Signature Scheme Implementation

Generated by Doxygen 1.14.0

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 code Struct Reference

code refers to the generator matrix of code 1, code 2 and the parity check matrix H_A's attributes. All the parameters are unsigned long integers.

```
#include <matrix.h>
```

Public Attributes

- unsigned long n
- unsigned long k
- unsigned long d

3.1.1 Detailed Description

code refers to the generator matrix of code 1, code 2 and the parity check matrix H_A's attributes. All the parameters are unsigned long integers.

3.1.2 Member Data Documentation

3.1.2.1 d

```
unsigned long code::d
```

minimum distance between 2 codewords

3.1.2.2 k

```
unsigned long code::k
```

length of the message

3.1.2.3 n

```
unsigned long code::n
```

length of the code

The documentation for this struct was generated from the following file:

- signature-scheme/include/matrix.h

3.2 Params Struct Reference

the derived datatype of the codes. it has the 3 parameters n, k, and d. n: length of the codeword, k: length of the message, d: minimum distance of the code.

```
#include <params.h>
```

Public Attributes

- uint32_t n
- uint32_t k
- uint32_t d

3.2.1 Detailed Description

the derived datatype of the codes. it has the 3 parameters n, k, and d. n: length of the codeword, k: length of the message, d: minimum distance of the code.

3.2.2 Member Data Documentation

3.2.2.1 d

```
uint32_t Params::d
```

Minimum distance of the code

3.2.2.2 k

```
uint32_t Params::k
```

Length of the message

3.2.2.3 n

```
uint32_t Params::n
```

Length of the code

The documentation for this struct was generated from the following file:

- signature-scheme/include/params.h

Chapter 4

File Documentation

4.1 signature-scheme/include/constants.h File Reference

It is the header file for the constants and flags used in the signature scheme.

```
#include <stdbool.h>
```

Macros

- #define MOD 2

  It is the modulus used in the signature scheme.
- #define PRINT true

  It is the flag to print the matrices and other information during the execution.
- #define SEED_SIZE 32

  It is the size of the seed used for random number generation in the signature scheme.
- #define PARAM_PATH "params.txt"

  It is the path to the parameters file.
- #define OUTPUT_DIR "output"

  It is the directory to the output file.
- #define OUTPUT_PATH OUTPUT_DIR "/output.txt"

  It is the path to the output file.
- #define CACHE_DIR "./matrix_cache/"

  It is the directory where the matrix cache is stored.
- #define MAX_FILENAME_LENGTH 256

  It is the maximum length of a filename in the signature scheme.

4.1.1 Detailed Description

It is the header file for the constants and flags used in the signature scheme.

4.1.2 Macro Definition Documentation

4.1.2.1 CACHE_DIR

```
#define CACHE_DIR "./matrix_cache/"
```

It is the directory where the matrix cache is stored.

This directory is used to store cached matrices used in the signature scheme. It is useful for storing matrices that are frequently used or generated, allowing for faster access and reducing the need to regenerate them each time the program runs. The cached matrices are in binary format and can be loaded or saved as needed.

4.1.2.2 MAX_FILENAME_LENGTH

```
#define MAX_FILENAME_LENGTH 256
```

It is the maximum length of a filename in the signature scheme.

4.1.2.3 MOD

```
#define MOD 2
```

It is the modulus used in the signature scheme.

4.1.2.4 OUTPUT_DIR

```
#define OUTPUT_DIR "output"
```

It is the directory to the output file.

4.1.2.5 OUTPUT_PATH

```
#define OUTPUT_PATH OUTPUT_DIR "/output.txt"
```

It is the path to the output file.

This file is used to store the output of the signature scheme, such as the generated signatures, debug information, and other relevant data.

4.1.2.6 PARAM_PATH

```
#define PARAM_PATH "params.txt"
```

It is the path to the parameters file.

This file contains the parameters for the signature scheme, such as the generator matrices and concatenated codes. The parameters are used to initialize the signature scheme and can be generated or read from this file.

4.1.2.7 PRINT

```
#define PRINT true
```

It is the flag to print the matrices and other information during the execution.

If set to true, the program will print debug information to the output file. If set to false, the program will not print debug information. This can be useful for debugging purposes or to reduce the output size in production runs.

4.1.2.8 SEED_SIZE

```
#define SEED_SIZE 32
```

It is the size of the seed used for random number generation in the signature scheme.

4.2 constants.h

Go to the documentation of this file.

```


#ifndef CONSTANTS_H
#define CONSTANTS_H

#include <stdbool.h>

#define MOD 2

#define PRINT true

#define SEED_SIZE 32

#define PARAM_PATH "params.txt"

#define OUTPUT_DIR "output"

#define OUTPUT_PATH OUTPUT_DIR "/output.txt"

#define CACHE_DIR "./matrix_cache/"

#define MAX_FILENAME_LENGTH 256

#endif /* CONSTANTS_H */
```

4.3 signature-scheme/include/keygen.h File Reference

It is the header file for the key generation module.

```
#include <flint/nmod_mat.h>
#include <stdbool.h>
#include "matrix.h"
```

Functions

- void create_generator_matrix_from_seed (slong n, slong k, slong d, nmod_mat_t gen_matrix, const unsigned char ∗seed, FILE ∗output_file)

  Creates a generator matrix from seed.
- void generate_parity_check_matrix_from_seed (slong n, slong k, slong d, nmod_mat_t H, const unsigned char ∗seed, FILE ∗output_file)

  Generates a parity check matrix from a seed.
- void get_or_generate_matrix_with_seed (const char ∗prefix, int n, int k, int d, nmod_mat_t matrix, void(∗generate_func)(slong, slong, slong, nmod_mat_t, FILE ∗), void(∗generate_from_seed_func)(slong, slong, slong, nmod_mat_t, const unsigned char ∗, FILE ∗), FILE ∗output_file, bool regenerate, bool use_↩ seed_mode, unsigned char ∗seed_out)

  Gets or generates matrix with seed.
- void generate_keys (struct code ∗C_A, struct code ∗C1, struct code ∗C2, nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2, bool use_seed_mode, bool regenerate, FILE ∗output_file, unsigned char ∗h_a_seed, unsigned char ∗g1_seed, unsigned char ∗g2_seed)

  Generates keys for a signature scheme.

4.3.1 Detailed Description

It is the header file for the key generation module.

4.3.2 Function Documentation

4.3.2.1 create_generator_matrix_from_seed()

```
void create_generator_matrix_from_seed (
            slong n,
            slong k,
            slong d,
            nmod_mat_t gen_matrix,
            const unsigned char * seed,
            FILE * output_file)
```

Creates a generator matrix from seed.

Creates a generator matrix from seed.

This function generates a generator matrix of size k x n using a deterministic approach based on a provided seed. The seed is used to generate random entries in the matrix, ensuring that the same seed will always produce the same matrix. The entries are generated modulo MOD. The process involves the following steps:

1. Stream Buffer Allocation: A buffer (stream) is allocated to hold the random bytes needed for the matrix entries. The size of this buffer is determined by the number of entries in the matrix (k ∗ n) multiplied by the size of each entry (4 bytes for a uint32_t).

2. Deterministic Random Generation: The randombytes_buf_deterministic function from the Sodium library is used to fill the buffer with random bytes based on the provided seed. This ensures that the same seed will always produce the same sequence of random bytes. It's important to note that this function uses the ChaCha20 algorithm under the hood for secure random number generation.

3. Matrix Entry Population: The function iterates over each entry in the generator matrix and fills it with values derived from the buffer. Each entry is constructed by combining 4 bytes from the buffer into a single uint32_t value, which is then reduced modulo MOD to fit within the required range.

4. Memory Cleanup: The allocated buffer is freed to avoid memory leaks.

Note

The function does not return any value; it directly modifies the provided gen_matrix object. It also requires a seed to ensure deterministic behavior, which is passed as an argument.

