t input_length = strlen(input);
    size_t length = B64_Decoded_Size(input);

    *output = (byte*)malloc(length+1);
    char * out = *output;

    char * pos = out;
    char * end = out + length;
    char * in = input;

    while (end - pos >= 3) {
        // 4 base64 -> 3 char
        // 4 * 6bits => 3 * 8bits = 24bits
        pos[0] = ((BASE64_REVERSE_SEQUENCE[in[0]] << 2) |
        (BASE64_REVERSE_SEQUENCE[in[1]] >> 4));
        pos[1] = (((BASE64_REVERSE_SEQUENCE[in[1]] & 0x0f) << 4) |
        (BASE64_REVERSE_SEQUENCE[in[2]] >> 2));
        pos[2] = (((BASE64_REVERSE_SEQUENCE[in[2]] & 0x03) << 6) |
        (BASE64_REVERSE_SEQUENCE[in[3]]));

        pos += 3;
        in += 4;
    }

    if (end - pos) {
```



```
// =
int padding_count = (int)(input[input_length - 1] == '=') + (int)(input[input_length -
2] == '=');
if (padding_count == 1) {
    pos[0] = ((BASE64_REVERSE_SEQUENCE[in[0]] << 2) |
(BASE64_REVERSE_SEQUENCE[in[1]] >> 4));
    pos[1] = (((BASE64_REVERSE_SEQUENCE[in[1]] & 0x0f) << 4) |
(BASE64_REVERSE_SEQUENCE[in[2]] >> 2));
    pos += 2;
    in += 2;
}
else if (padding_count == 2) {
    pos[0] = ((BASE64_REVERSE_SEQUENCE[in[0]] << 2) |
(BASE64_REVERSE_SEQUENCE[in[1]] >> 4));
    pos++;
    in += 2;
}
else {
    // Error?
}
}

pos[0] = 0x0;

return length;
}
```

Network

TCP

P2P

Works Cited

- “4.6 — Fixed-Width Integers and Size_t | Learn C++.” *Learn CPP*, www.learncpp.com/cpp-tutorial/fixed-width-integers-and-size-t. Accessed 9 June 2021.
- “7.15 Using the Digital Signature Algorithm (DSA).” *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*, flylib.com/books/en/2.878.1.133/1. Accessed 27 May 2021.
- Anssi-Fr. “ANSSI-FR/Libecc.” *GitHub*, github.com/ANSSI-FR/libecc/blob/master/src/sig/ecdsa.c. Accessed 27 May 2021.
- “Are There Public p and q Numbers for Use in DSA?” *Cryptography Stack Exchange*, 11 Nov. 2013, crypto.stackexchange.com/questions/11655/are-there-public-p-and-q-numbers-for-use-in-dsa.
- “Ascii Table - ASCII Character Codes and Html, Octal, Hex and Decimal Chart Conversion.” *ASCII Table*, www.asciitable.com. Accessed 27 May 2021.
- “Base64 Decode and Encode - Online.” *Base64 Decode*, www.base64decode.org. Accessed 27 May 2021.
- “The Best Way to Check If a File Exists Using Standard C/C++.” *Tutorials Point*, www.tutorialspoint.com/the-best-way-to-check-if-a-file-exists-using-standard-c-cplusplus. Accessed 27 May 2021.
- “Bit Shifting (Left Shift, Right Shift) | Interview Cake.” *Interview Cake: Programming Interview Questions and Tips*, www.interviewcake.com/concept/java/bit-shift. Accessed 27 May 2021.
- “Bitcoin (BTC) and United States Dollar (USD) Year 2021 Exchange Rate History. Free Currency Rates (FCR).” *Copyright Mobilesoftjungle.Com, MobileSoftJungle Ltd.*, freecurrencyrates.com/en/exchange-rate-history/BTC-USD/2021. Accessed 9 June 2021.
- “C: Is There Anyway i Can Get the modulo Operator to Work on Non Integer Values?” *Stack Overflow*, 26 June 2020, stackoverflow.com/questions/62597353/c-is-there-anyway-i-can-get-the-modulo-operator-to-work-on-non-integer-values.
