

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](mailto: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:** [yoooda@gmail.com](mailto:yoooda@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**](#_h4un56kmqh7v) **4**

[**Theoretical Background**](#_cxzhub6oj394) **4**

[**Cryptography**](#_lu5dvmlh4mm) **4**

[Hash Functions](#_84ygm98j54i3) 4

[SHA-256 - Secure Hash Algorithm](#_8tdlfjxflfao) 5

[Code](#_wh2nmb6jjpp0) 5

[SHA\_OPs.h + SHA\_OPs.c](#_6joqidtbnauq) 5

[SHA\_Rotate.h + SHA\_Rotate.c](#_816b8815ltty) 7

[SHA256.h + SHA256.c](#_yp8dl6djdfmz) 7

[DSA - Digital Signature Algorithm](#_6pu9b5os3sq4) 16

[Bitwise Math](#_dvvwghsdvtwd) 16

[Code](#_x0zr6des79rj) 16

[BNMath.h + BNMath.c](#_oggce513tn1f) 16

[**BlockChain**](#_mus1fec0b6wi) **51**

[Transactions](#_89bcl0r33ah8) 51

[Signing & Verification](#_g8sa03ub912o) 51

[Signing](#_liabxj2rluv2) 51

[Verification](#_qyyn0upv95c8) 51

[Blocks](#_sddeh993xdt6) 51

[Mining](#_mxe36q1kh4ok) 51

[Wallets](#_1w66bp5zcxf1) 51

[Base-64 Encoding](#_9eeil4soh3dz) 51

[Code](#_kshl4w8tb923) 51

[Base64.h + Base64.c](#_j5jo1ve6jzd6) 51

[**Works Cited**](#_okgl9r8qz865) **56**

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

# Introduction

# Theoretical Background

# Cryptography

## 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](#_mus1fec0b6wi)) 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 (strcpy(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://hashtoolkit.com/)).

Due to hash functions’ one-way functionality, deriving data from a hash is not possible. This feature will be exploited in [BlockChain Mining](#_mxe36q1kh4ok).

### SHA-256 - Secure Hash Algorithm

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](#_6pu9b5os3sq4) 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](#_mxe36q1kh4ok).

#### 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 missmatch!\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

### Bitwise Math

#### 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, -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, -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, -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, -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, -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  };    int\_t MAX(int\_t a, int\_t b) {  return (a > b) ? a : b;  }  int\_t MIN(int\_t a, int\_t b) {  return (a < b) ? a : b;  }  int\_t ceil\_div(float a, float q) {  float b = ceil(a/q);  int c = b / 1;  return c;  }    void BN\_Init(BN \*\* r) {  \*r = (BN\*)malloc(sizeof(BN));  BN \* rt = \*r;  rt->sign = 1;  rt->size = 0;  rt->data = NULL\_DATA;  }    void BN\_Init\_Stack(BN \* r) {  r->sign = 1;  r->size = 0;  r->data = NULL\_DATA;  }    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);    //Ajust 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 mod 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 mod 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 mod 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 mod P  for (i = 1; i < (1 << (d - 1)); i++)  {  BN\_Montgomery\_Mul(&s[i], &s[i - 1], r, k, p, &t);  }    //Let R = C mod 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 mod 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 mod 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 mod 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 mod 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) mod 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 mod 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;  } |

# BlockChain

## Transactions

### Signing & Verification

#### Signing

#### Verification

## 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"    const char BASE64\_CHAR\_SEQUENCE[64] = {  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',  'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',  'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',  'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',  'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',  'o', 'p', 'q', 'r', 's', 't', 'u', 'v',  'w', 'x', 'y', 'z', '0', '1', '2', '3',  '4', '5', '6', '7', '8', '9', '+', '/'  };    static const signed char BASE64\_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, 62, -1, -1, -1, 63,  52, 53, 54, 55, 56, 57, 58, 59, 60, 61, -1, -1, -1, -1, -1, -1,  -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,  15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -1, -1, -1, -1, -1,  -1, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,  41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, -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, -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, -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  };    size\_t B64\_Encoded\_Size(size\_t len) {  if (len % 3 != 0) {  len += 3 - (len % 3);  }  len /= 3;  len \*= 4;    return len;  }    size\_t B64\_Decoded\_Size(char \* base64\_array) {    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;  } |

# Works Cited

“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.

“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 - 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#:%7E:text=For%20ECDSA%2C%20the%20signature%20is,bytes%20for%20P%2D256. Accessed 27 May 2021.

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

“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.

“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.

“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.

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.

“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#:%7E:text=What%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#:%7E:text=A%20hash%20function%20is%20any,table%20called%20a%20hash%20table.

---. “Modular Exponentiation.” *Wikipedia*, 15 Jan. 2021, en.wikipedia.org/wiki/Modular\_exponentiation.

---. “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.0.0i433j0l2j46i175i199j0l4j46l2.881j0j7&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://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/DSAFunctionality?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/littlstar/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)

<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-qRQIQT4I>

<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>

ALLREADY BACKED UP UP TO HERE