Chacha20:- It is a stream cipher that is designed to be fast and secure. It operates on 64-byte blocks and uses a 256-bit key, which is suitable for generating random numbers in this context. It takes streams of bytes, a key and nonce/count as input and produces a key stream as output. This is particularly useful for generating random matrices in cryptographic applications, ensuring that the same seed will always produce the same matrix. In this implementation, the seed is used as key ,stream as the input stream and the nonce is fixed(0). So evevrytime the function is called with the same seed, it will produce the same output stream.

Parameters

n	The total number of columns in the generator matrix. It represents the length of the codewords generated by the matrix.
k	The number of rows in the generator matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
gen_matrix	A pointer to an nmod_mat_t type, which represents the generator matrix to be created. The function initializes this matrix with random entries modulo MOD.
seed	A pointer to an unsigned char array that serves as the seed for the deterministic random number generation. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.
output_file	A pointer to a FILE object where any output or debug information can be written.

Returns

void This function does not return any value. It modifies the provided gen_matrix object directly and fills it with values derived from the seed.

4.3.2.2 generate_keys()

```
void generate_keys (
            struct code * C_A,
            struct code * C1,
            struct code * C2,
            nmod_mat_t H_A,
            nmod_mat_t G1,
            nmod_mat_t G2,
            bool use_seed_mode,
            bool regenerate,
            FILE * output_file,
            unsigned char * h_a_seed,
            unsigned char * g1_seed,
            unsigned char * g2_seed)
```

Generates keys for a signature scheme.

Generates keys for a signature scheme.

This function generates keys for a signature scheme by creating a parity check matrix (H_A) and two generator matrices (G1 and G2) based on the provided code parameters (C_A, C1, C2). It supports both random generation and seed-based generation of matrices. The generated matrices are saved to files, and if seed-based generation is used, the seeds are also saved.

Parameters

C_A	A pointer to a struct code object that contains the parameters for the first code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the parity check matrix H_A.
C1	A pointer to a struct code object that contains the parameters for the first generator code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the first generator matrix G1.
C2	A pointer to a struct code object that contains the parameters for the second generator code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the second generator matrix G2.
H_A	A pointer to an nmod_mat_t type, which represents the parity check matrix H_A to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
G1	A pointer to an nmod_mat_t type, which represents the first generator matrix G1 to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
G2	A pointer to an nmod_mat_t type, which represents the second generator matrix G2 to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
use_seed_mode	A boolean flag that indicates whether to use seed-based generation for the matrices. If set to true, the function will generate the matrices using a seed; if false, it will generate the matrices randomly.
regenerate	A boolean flag that indicates whether to regenerate the matrices even if they already exist. If set to true, the function will always generate new matrices; if false, it will load the existing matrices if available.
output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.
h_a_seed	A pointer to an unsigned char array where the generated seed for the parity check matrix H_A will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.
g1_seed	A pointer to an unsigned char array where the generated seed for the first generator matrix G1 will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.
g2_seed	A pointer to an unsigned char array where the generated seed for the second generator matrix G2 will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.

Returns

void This function does not return any value. It directly modifies the provided matrices (H_A, G1, G2) and saves them to files if necessary. If seed-based generation is used, it also saves the generated seeds to files.

4.3.2.3 generate_parity_check_matrix_from_seed()

```
void generate_parity_check_matrix_from_seed (
            slong n,
            slong k,
            slong d,
            nmod_mat_t H,
            const unsigned char * seed,
            FILE * output_file)
```

Generates a parity check matrix from a seed.

Generates a parity check matrix from a seed.

The function initializes a parity check matrix of size (n-k) x n using a deterministic approach based on a provided seed. The seed is used to generate random entries in the matrix, ensuring that the same seed will always produce the same matrix. The entries are generated modulo MOD. The process involves the following steps:

1. Stream Buffer Allocation: A buffer (stream) is allocated to hold the random bytes needed for the matrix entries. The size of this buffer is determined by the number of entries in the matrix ((n-k) ∗ n) multiplied by the size of each entry (4 bytes for a uint32_t).
2. Deterministic Random Generation: The randombytes_buf_deterministic function from the Sodium library is used to fill the buffer with random bytes based on the provided seed. This ensures that the same seed will always produce the same sequence of random bytes. It's important to note that this function uses the ChaCha20 algorithm under the hood for secure random number generation.
3. Matrix Entry Population: The function iterates over each entry in the parity check matrix and fills it with values derived from the buffer. Each entry is constructed by combining 4 bytes from the buffer into a single uint32_t value, which is then reduced modulo MOD to fit within the required range.
4. Memory Cleanup: The allocated buffer is freed to avoid memory leaks.

Note

The function does not return any value; it directly modifies the provided H object. It also requires a seed to ensure deterministic behavior, which is passed as an argument.

Parameters

n	The total number of columns in the parity check matrix. It represents the length of the codewords that the matrix checks for validity.
k	The number of rows in the parity check matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
H	A pointer to an nmod_mat_t type, which represents the parity check matrix to be created. The function initializes this matrix with random entries modulo MOD.
seed	A pointer to an unsigned char array that serves as the seed for the deterministic random number generation. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.
output_file	A pointer to a FILE object where any output or debug information can be written.

Returns

void This function does not return any value. It modifies the provided H object directly and fills it with values derived from the seed.

4.3.2.4 get_or_generate_matrix_with_seed()

```
void get_or_generate_matrix_with_seed (
             const char * prefix,
             int n,
             int k,
             int d,
             nmod_mat_t matrix,
             void(* generate_func )(slong, slong, slong, nmod_mat_t, FILE *),
             void(* generate_from_seed_func )(slong, slong, slong, nmod_mat_t, const unsigned
char *, FILE *),
             FILE * output_file,
             bool regenerate,
             bool use_seed_mode,
             unsigned char * seed_out)
```

Gets or generates matrix with seed.

Gets or generates matrix with seed.

This function checks if a matrix file exists for the given parameters (prefix, n, k, d). If it does and regeneration is not requested, it loads the matrix from the file. If the file does not exist or regeneration is requested, it generates a new matrix either using a random generation function or a seed-based generation function. The generated matrix is then saved to a file. This function is designed to handle both random and seed-based generation of matrices, allowing for reproducibility when using the same seed.

Parameters

prefix	A string that serves as a prefix for the filename of the matrix. This prefix is used to create a unique filename based on the parameters n, k, and d.
n	The total number of columns in the matrix. It represents the length of the codewords generated by the matrix.
k	The number of rows in the matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
matrix	A pointer to an nmod_mat_t type, which represents the matrix to be generated or loaded. The function initializes this matrix with random entries or loads it from a file if it already exists.
generate_func	A pointer to a function that generates a matrix with random entries. This function should take parameters (n, k, d, matrix, output_file) and fill the matrix with random values.
generate_from_seed_func	A pointer to a function that generates a matrix from a seed. This function should take parameters (n, k, d, matrix, seed, output_file) and fill the matrix with values derived from the seed.
output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.
regenerate	A boolean flag that indicates whether to regenerate the matrix even if it already exists. If set to true, the function will always generate a new matrix; if false, it will load the existing matrix if available.
use_seed_mode	A boolean flag that indicates whether to use seed-based generation for the matrix. If set to true, the function will generate the matrix using a seed; if false, it will generate the matrix randomly.
seed_out	A pointer to an unsigned char array where the generated seed will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.

Returns

void This function does not return any value. It modifies the provided matrix object directly and saves it to a file if necessary.