- “C Library Function - Ceil() - Tutorialspoint.” *Tutorials Point*, www.tutorialspoint.com/c_standard_library/c_function_ceil.htm. Accessed 27 May 2021.

- “C Library Function - Fgets() - Tutorialspoint.” *Tutorials Point*,
www.tutorialspoint.com/c_standard_library/c_function_fgets.htm. Accessed 9 June 2021.
- “C Library Function - Fopen() - Tutorialspoint.” *Tutorials Point*,
www.tutorialspoint.com/c_standard_library/c_function_fopen.htm. Accessed 27 May 2021.
- “Compilation Error, ‘Uses Undefined Struct.’” *Stack Overflow*, 12 Jan. 2013,
stackoverflow.com/questions/14298417/compilation-error-uses-undefined-struct.
- “Digital Signature Algorithm (DSA and ECDSA) — PyCryptodome 3.9.9 Documentation.”
PyCryptodome 3.9.9 Documentation,
pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html#:~:text=For%20ECDSA%2C%20the%20signature%20is,bytes%20for%20P%2D256. Accessed 27 May 2021.
- “DSA Generate Keys, Generate Signature and Verify Signature File.” *8gwifi*, 4 Mar. 2018,
8gwifi.org/DSAFunctionality?keysize=512.
- “Dsa.c Source Code - DSA (Digital Signature Algorithm).” *ORYX*,
oryx-embedded.com/doc/dsa_8c_source.html. Accessed 27 May 2021.
- Edpresso Team. “C: Reading Data from a File Using Fread().” *Educative: Interactive Courses for Software Developers*, 23 Apr. 2021,
www.educative.io/edpresso/c-reading-data-from-a-file-using-fread.
- “Elliptic Curve Digital Signature Algorithm - Bitcoin Wiki.” *Bitcoin Wiki*, 2011,
en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm.
- Eric O Meehan. “Creating a Peer to Peer Network in C.” *YouTube*, 14 Mar. 2021,
www.youtube.com/watch?v=oHBi8k31fgM.
- “Free Cartoon Pictures Of Trees, Download Free Cartoon Pictures Of Trees Png Images, Free ClipArts on Clipart Library.” *Clipart Library*,
clipart-library.com/cartoon-pictures-of-trees.html. Accessed 27 May 2021.
- “Generate Random Prime Numbers.” *A Security Site*,
asecuritysite.com/encryption/random3?val=8. Accessed 27 May 2021.
- Hertig, Alyssa. “What Is Bitcoin Halving? Here’s Everything You Need to Know –.”
CoinDesk, 17 Dec. 2020, www.coindesk.com/bitcoin-halving-explainer.
- “How Do Banks Become Insolvent?” *Positive Money*, 9 Nov. 2018,
positivemoney.org/how-money-works/advanced/how-do-banks-become-insolvent/#:~:text=For%20a%20bank%2C%20being%20insolvent,the%20availability%20of%20deposit%20insurance.

- “How Do I Base64 Encode (Decode) in C?” *Stack Overflow*, 4 Dec. 2008, stackoverflow.com/questions/342409/how-do-i-base64-encode-decode-in-c.
- “How Is the Generator Point ‘G’ in ECDSA Generated?” *Reddit*, 28 Aug. 2019, www.reddit.com/r/cryptography/comments/cwh723/how_is_the_generator_point_g_in_ecdsa_generated.
- “Index Of /.” *Apache*, svn.apache.org. Accessed 27 May 2021.
- Jakob Jenkov. “Peer-to-Peer (P2P) Networks - Basic Algorithms.” *YouTube*, 20 Feb. 2012, www.youtube.com/watch?v=kXyVqk3EbwE.
- Leandro Junes. “Applied Cryptography: The Digital Signature Algorithm - Part 1.” *YouTube*, 1 Dec. 2016, www.youtube.com/watch?v=PQ8AruHaoLo.
- Lisk. “What Is a Peer to Peer Network? Blockchain P2P Networks Explained.” *YouTube*, 15 Jan. 2019, www.youtube.com/watch?v=ie-qRQIQT4I.