4.4 keygen.h

Go to the documentation of this file.

```
#ifndef KEYGEN_H
#define KEYGEN_H

#include <flint/nmod_mat.h>
#include <stdbool.h>
#include "matrix.h"

void create_generator_matrix_from_seed(slong n, slong k, slong d,
                                       nmod_mat_t gen_matrix,
                                       const unsigned char *seed,
                                       FILE *output_file);

void generate_parity_check_matrix_from_seed(slong n, slong k, slong d, nmod_mat_t H,
                                           const unsigned char *seed, FILE *output_file);

void get_or_generate_matrix_with_seed(const char* prefix, int n, int k, int d, nmod_mat_t matrix,
                                     void (*generate_func)(slong, slong, slong, nmod_mat_t, FILE*),
                                     void (*generate_from_seed_func)(slong, slong, slong, nmod_mat_t, const unsigned char*, FILE*),
                                     FILE* output_file, bool regenerate, bool use_seed_mode,
                                     unsigned char *seed_out);

void generate_keys(struct code* C_A, struct code* C1, struct code* C2,
                   nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2,
                   bool use_seed_mode, bool regenerate, FILE* output_file,
                   unsigned char* h_a_seed, unsigned char* g1_seed, unsigned char* g2_seed);

#endif
```

4.5 signature-scheme/include/matrix.h File Reference

Header filr for declaration of matrix operations and error-correcting code structures for signature scheme.

```
#include <stdio.h>
#include <flint/nmod_mat.h>
```

Classes

- struct code

  code refers to the generator matrix of code 1, code 2 and the parity check matrix H_A's attributes. All the parameters are unsigned long integers.

Functions

- void print_matrix (FILE ∗fp, nmod_mat_t matrix)

  Prints the contents of a matrix to a file stream.
- void transpose_matrix (int rows, int cols, int matrix[rows][cols], int transpose[cols][rows])

  Computes the transpose of a matrix.
- void multiply_matrices_gf2 (nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

  Performs matrix multiplication over GF(2).
- void make_systematic (unsigned long n, unsigned long k, nmod_mat_t H)

  Makes a matrix systematic form.
- void rref (int num_rows, int num_cols, int(∗H)[num_cols])

  Transforms a matrix into Reduced Row Echelon Form.

4.5.1 Detailed Description

Header filr for declaration of matrix operations and error-correcting code structures for signature scheme.

Header filr for declaration of matrix operations and error-correcting code structures for signature scheme.

4.5.2 Function Documentation

4.5.2.1 make_systematic()

```
void make_systematic (
            unsigned long n,
            unsigned long k,
            nmod_mat_t H)
```

Makes a matrix systematic form.

4.5.2.2 multiply_matrices_gf2()

```
void multiply_matrices_gf2 (
            nmod_mat_t C,
            const nmod_mat_t A,
            const nmod_mat_t B)
```

Performs matrix multiplication over GF(2).

Performs matrix multiplication over GF(2).

This function multiplies two matrices A and B, both defined over the finite field GF(2), and stores the result in matrix C. The multiplication is performed using bitwise operations, where addition is equivalent to XOR and multiplication is equivalent to AND. The function multiplies matrices over GF(2) using three nested loops. The outer loops iterate over rows of A and columns of B to compute each entry of the result matrix C. For each (i, j) entry, it initializes C[i][j] to 0, then uses the inner loop to XOR the bitwise AND of A[i][k] and B[k][j] into C[i][j]. This performs matrix multiplication using bitwise operations, with & as multiplication and ^ as addition in GF(2). The function assumes the matrices are properly initialized and dimensionally compatible.

Parameters

A	The first matrix to be multiplied, represented as an nmod_mat_t type from the FLINT library.
B	The second matrix to be multiplied, also represented as an nmod_mat_t type from the FLINT library.

Note

> The function assumes that the matrices A and B are compatible for multiplication, meaning the number of columns in A must equal the number of rows in B. It also assumes that the result matrix C has been initialized with appropriate dimensions to hold the product of A and B.

The function uses the nmod_mat_get_entry and nmod_mat_set_entry functions from the FLINT library to access and modify matrix entries. It iterates through each row of A and each column of B, computing the product for each entry in C.

Parameters

C	The result matrix where the product of A and B will be stored, also represented as an nmod_mat_t type from the FLINT library.

Returns

> void This function does not return a value; it modifies the matrix C in place to store the result of the multiplication.

4.5.2.3 print_matrix()

```
void print_matrix (
            FILE * fp,
            nmod_mat_t matrix)
```

Prints the contents of a matrix to a file stream.

Prints the contents of a matrix to a file stream.

It takes two parameters: a FILE ∗fp, which is the output stream, and an nmod_mat_t matrix, which represents the matrix to be printed. The function begins by printing the dimensions of the matrix in the format <rows x columns matrix>, using the r and c fields of nmod_mat_t matrix.

It then iterates through each row and column of the matrix, printing each entry in a formatted manner. Each row is enclosed in square brackets, and entries are separated by spaces. After printing all entries in a row, it moves to the next line for the next row.

Parameters

fp	The file pointer to which the matrix will be printed. This can be a file opened in write mode or stdout for console output.

Note

> The function assumes that the matrix is non-empty and that the nmod_mat_t structure is properly initialized. It does not handle any errors related to file operations or matrix initialization.

Parameters

matrix	The matrix to be printed, represented as an nmod_mat_t type from the FLINT library. This structure allows efficient access and manipulation of matrix data under modular arithmetic.

Returns

void This function does not return a value; it performs output operations directly to the specified file stream.

4.5.2.4 rref()

```
void rref (
            int num_rows,
            int num_cols,
            int(*) H[num_cols])
```

Transforms a matrix into Reduced Row Echelon Form.

Transforms a matrix into Reduced Row Echelon Form.

This function takes a binary matrix H, represented as a two-dimensional array of integers, and transforms it into its Reduced Row Echelon Form (RREF). The RREF is a form where each leading entry in a row is 1, and all entries in the column above and below each leading 1 are 0. The function performs forward and back substitution to achieve this form. It iterates through the columns of the matrix, finding non-zero elements to use as pivot points, and then eliminates other entries in the same column by XORing rows.

Parameters

num_rows	The number of rows in the matrix H, which is used to determine the range of rows that will be affected by the transformation.
num_cols	The number of columns in the matrix H, which is used to determine the range of columns that will be affected by the transformation.
H	The binary matrix that will be transformed into Reduced Row Echelon Form, represented as a two-dimensional array of integers. Each entry in the matrix is either 0 or 1, representing elements in GF(2).

Note

The notation (∗H)[num_cols] in the function parameter list means that H is a pointer to an array of num_↩ cols integers. In other words, H points to the first element of a 2D array where each row contains num_cols elements. This allows you to use H[i][j] inside the function to access the element at row i and column j, just like with a regular 2D array. The compiler needs to know the size of each row (num_cols) to correctly compute the memory offset for each element. This is why num_cols must be specified in the parameter type.

4.5.2.5 transpose_matrix()

```
void transpose_matrix (
             int rows,
             int cols,
             int matrix[rows][cols],
             int transpose[cols][rows])
```

Computes the transpose of a matrix.

Computes the transpose of a matrix.

This function takes a matrix defined by its number of rows and columns, and fills a new matrix with the transposed values. The transpose of a matrix is obtained by swapping its rows and columns, meaning that the element at position (i, j) in the original matrix becomes the element at position (j, i) in the transposed matrix. So that the elements of the firsr row in the original matrix becomes the elements at the first column in the transposed matrix.

Parameters

rows	The number of rows in the original matrix.
cols	The number of columns in the original matrix.

Note

> The function assumes that the input matrix is well-formed and that the transpose matrix has been allocated with appropriate dimensions.

Parameters

matrix	The original matrix to be transposed, represented as a two-dimensional array of integers.
transpose	The transposed matrix, which will be filled with the transposed values of the original matrix.

Returns

> void This function does not return a value; it modifies the transpose matrix in place.