- Littlstar. “Littlstar/B64.c.” *GitHub*, github.com/littlstar/b64.c/blob/master/decode.c. Accessed 27 May 2021.
- LiveOverflow. “Breaking ECDSA (Elliptic Curve Cryptography) - Rhme2 Secure Filesystem v1.92r1 (Crypto 150).” *YouTube*, 19 May 2017, www.youtube.com/watch?v=-UcCMjQab4w.
- “Mastering Bitcoin.” *O’Reilly Online Learning*, www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html. Accessed 27 May 2021.
- “National Institute of Standards and Technology.” *NIST*, 20 May 2021, www.nist.gov.
- Noogin. “The Financial Crisis and History of Bitcoin - Noogin.” *Medium*, 15 May 2018, medium.com/@noogin/the-financial-crisis-and-history-of-bitcoin-27ebdb932b99#:~:text=Bitcoin%20was%20invented%20in%20the,the%20expense%20of%20their%20taxpayers.
- “Reverse Sha256 Hash.” *Hashtoolkit.Com*, hashtoolkit.com/reverse-sha256-hash/65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5. Accessed 27 May 2021.
- “Set an Exe Icon for My Program.” *Stack Overflow*, 6 Mar. 2010, stackoverflow.com/questions/2393863/set-an-exe-icon-for-my-program/26130514.
- “SHA-256 Hash Calculator.” *Xorbin*, xorbin.com/tools/sha256-hash-calculator. Accessed 27 May 2021.

“SHA256 Online.” *Github*, emn178.github.io/online-tools/sha256.html. Accessed 27 May 2021.

“---.” *Github*, emn178.github.io/online-tools/sha256.html. Accessed 27 May 2021.

Simply Explained. “How Bitcoin Wallets Work (Public & Private Key Explained).” *YouTube*, 6 Aug. 2019, www.youtube.com/watch?v=GSTiKjnBaes.

---. “How Bitcoin Wallets Work (Public & Private Key Explained).” *YouTube*, 6 Aug. 2019, www.youtube.com/watch?v=GSTiKjnBaes.

Statista. “Bitcoin (BTC) Price History from 2013 to June 8, 2021.” *Statista*, 8 June 2021, www.statista.com/statistics/326707/bitcoin-price-index.

---. “United States - Monthly Inflation Rate in April 2021.” *Statista*, 12 May 2021, www.statista.com/statistics/273418/unadjusted-monthly-inflation-rate-in-the-us.

Stevewhims. “Complete Winsock Server Code - Win32 Apps.” *Microsoft Docs*, 31 May 2018, docs.microsoft.com/en-us/windows/win32/winsock/complete-server-code.

Terrazzoni, Jean-Baptiste. “Implementing the Sha256 and Md5 Hash Functions in C.” *Medium*, 28 Apr. 2021, medium.com/a-42-journey/implementing-the-sha256-and-md5-hash-functions-in-c-78c17e657794.

“Transmission Control Protocol (TCP) (Article).” *Khan Academy*, www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp. Accessed 9 June 2021.

“Typedef Fixed Length Array.” *Stack Overflow*, 24 Dec. 2010, stackoverflow.com/questions/4523497/typedef-fixed-length-array/4523537.

“What Is the Difference between Fwrite and Fprintf in C? - Quora.” *Quora*, [www.quora.com/What-is-the-difference-between-fwrite-and-fprintf-in-C#:~:text=W hat%20is%20the%20difference%20between%20a%20fprintf%20\(\)%20and%20a,other%20arrays](https://www.quora.com/What-is-the-difference-between-fwrite-and-fprintf-in-C#:~:text=W hat%20is%20the%20difference%20between%20a%20fprintf%20()%20and%20a,other%20arrays). Accessed 27 May 2021.

Wikipedia contributors. “Bitcoin Scalability Problem.” *Wikipedia*, 25 May 2021, en.wikipedia.org/wiki/Bitcoin_scalability_problem.