4.6 matrix.h

Go to the documentation of this file.

```

#ifndef MATRIX_H
#define MATRIX_H

#include <stdio.h>
#include <flint/nmod_mat.h>

struct code {
    unsigned long n;
    unsigned long k;
    unsigned long d;
};

void print_matrix(FILE *fp, nmod_mat_t matrix);

void transpose_matrix(int rows, int cols, int matrix[rows][cols], int transpose[cols][rows]);

void multiply_matrices_gf2(nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B);

void make_systematic(unsigned long n, unsigned long k, nmod_mat_t H);

void rref(int num_rows, int num_cols, int (*H)[num_cols]);

#endif /* MATRIX_H */
```

4.7 signature-scheme/include/params.h File Reference

Header file for declaration of parameter handling functions and Params structure.

```
#include <stdint.h>
#include <stdbool.h>
#include "utils.h"
```

Classes

- struct Params

  the derived datatype of the codes. it has the 3 parameters n, k, and d. n: length of the codeword, k: length of the message, d: minimum distance of the code.

Functions

- void init_params (void)

  Initializes the libsodium library.
- bool get_yes_no_input (const char ∗prompt)

  Gets yes/no input from user.
- void get_user_input (Params ∗g1, Params ∗g2, Params ∗h)

  Get user input for parameters.
- uint32_t random_range (uint32_t min, uint32_t max)

  It generates a random number in the range [min, max].
- uint32_t get_H_A_n (void)

  Returns the n parameter of the concatenated code H_A.
- uint32_t get_H_A_k (void)

  Returns the k parameter of the concatenated code H_A.
- uint32_t get_H_A_d (void)

  Returns the d parameter of the concatenated code H_A.
- uint32_t get_G1_n (void)

  Returns the n parameter of the generator matrix G1.
- uint32_t get_G1_k (void)

  Returns the k parameter of the generator matrix G1.
- uint32_t get_G1_d (void)

  Returns the d parameter of the generator matrix G1.
- uint32_t get_G2_n (void)

  Returns the n parameter of the generator matrix G2.
- uint32_t get_G2_k (void)

  Returns the k parameter of the generator matrix G2.
- uint32_t get_G2_d (void)

  Returns the d parameter of the generator matrix G2.

4.7.1 Detailed Description

Header file for declaration of parameter handling functions and Params structure.

Header file for declaration of parameter handling functions and Params structure.

4.7.2 Function Documentation

4.7.2.1 get_G1_d()

```
uint32_t get_G1_d (
            void )
```

Returns the d parameter of the generator matrix G1.

Returns

The value of G1.d

4.7.2.2 get_G1_k()

```
uint32_t get_G1_k (
            void )
```

Returns the k parameter of the generator matrix G1.

Returns

The value of G1.k

4.7.2.3 get_G1_n()

```
uint32_t get_G1_n (
            void )
```

Returns the n parameter of the generator matrix G1.

Returns

The value of G1.n

4.7.2.4 get_G2_d()

```
uint32_t get_G2_d (
            void )
```

Returns the d parameter of the generator matrix G2.

Returns

The value of G2.d

4.7.2.5 get_G2_k()

```
uint32_t get_G2_k (
            void )
```

Returns the k parameter of the generator matrix G2.

Returns

The value of G2.k

4.7.2.6 get_G2_n()

```
uint32_t get_G2_n (
            void )
```

Returns the n parameter of the generator matrix G2.

Returns

The value of G2.n

4.7.2.7 get_H_A_d()

```
uint32_t get_H_A_d (
            void )
```

Returns the d parameter of the concatenated code H_A.

Returns

The value of H_A.d

4.7.2.8 get_H_A_k()

```
uint32_t get_H_A_k (
            void )
```

Returns the k parameter of the concatenated code H_A.

Returns

The value of H_A.k

4.7.2.9 get_H_A_n()

```
uint32_t get_H_A_n (
            void )
```

Returns the n parameter of the concatenated code H_A.

Returns

The value of H_A.n

4.7.2.10 get_user_input()

```
void get_user_input (
            Params * g1,
            Params * g2,
            Params * h)
```

Get user input for parameters.

This function checks if a saved parameter file exists and prompts the user to use it. If not, it asks the user whether they want to use BCH code or input G1 and G2 parameters manually. It generates random parameters if the user chooses not to input them. The function also saves the parameters to a file for future use. At first it checks if a saved parameter file exists. If it does, it prompts the user to use it. If the user chooses not to use the saved parameters, it asks whether they want to use BCH code or input G1 and G2 parameters manually.

• For the BCH code, it calculates the parameters based on user input for m and t, ensuring that the derived values for n, k, and d are consistent across G1 and G2. The parameters for H_A are derived from G1 and G2. m is the degree of the BCH code, and t is the error-correcting capability. The parameters are calculated as follows: • - n = 2^m - 1 (length of the codeword) • - k = m * t (length of the message) • - d = 2 * t + 1 (minimum distance of the code) • If the user chooses to input G1 and G2 parameters manually, it prompts for each parameter (n, k, d) and checks their validity. If the user does not want to input parameters, it generates random parameters for G1 and G2.

After gathering the parameters, it saves them to a file named Defined as PARAM_PATH for future use. The parameters for G1, G2, and H_A are printed to the console for confirmation.

Parameters

g1	Pointer to Params structure for G1 parameters.
g2	Pointer to Params structure for G2 parameters.
h	Pointer to Params structure for H_A parameters.

This function checks if a saved parameter file exists and prompts the user to use it. If not, it asks the user whether they want to use BCH code or input G1 and G2 parameters manually. It generates random parameters if the user chooses not to input them. The function also saves the parameters to a file for future use. At first it checks if a saved parameter file exists. If it does, it prompts the user to use it. If the user chooses not to use the saved parameters, it asks whether they want to use BCH code or input G1 and G2 parameters manually.

- For the BCH code, it calculates the parameters based on user input for `m` and `t`, ensuring that the derived values for `n`, `k`, and `d` are consistent across G1 and G2. The parameters for H_A are derived from G1 and G2. `m` is the degree of the BCH code, and `t` is the error-correcting capability. The parameters are calculated as follows:
- - n = 2^m - 1 (length of the codeword)
- - k = m * t (length of the message)
- - d = 2 * t + 1 (minimum distance of the code)
- If the user chooses to input G1 and G2 parameters manually, it prompts for each parameter (n, k, d) and checks their validity. If the user does not want to input parameters, it generates random parameters for G1 and G2.

After gathering the parameters, it saves them to a file named Defined as PARAM_PATH for future use. The parameters for G1, G2, and H_A are printed to the console for confirmation.

Parameters

g1	Pointer to Params structure for G1 parameters.
g2	Pointer to Params structure for G2 parameters.
h	Pointer to Params structure for H_A parameters.

4.7.2.11 get_yes_no_input()

```
bool get_yes_no_input (
            const char * prompt)
```

Gets yes/no input from user.

Gets yes/no input from user.

Parameters

prompt	It is a string that will be displayed to the user as a prompt.

This function prompts the user for a yes or no response. It reads the user's input and checks for the 1st character of it. It returns true for 'y' or 'Y', and false for 'n' or 'N'. If the input is invalid, it will terminate the program with failure.

Returns

true if the input's 1st character is 'y' or 'Y'.

false if the input's 1st character is 'n' or 'N'.

4.7.2.12 init_params()

```
void init_params (
            void )
```

Initializes the libsodium library.

This function should be called before using any other libsodium functions. The libsodium library is used for generating random numbers in this implementation.

Note

If libsodium fails to initialize, the program will exit with an error message.

See also

sodium_init

4.7.2.13 random_range()

```
uint32_t random_range (
            uint32_t min,
            uint32_t max)
```

It generates a random number in the range [min, max].

This function uses the libsodium library to generate a uniform random unsigned 32bit number. Basically, it generates a random number in the range [0, max-min] and adds with min.

Parameters

min	The minimum value of the range (inclusive).
max	The maximum value of the range (inclusive).

Returns

uint32_t The generated random number.

Note

This function assumes that max is greater than or equal to min.

See also

randombytes_uniform

This function uses the libsodium library to generate a uniform random unsigned 32bit number. Basically, it generates a random number in the range [0, max-min] and adds with min.

Parameters

min	The minimum value of the range (inclusive).
max	The maximum value of the range (inclusive).

Returns

uint32_t The generated random number.

Note

This function assumes that `max` is greater than or equal to `min`.