---. “C Data Types.” *Wikipedia*, 30 Apr. 2021, en.wikipedia.org/wiki/C_data_types.

---. “Convergent Series.” *Wikipedia*, 8 May 2021, en.wikipedia.org/wiki/Convergent_series.

---. “Digital Signature Algorithm.” *Wikipedia*, 17 May 2021, en.wikipedia.org/wiki/Digital_Signature_Algorithm.

- . "Double-Precision Floating-Point Format." *Wikipedia*, 24 May 2021, en.wikipedia.org/wiki/Double-precision_floating-point_format.
- . "Elliptic Curve Digital Signature Algorithm." *Wikipedia*, 26 May 2021, en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.
- . "Elliptic-Curve Cryptography." *Wikipedia*, 4 May 2021, en.wikipedia.org/wiki/Elliptic-curve_cryptography.
- . "Hash Function." *Wikipedia*, 27 May 2021, en.wikipedia.org/wiki/Hash_function#:~:text=A%20hash%20function%20is%20any,table%20called%20a%20hash%20table.
- . "Modular Exponentiation." *Wikipedia*, 15 Jan. 2021, en.wikipedia.org/wiki/Modular_exponentiation.
- . "Natural Number." *Wikipedia*, 26 May 2021, en.wikipedia.org/wiki/Natural_number.
- . "One-Way Function." *Wikipedia*, 19 May 2021, en.wikipedia.org/wiki/One-way_function.
- . "Peer-to-Peer." *Wikipedia*, 9 May 2021, en.wikipedia.org/wiki/Peer-to-peer.
- . "Printf Format String." *Wikipedia*, 29 Apr. 2021, en.wikipedia.org/wiki/Printf_format_string.
- . "Satoshi Nakamoto." *Wikipedia*, 24 May 2021, en.wikipedia.org/wiki/Satoshi_Nakamoto.
- . "SHA-2." *Wikipedia*, 19 May 2021, en.wikipedia.org/wiki/SHA-2.

Digital Signature Standard (DSS), U.S. Dept. of Commerce, National Institute of Standards and Technology, 2009.

Links:

<https://blockgeeks.com/guides/cryptocurrency-wallet-guide/>
<https://www.youtube.com/watch?v=GSTiKjnBaes>
https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm
https://www.google.com/search?q=double&rlz=1C1CHBF_enCH781CH781&oq=double&aqs=chrome..69j69l69j46l69j175i199j0l4j46l2.881j0j7&sourceid=chrome&ie=UTF-8
[https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html#:~:text=For%20ECDSA%2C%20the%20signature%20is,bytes%20for%20P%2D256\).](https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html#:~:text=For%20ECDSA%2C%20the%20signature%20is,bytes%20for%20P%2D256).)
<https://stackoverflow.com/questions/14298417/compilation-error-uses-undefined-struct>

<https://github.com/ANSSI-FR/libecc/blob/master/src/sig/ecdsa.c>
https://en.wikipedia.org/wiki/C_data_types
https://en.wikipedia.org/wiki/Double-precision_floating-point_format
<https://www.youtube.com/watch?v=-UcCMjQab4w>
<https://xorbin.com/tools/sha256-hash-calculator>
<https://medium.com/a-42-journey/implementing-the-sha256-and-md5-hash-functions-in-c-78c17e657794>
https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
https://www.reddit.com/r/cryptography/comments/cwh723/how_is_the_generator_point_g_in_ecdsa_generated/
https://csrc.nist.gov/csrc/media/publications/fips/186/3/archive/2009-06-25/documents/fips_186-3.pdf
<https://stackoverflow.com/questions/4523497/typedef-fixed-length-array/4523537>
<https://www.nist.gov/>
<https://flylib.com/books/en/2.878.1.133/1/>
https://oryx-embedded.com/doc/dsa_8c_source.html
<https://www.interviewcake.com/concept/java/bit-shift>

https://www.tutorialspoint.com/c_standard_library/c_function_ceil.htm
<https://emn178.github.io/online-tools/sha256.html>
<https://8gwifi.org/DSAF functionality?keysize=512>
https://en.wikipedia.org/wiki/Digital_Signature_Algorithm