See also

randombytes_uniform

4.8 params.h

Go to the documentation of this file.

```
#ifndef PARAMS_H
#define PARAMS_H

#include <stdint.h>
#include <stdbool.h>
#include "utils.h"

typedef struct {
    uint32_t n;
    uint32_t k;
    uint32_t d;
} Params;

void init_params(void);

bool get_yes_no_input(const char *prompt);

void get_user_input(Params *g1, Params *g2, Params *h);

uint32_t random_range(uint32_t min, uint32_t max);

uint32_t get_H_A_n(void);

uint32_t get_H_A_k(void);

uint32_t get_H_A_d(void);

uint32_t get_G1_n(void);

uint32_t get_G1_k(void);

uint32_t get_G1_d(void);

uint32_t get_G2_n(void);

uint32_t get_G2_k(void);

uint32_t get_G2_d(void);

#endif
```

4.9 signature-scheme/include/signer.h File Reference

Header file for the signer module.

```
#include <flint/nmod_mat.h>
#include <stdio.h>
#include "matrix.h"
```

Functions

- void generate_signature (nmod_mat_t bin_hash, const unsigned char *message, size_t message_len, struct code C_A, struct code C1, struct code C2, nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2, nmod_mat_t F, nmod_mat_t signature, FILE *output_file)

  Generates a signature based on the provided parameters.

4.9.1 Detailed Description

Header file for the signer module.

4.9.2 Function Documentation

4.9.2.1 generate_signature()

```
void generate_signature (
            nmod_mat_t bin_hash,
            const unsigned char * message,
            size_t message_len,
            struct code C_A,
            struct code C1,
            struct code C2,
            nmod_mat_t H_A,
            nmod_mat_t G1,
            nmod_mat_t G2,
            nmod_mat_t F,
            nmod_mat_t signature,
            FILE * output_file)
```

Generates a signature based on the provided parameters.

Generates a signature based on the provided parameters.

This function generates a digital signature for a given message using a code-based cryptographic approach. The signature is created by constructing a hybrid generator matrix from two codes (C1 and C2), salting the message, hashing it using SHA-256, and encoding the resulting binary hash vector as a codeword. The process ensures that the signature meets the minimum weight requirement specified by code C_A. It uses matrix operations over finite fields via the FLINT library (`nmod_mat_t`) and cryptographic hashing and randomness via Libsodium.

The function performs the following steps:

1. Allocate an array J to hold a random selection of indices from [0, C_A.n - 1].
2. Generate a random permutation of size C1.n and store it in J.
3. Initialize the matrix G_star (size C1.k × C_A.n) which will hold the combined generator matrix.
4. For each column index i from 0 to C_A.n - 1: a. If i is in J (i.e., selected for G1), copy the corresponding column from G1 to G_star. b. Otherwise, copy the next available column from G2 to G_star.
5. Free the random index array J as it's no longer needed.
6. If PRINT is enabled, print the contents of G_star to the output file.
7. Transpose G_star to obtain G_star_T.
8. Compute F = H_A × G_star_T. This may be used for constraint checking or debugging.
9. Begin a loop to compute a valid signature: a. Allocate and fill a buffer with the original message followed by a random salt of the same length. b. Hash the salted message using SHA-256. c. Convert the resulting hash into a binary vector (0s and 1s) to fill the bin_hash matrix. d. Multiply bin_hash with G_star to produce the signature matrix. e. If the weight (Hamming weight) of the signature is less than C_A.d (minimum distance), repeat the loop.
10. If PRINT is enabled, print the binary hash matrix to the output file.
11. Clear memory used by G_star and G_star_T.

    Note

    This function assumes that the input matrices and codes are properly initialized and that the MOD constant is defined for finite field operations.

Parameters

bin_hash	It is a matrix that will hold the binary hash of the message.
message	It is the input message for which the signature is being generated.
message_len	It is the length of the input message in bytes.
C_A	It is the derived code that defines the parameters for the signature generation, including the length of the code (n), the length of the message (k), and the minimum distance (d).

Parameters

C1	It is the first code used in the signature generation process, which provides part of the generator matrix.
C2	It is the second code used in the signature generation process, which provides the remaining part of the generator matrix.
H_A	It is the parity-check matrix for the derived code C_A, which is used to ensure that the generated signature meets the required properties.
G1	It is the generator matrix for the first code C1, which is used to construct part of the hybrid generator matrix.
G2	It is the generator matrix for the second code C2, which is used to construct the remaining part of the hybrid generator matrix.
F	It is a matrix that will hold the product of the parity-check matrix H_A and the transposed hybrid generator matrix G_star_T. This is used for debugging or verification purposes.
signature	It is the output matrix that will hold the generated signature for the input message.
output_file	It is a file pointer to the output file where debug information will be printed if the PRINT flag is set.

Note

The function uses the Sodium library for cryptographic operations and the FLINT library for matrix operations.

4.10 signer.h

Go to the documentation of this file.

```


#ifndef SIGNER_H
#define SIGNER_H

#include <flint/nmod_mat.h>
#include <stdio.h>
#include "matrix.h"

void generate_signature(nmod_mat_t bin_hash, const unsigned char* message, size_t message_len,
                    struct code C_A, struct code C1, struct code C2,
                    nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2,
                    nmod_mat_t F, nmod_mat_t signature, FILE* output_file);

#endif
```

4.11 signature-scheme/include/utils.h File Reference

Header file for utility functions used in the signature scheme.

```
#include <math.h>
#include <stdlib.h>
#include <sodium.h>
#include <stdbool.h>
#include <flint/flint.h>
#include <flint/nmod_mat.h>
#include "matrix.h"
```

Functions

- long weight (nmod_mat_t array)

  Calculates the Hamming weight of a matrix row.
- double binary_entropy (double p)

  Calculates the binary entropy of a probability.
- void generate_random_set (unsigned long upper_bound, unsigned long size, unsigned long set[size])

  Generates a random set of unique integers within a specified range.
- char ∗ generate_matrix_filename (const char ∗prefix, int n, int k, int d)

  Generates a filename for a matrix based on prefix and dimensions.
- void save_matrix (const char ∗filename, const nmod_mat_t matrix)

  Saves a matrix to a text file in FLINT format.
- int load_matrix (const char ∗filename, nmod_mat_t matrix)

  Loads a matrix from a text file in FLINT format.
- int file_exists (const char ∗filename)

  Checks if a file exists.
- char ∗ generate_seed_filename (const char ∗prefix, int n, int k, int d)

  Generates a filename for a seed based on prefix and parameters.
- bool save_seed (const char ∗filename, const unsigned char ∗seed)

  Saves a seed to a binary file.
- bool load_seed (const char ∗filename, unsigned char ∗seed)

  Loads a seed from a binary file.
- char ∗ read_file (const char ∗filename)

  Reads the contents of a file into a string.
- char ∗ read_file_or_generate (const char ∗filename, int msg_len)

  Reads a message from file or generates random message.
- bool load_params (struct code ∗C_A, struct code ∗C1, struct code ∗C2)

  Loads parameters for codes from a file.
- void ensure_matrix_cache ()

  Ensures matrix cache directory exists.
- void ensure_output_directory ()

  Ensures output directory exists.
- char ∗ normalize_message_length (const char ∗msg, size_t msg_len, size_t target_len, size_t ∗final_len_out)

  Normalizes message length by padding or truncating.

4.11.1 Detailed Description

Header file for utility functions used in the signature scheme.

4.11.2 Function Documentation

4.11.2.1 binary_entropy()

```
double binary_entropy (
            double p)
```

Calculates the binary entropy of a probability.

Calculates the binary entropy of a probability.

The binary entropy function computes the entropy of a binary random variable with probability p of being 1. It uses the formula: H(p) = -p ∗ log2(p) - (1 - p) ∗ log2(1 - p). The function checks if p is within the valid range (0, 1) and returns 0 if p is 0 or 1, as there is no uncertainty in those cases. The logarithm is computed using the log2 function from the math library, which calculates the base-2 logarithm.

Parameters

p	A double representing the probability of a binary event occurring, where $0 < p < 1$.

Returns

> double The binary entropy of the given probability p, which is a measure of uncertainty in bits.

Note

> The function assumes that the input probability p is a valid value between 0 and 1 (exclusive). If p is outside this range, the function will return 0, indicating no uncertainty. It does not handle cases where p is NaN or infinite.

4.11.2.2 ensure_matrix_cache()

```
void ensure_matrix_cache ()
```

Ensures matrix cache directory exists.

Ensures matrix cache directory exists.

This function checks if the directory "matrix_cache" exists. If it does not, it creates the directory with permissions set to 0700 (read, write, and execute permissions for the owner only). This is useful for storing cached matrices used in the signature scheme. It uses the stat function to check for the directory's existence and mkdir to create it if necessary. The function does not return any value; it simply ensures that the directory is present before any matrix operations are performed.

Returns

> It does not return any value; it simply ensures that the matrix cache directory is present before any matrix operations are performed.

4.11.2.3 ensure_output_directory()

```
void ensure_output_directory ()
```

Ensures output directory exists.

Ensures output directory exists.

This function checks if the directory "output" exists. If it does not, it creates the directory with permissions set to 0700 (read, write, and execute permissions for the owner only). This is useful for storing output files generated by the signature scheme. It uses the stat function to check for the directory's existence and mkdir to create it if necessary. The function does not return any value; it simply ensures that the output directory is present before any output operations are performed.

Returns

> It does not return any value; it simply ensures that the output directory is present before any output operations are performed.

4.11.2.4 file_exists()

```
int file_exists (
            const char * filename)
```

Checks if a file exists.

Checks if a file exists.

It takes a filename as input and tries to open the file using fopen with the "r" mode, which is for reading. If the file cannot be opened (for example, if it does not exist), fopen returns NULL. In this case, the function returns 0 to indicate that the file does not exist. If the file is successfully opened, it is immediately closed using fclose, and the function returns 1 to indicate that the file exists.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to check for existence. The function will attempt to open this file in read mode.

Returns

int 1 if the file exists (i.e., it can be opened in read mode), or 0 if the file does not exist (i.e., it cannot be opened).

Note

The function does not perform any additional checks on the file, such as verifying its contents or permissions. It simply checks for the existence of the file by trying to open it. If the file is successfully opened, it is closed immediately after checking.

4.11.2.5 generate_matrix_filename()

```
char * generate_matrix_filename (
            const char * prefix,
            int n,
            int k,
            int d)
```

Generates a filename for a matrix based on prefix and dimensions.

Generates a filename for a matrix based on prefix and dimensions.

The function constructs a filename for a matrix by concatenating a predefined cache directory with a prefix and the dimensions of the matrix (n, k, d). The resulting filename is formatted as "cache_dir/prefix_n_k_d.txt", where cache_dir is defined as "matrix_cache/", and prefix, n, k, and d are provided as parameters. The function allocates memory for the filename string, formats it using sprintf, and returns the pointer to the generated filename.

Parameters

prefix	A pointer to a constant character string that serves as a prefix for the filename. This prefix is typically used to identify the type of matrix or its specific characteristics. Example prefixes could be "H_A", "G1", or "G2", depending on the context of the matrix being generated or stored.
n	The length of the code.
k	The dimension of the code.
d	The minimum distance of the code.

Returns

char∗ A pointer to a dynamically allocated string containing the generated filename. If memory allocation fails, it returns NULL.

Note

The function allocates memory for the filename string, so it is the caller's responsibility to free this memory when it is no longer needed. The maximum length of the generated filename is defined by MAX_FILENAME←_LENGTH, which should be set appropriately to accommodate the longest expected filename.

4.11.2.6 generate_random_set()

```
void generate_random_set (
            unsigned long upper_bound,
            unsigned long size,
            unsigned long set[size])
```

Generates a random set of unique integers within a specified range.

Generates a random set of unique integers within a specified range.

The function generates a random set of unique integers from 0 to upper_bound - 1, ensuring that the size of the set is equal to size. It uses the modern Fisher-Yates shuffle algorithm to randomly permute an array of integers from 0 to upper_bound - 1, and then selects the first size elements from this shuffled array. The resulting set is sorted in ascending order using the qsort function with a custom comparison function.

Parameters

upper_bound	An unsigned long integer representing the upper limit of the range from which unique integers will be selected. The function will generate integers in the range [0, upper_bound).
size	An unsigned long integer representing the number of unique integers to be generated in the set. The function will ensure that the size of the generated set is equal to this value.
set	A pointer to an array of unsigned long integers where the generated unique integers will be stored. The size of this array should be at least size elements to hold the generated set.

Note

Algorithm:

1. Initialize an array arr containing all integers from 0 to upper_bound - 1.
2. Shuffle the array in-place using the modern Fisher-Yates shuffle: ○ Iterate from the last element to the second element. ○ In each iteration, generate a random index j such that 0 <= j <= i. ○ Swap the elements at indices i and j.
3. Copy the first size elements from the shuffled array into set.
4. Sort the set array in ascending order.

4.11.2.7 generate_seed_filename()

```
char * generate_seed_filename (
            const char * prefix,
            int n,
            int k,
            int d)
```

Generates a filename for a seed based on prefix and parameters.

Generates a filename for a seed based on prefix and parameters.

The function constructs a filename for a seed by concatenating a predefined cache directory with a prefix and the parameters n, k, and d. The resulting filename is formatted as "cache_dir/prefix_n_k_d_seed.bin", where cache_dir is defined as "matrix_cache/", and prefix, n, k, and d are provided as parameters. The function allocates memory for the filename string, formats it using snprintf, and returns the pointer to the generated filename.

Parameters

prefix	A pointer to a constant character string that serves as a prefix for the filename. This prefix is typically used to identify the type of seed or its specific characteristics, such as "H_A", "G1", or "G2", depending on the context of the seed being generated or stored.
n	An integer representing the length of the code. This value is used to uniquely identify the seed associated with a specific code length.
k	An integer representing the dimension of the code. This value is used to uniquely identify the seed associated with a specific code dimension.
d	An integer representing the minimum distance of the code. This value is used to uniquely identify the seed associated with a specific code minimum distance.

Returns

char* A pointer to a dynamically allocated string containing the generated filename. If memory allocation fails, it returns NULL. The caller is responsible for freeing this memory when it is no longer needed.

4.11.2.8 load_matrix()

```
int load_matrix (
            const char * filename,
            nmod_mat_t matrix)
```

Loads a matrix from a text file in FLINT format.

Loads a matrix from a text file in FLINT format.

It opens a file with the specified filename for reading. If it fails to open the file, it returns 0. It then reads the dimensions of the matrix (number of rows and columns) from the file. If it fails to read these dimensions, it closes the file and returns 0. The function clears any existing data in the provided matrix, initializes it with the specified dimensions, and then reads each entry of the matrix from the file, setting the corresponding entry in the matrix using nmod_mat_set_entry. If it fails to read any value, it closes the file and returns 0. Finally, it closes the file and returns 1 to indicate success.

Parameters

filename	A pointer to a constant character string that specifies the name of the file from which the matrix will be loaded.
matrix	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library.

Returns

int 1 if the matrix was successfully loaded from the file, or 0 if there was an error (e.g., file not found, failed to read dimensions or values).

Note

The file should be in a specific text format that includes the dimensions of the matrix followed by its entries.

4.11.2.9 load_params()

```
bool load_params (
            struct code * C_A,
            struct code * C1,
            struct code * C2)
```

Loads parameters for codes from a file.

Loads parameters for codes from a file.

It opens a file named "params.txt" in read mode and reads key-value pairs from it. The keys correspond to parameters of the codes, such as "H_A_n", "H_A_k", "H_A_d", "G1_n", "G1_k", "G1_d", "G2_n", "G2_k", and "G2_d". For each key, it assigns the corresponding value to the appropriate field in the provided code structures (C_A, C1, and C2). If the file cannot be opened, it prints an error message and returns false. If all parameters are successfully loaded, it returns true.