https://en.wikipedia.org/wiki/Modular_exponentiation
<https://stackoverflow.com/questions/62597353/c-is-there-anyway-i-can-get-the-modulo-operator-to-work-on-non-integer-values>
<https://asecuritysite.com/encryption/random3?val=8>
<https://crypto.stackexchange.com/questions/11655/are-there-public-p-and-q-numbers-for-use-in-dsa>
<https://www.tutorialspoint.com/the-best-way-to-check-if-a-file-exists-using-standard-c-cplusplus>
https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
<https://www.educative.io/edpresso/c-reading-data-from-a-file-using-fread>
<https://github.com/littlestar/b64.c/blob/master/decode.c>
<https://stackoverflow.com/questions/342409/how-do-i-base64-encode-decode-in-c>
<https://www.base64decode.org/>
https://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_subr/base64.c
<https://www.youtube.com/watch?v=PQ8AruHaoLo>
<http://www.asciitable.com/>
https://en.wikipedia.org/wiki/Printf_format_string
[https://www.quora.com/What-is-the-difference-between-fwrite-and-fprintf-in-C#:~:text=What%20is%20the%20difference%20between%20a%20fprintf%20\(\)%20and%20a,other%20arrays%20output%20to%20files.](https://www.quora.com/What-is-the-difference-between-fwrite-and-fprintf-in-C#:~:text=What%20is%20the%20difference%20between%20a%20fprintf%20()%20and%20a,other%20arrays%20output%20to%20files.)
https://en.wikipedia.org/wiki/Convergent_series
https://en.wikipedia.org/wiki/Bitcoin_scalability_problem

<https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch08.html>
https://en.wikipedia.org/wiki/Satoshi_Nakamoto
<https://en.wikipedia.org/wiki/Peer-to-peer>
<https://docs.microsoft.com/en-us/windows/win32/winsock/complete-server-code>
<https://www.youtube.com/watch?v=ie-qROIQT4I>
<https://www.youtube.com/watch?v=kXyVqk3EbwE>
<https://www.youtube.com/watch?v=oHBI8k31fgM>
https://en.wikipedia.org/wiki/Hash_function#:~:text=A%20hash%20function%20is%20any,table%20called%20a%20hash%20table.
<https://en.wikipedia.org/wiki/SHA-2>
<https://stackoverflow.com/questions/2393863/set-an-exe-icon-for-my-program/26130514>
<http://clipart-library.com/cartoon-pictures-of-trees.html>
<https://hashtoolkit.com/reverse-sha256-hash/65e84be33532fb784c48129675f9eff3a682b27168c0ea744b2cf58ee02337c5>
<https://emn178.github.io/online-tools/sha256.html>
https://en.wikipedia.org/wiki/One-way_function
https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm
<https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp>
<https://positivemoney.org/how-money-works/advanced/how-do-banks-become-insolvent/#:~:text=For%20a%20bank%2C%20being%20insolvent,the%20availability%20of%20deposit%20insurance.>
<https://medium.com/@noogin/the-financial-crisis-and-history-of-bitcoin-27ebdb932b99#:~:text=Bitcoin%20was%20invented%20in%20the,the%20expense%20of%20their%20taxpayers.>
<https://www.statista.com/statistics/326707/bitcoin-price-index/>
<https://www.statista.com/statistics/273418/unadjusted-monthly-inflation-rate-in-the-us/>
<https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.coindesk.com%2Fbitcoin-halving-explainer&psig=AOvVaw1nb0W09i4DRVE1hpwIjUMP&ust=1623195250199000&source=images&cd=vfe&ved=0CA0QjhXqFwoTCJDEztbXhvECFQAAAAAdAAAAABAD>
<https://freecurrencyrates.com/en/exchange-rate-history/BTC-USD/2021>
<https://www.learncpp.com/cpp-tutorial/fixed-width-integers-and-size-t/>
https://en.wikipedia.org/wiki/Natural_number

ALREADY BACKED UP UP TO HERE