Parameters

C↩ _A	A pointer to the concatenated code structure, which contains parameters for the concatenated code (C_A).
C1	A pointer to the first generator code structure, which contains parameters for the first code (C1).
C2	A pointer to the second generator code structure, which contains parameters for the second code (C2).

Returns

true If the parameters were successfully loaded from the file, meaning that the file was opened, all key-value pairs were read, and the corresponding fields in the code structures were set.

false If there was an error opening the file or if any key-value pair could not be read, indicating that the parameters were not loaded successfully.

Note

The function assumes that the file "params.txt" exists and is formatted correctly with key-value pairs. If any key is missing or if the file cannot be read, the function will not set the corresponding fields in the code structures.

4.11.2.10 load_seed()

```
bool load_seed (
            const char * filename,
            unsigned char * seed)
```

Loads a seed from a binary file.

The function takes a filename and a pointer to an unsigned char array (seed) as input. It opens the specified file in binary read mode ("rb"). If the file cannot be opened, it returns false. It then reads SEED_SIZE bytes from the file into the seed array using fread. After reading, it closes the file and checks if the number of bytes read matches SEED_SIZE. If they match, it returns true, indicating that the seed was successfully loaded; otherwise, it returns false.

Parameters

filename	A pointer to a constant character string that specifies the name of the file from which the seed will be loaded. The file should contain binary data representing the seed.
seed	A pointer to an unsigned char array where the loaded seed data will be stored. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.

Returns

true If the seed was successfully loaded from the file, meaning that the file was opened, the seed data was read, and the correct number of bytes was read.

false If there was an error opening the file or if the number of bytes read does not match SEED_SIZE, indicating that the seed was not loaded successfully.

4.11.2.11 normalize_message_length()

```
char * normalize_message_length (
            const char * msg,
            size_t msg_len,
            size_t target_len,
            size_t * final_len_out)
```

Normalizes message length by padding or truncating.

Normalizes message length by padding or truncating.

The function takes a message, its length, a target length, and an optional pointer to store the final length. It allocates memory for a new message of the target length. If the original message is shorter than the target length, it copies the original message and fills the remaining space with random uppercase letters (A-Z). If the original message is longer than the target length, it truncates it to fit. If the lengths match, it simply copies the original message. The function returns the newly created message and updates the final length if requested.

Parameters

msg	A pointer to a constant character string that represents the original message to be normalized. The message can be of any length, and the function will either pad it with random characters or truncate it to fit the target length.
msg_len	The length of the original message in bytes. This value is used to determine whether the message needs to be padded or truncated to match the target length.
target_len	The desired length of the normalized message. The function will ensure that the final message has this exact length by either padding it with random characters or truncating it if necessary.
final_len_out	A pointer to a size_t variable where the final length of the normalized message will be stored. This parameter is optional; if it is NULL, the function will not update the final length.

Returns

char∗ A pointer to a dynamically allocated string containing the normalized message.

Note

The function allocates memory for the new message, so it is the caller's responsibility to free this memory when it is no longer needed. If memory allocation fails, the function will print an error message and return NULL.

4.11.2.12 read_file()

```
char * read_file (
            const char * filename)
```

Reads the contents of a file into a string.

Reads the contents of a file into a string.

It opens the specified file in read mode, checks if the file was opened successfully, and then reads its contents into a buffer. The function first seeks to the end of the file to determine its length, rewinds to the beginning, allocates memory for the buffer, and reads the file's contents into it. Finally, it closes the file and returns the buffer containing the file's contents as a null-terminated string. If any step fails (e.g., file not found, memory allocation failure), it prints an error message and returns NULL.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to be read. The function will attempt to open this file in read mode and read its contents.

Returns

char∗ A pointer to a dynamically allocated string containing the contents of the file. If the file cannot be opened or is empty, it returns NULL.

Note

The caller is responsible for freeing the returned buffer after use to avoid memory leaks.

4.11.2.13 read_file_or_generate()

```
char * read_file_or_generate (
            const char * filename,
            int msg_len)
```

Reads a message from file or generates random message.

Reads a message from file or generates random message.

It attempts to open the specified file in read mode. If the file is successfully opened, it checks its length. If the length is zero or less, it generates a random message of a specified length and saves it to the file. If the file contains valid data, it reads the contents into a dynamically allocated string and returns it. If the file cannot be opened, it generates a random message, saves it to the file, and returns that message.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to read the message from. If the file does not exist or is empty, a random message will be generated and saved to this file.
msg_len	An integer representing the length of the message to be generated if the file is empty or does not exist. The function will generate a random message of this length using uppercase letters (A-Z).

Returns

> char∗ A pointer to a dynamically allocated string containing the message read from the file or the generated random message. If memory allocation fails or if there is an error reading the file, it returns NULL. The caller is responsible for freeing the returned string after use.

4.11.2.14 save_matrix()

```
void save_matrix (
            const char * filename,
            const nmod_mat_t matrix)
```

Saves a matrix to a text file in FLINT format.

Saves a matrix to a text file in FLINT format.

The function opens a file with the specified filename for writing. If it fails to open the file, it prints an error message and returns. It then retrieves the number of rows and columns of the matrix using nmod_mat_nrows and nmod↩_mat_ncols, respectively, and writes these dimensions to the file. After that, it iterates through each entry of the matrix, retrieves its value using nmod_mat_entry, and writes it to the file in a space-separated format. Finally, it closes the file.

Parameters

filename	A pointer to a constant character string that specifies the name of the file where the matrix will be saved. The file will be created if it does not exist, or overwritten if it does.
matrix	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library. The matrix should be initialized and populated with values before calling this function. The function will save the matrix in a specific text format that includes its dimensions followed by its entries.

Note

> The function does not perform any error checking on the matrix itself, such as ensuring it is initialized or has valid dimensions. It assumes that the matrix is properly set up before calling this function. The file will be created in the current working directory, and if the file already exists, it will be overwritten.

4.11.2.15 save_seed()

```
bool save_seed (
            const char * filename,
            const unsigned char * seed)
```

Saves a seed to a binary file.

The function takes a filename and a pointer to an unsigned char array (seed) as input. It opens the specified file in binary write mode ("wb"). If the file cannot be opened, it returns false. It then writes the seed data to the file using fwrite, which writes SEED_SIZE bytes from the seed array to the file. After writing, it closes the file and checks if the number of bytes written matches SEED_SIZE. If they match, it returns true, indicating that the seed was successfully saved; otherwise, it returns false.

Parameters

filename	A pointer to a constant character string that specifies the name of the file where the seed will be saved. The file will be created if it does not exist, or overwritten if it does.
seed	A pointer to an unsigned char array that contains the seed data to be saved. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.

Returns

true If the seed was successfully saved to the file, meaning that the file was opened, the seed data was written, and the correct number of bytes was written.

false If there was an error opening the file or if the number of bytes written does not match SEED_SIZE, indicating that the seed was not saved successfully.

4.11.2.16 weight()

```
long weight (
             nmod_mat_t array)
```

Calculates the Hamming weight of a matrix row.

Calculates the Hamming weight of a matrix row.

The Hamming weight is the number of non-zero elements in a row of a matrix. This function iterates through the first row of the provided matrix and counts how many entries are equal to 1, which corresponds to the Hamming weight. It assumes that the matrix is represented as an nmod_mat_t type from the FLINT library, which allows for efficient access to matrix entries.

Parameters

array	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library.

Returns

long The Hamming weight of the first row of the matrix, which is the count of entries equal to 1.

Note

The function assumes that the matrix has at least one row and that the entries are in the range of 0 to MOD-1, where MOD is defined in the FLINT library. It does not handle cases where the matrix is empty or has no rows.

4.12 utils.h

Go to the documentation of this file.

```


#ifndef UTILS_H
#define UTILS_H

#include <math.h>
#include <stdlib.h>
#include <sodium.h>
#include <stdbool.h>
#include <flint/flint.h>
#include <flint/nmod_mat.h>
#include "matrix.h"

long weight(nmod_mat_t array);

double binary_entropy(double p);

void generate_random_set(unsigned long upper_bound, unsigned long size, unsigned long set[size]);

char* generate_matrix_filename(const char* prefix, int n, int k, int d);

void save_matrix(const char* filename, const nmod_mat_t matrix);

int load_matrix(const char* filename, nmod_mat_t matrix);

int file_exists(const char* filename);

char* generate_seed_filename(const char* prefix, int n, int k, int d);

bool save_seed(const char* filename, const unsigned char *seed);

bool load_seed(const char* filename, unsigned char *seed);

char *read_file(const char *filename);

char *read_file_or_generate(const char *filename, int msg_len);

bool load_params(struct code *C_A, struct code *C1, struct code *C2);

void ensure_matrix_cache();

void ensure_output_directory();

char *normalize_message_length(const char *msg, size_t msg_len, size_t target_len, size_t
*final_len_out);

#endif
```

4.13 signature-scheme/include/verifier.h File Reference

It is the header file for the verifier module.

```
#include <flint/nmod_mat.h>
#include <stdio.h>
#include "matrix.h"
```

Functions

- void verify_signature (nmod_mat_t bin_hash, size_t message_len, unsigned long sig_len, nmod_mat_t signature, nmod_mat_t F, struct code C_A, nmod_mat_t H_A, FILE *output_file)

  Verifies a digital signature.

4.13.1 Detailed Description

It is the header file for the verifier module.

4.13.2 Function Documentation

4.13.2.1 verify_signature()

```
void verify_signature (
            nmod_mat_t bin_hash,
            size_t message_len,
            unsigned long sig_len,
            nmod_mat_t signature,
            nmod_mat_t F,
            struct code C_A,
            nmod_mat_t H_A,
            FILE * output_file)
```

Verifies a digital signature.

This function verifies a digital signature by checking if the product of the binary hash of the message and the generator matrix equals the product of the parity-check matrix and the transposed signature. It uses the FLINT library for matrix operations and prints debug information.

Parameters

bin_hash	Binary hash matrix for the message
message_len	Length of the message
sig_len	Length of the signature
signature	Signature matrix to verify
F	Combined matrix F = H_A ∗ G∗^T
C_A	Code parameters for the concatenated code
H_A	Parity check matrix for the concatenated code
output_file	File to write verification output to

4.14 verifier.h

Go to the documentation of this file.

```


#ifndef VERIFIER_H
#define VERIFIER_H

#include <flint/nmod_mat.h>
#include <stdio.h>
#include "matrix.h"

void verify_signature(nmod_mat_t bin_hash, size_t message_len,
                      unsigned long sig_len, nmod_mat_t signature,
                      nmod_mat_t F, struct code C_A,
                      nmod_mat_t H_A, FILE *output_file);

#endif
```

4.15 signature-scheme/src/keygen.c File Reference

Implementation of key generation functions for the signature scheme.

```
#include <string.h>
#include <sodium.h>
#include "keygen.h"
#include "matrix.h"
#include "utils.h"
#include "params.h"
#include "constants.h"
```

Functions

- void generate_random_seed (unsigned char ∗seed)

  This function generates a random seed of a fixed size.
- void create_generator_matrix (slong n, slong k, slong d, nmod_mat_t gen_matrix, FILE ∗output_file)

  Creates a generator matrix object.
- void generate_parity_check_matrix (slong n, slong k, slong d, nmod_mat_t H, FILE ∗output_file)

  This function generates a parity check matrix with random entries modulo MOD.
- void create_generator_matrix_from_seed (slong n, slong k, slong d, nmod_mat_t gen_matrix, const unsigned char ∗seed, FILE ∗output_file)

  Creates a generator matrix from seed object.
- void generate_parity_check_matrix_from_seed (slong n, slong k, slong d, nmod_mat_t H, const unsigned char ∗seed, FILE ∗output_file)

  This function generates a parity check matrix from a seed.
- void get_or_generate_matrix_with_seed (const char ∗prefix, int n, int k, int d, nmod_mat_t matrix, void(∗generate_func)(slong, slong, slong, nmod_mat_t, FILE ∗), void(∗generate_from_seed_func)(slong, slong, slong, nmod_mat_t, const unsigned char ∗, FILE ∗), FILE ∗output_file, bool regenerate, bool use_↩ seed_mode, unsigned char ∗seed_out)

  Gets or generates matrix with seed object.
- void generate_keys (struct code ∗C_A, struct code ∗C1, struct code ∗C2, nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2, bool use_seed_mode, bool regenerate, FILE ∗output_file, unsigned char ∗h_a_seed, unsigned char ∗g1_seed, unsigned char ∗g2_seed)

  A function that generates keys for a signature scheme based on the provided parameters and options.

4.15.1 Detailed Description

Implementation of key generation functions for the signature scheme.

This file implements the key generation process for the post-quantum signature scheme, including functions for generating random seeds, creating generator matrices, generating parity check matrices, and producing public/private key pairs.

4.15.2 Function Documentation

4.15.2.1 create_generator_matrix()

```
void create_generator_matrix (
            slong n,
            slong k,
            slong d,
            nmod_mat_t gen_matrix,
            FILE * output_file)
```

Creates a generator matrix object.

This function initializes a generator matrix of size k x n with random entries modulo MOD. It uses the FLINT library's nmod_mat_t type to represent the matrix and fills it with random values using the nmod_mat_randtest function. The matrix is initialized with dimensions k (number of rows) and n (number of columns), and the entries are generated randomly in the range of 0 to MOD-1. The process involves the following steps:

1. Random State Initialization: A FLINT random state (flint_rand_t) is initialized using flint_randinit. This state is required for generating random numbers in FLINT.
2. Matrix Initialization: The generator matrix (gen_matrix) is initialized with k rows and n columns, and modulus MOD, using nmod_mat_init.
3. Random Entry Generation: The matrix is filled with random values using nmod_mat_randtest, which uses the previously initialized random state to populate each entry.
4. Random State Cleanup: The random state is cleared with flint_randclear to free any resources associated with it.

Note

The function does not return any value; it directly modifies the provided gen_matrix object. It also initializes a random state using flint_rand_t to ensure that the random values are generated correctly.

Parameters

n	The total number of columns in the generator matrix. It represents the length of the codewords generated by the matrix.
k	The number of rows in the generator matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
gen_matrix	A pointer to an nmod_mat_t type, which represents the generator matrix to be created. The function initializes this matrix with random entries modulo MOD.
output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.

Returns

void This function does not return any value. It modifies the provided gen_matrix object directly and fills it with random values.

4.15.2.2 create_generator_matrix_from_seed()

```
void create_generator_matrix_from_seed (
            slong n,
            slong k,
            slong d,
            nmod_mat_t gen_matrix,
            const unsigned char * seed,
            FILE * output_file)
```

Creates a generator matrix from seed object.

Creates a generator matrix from seed.

This function generates a generator matrix of size k x n using a deterministic approach based on a provided seed. The seed is used to generate random entries in the matrix, ensuring that the same seed will always produce the same matrix. The entries are generated modulo MOD. The process involves the following steps:

1. Stream Buffer Allocation: A buffer (stream) is allocated to hold the random bytes needed for the matrix entries. The size of this buffer is determined by the number of entries in the matrix (k * n) multiplied by the size of each entry (4 bytes for a uint32_t).
2. Deterministic Random Generation: The randombytes_buf_deterministic function from the Sodium library is used to fill the buffer with random bytes based on the provided seed. This ensures that the same seed will always produce the same sequence of random bytes. It's important to note that this function uses the ChaCha20 algorithm under the hood for secure random number generation.
3. Matrix Entry Population: The function iterates over each entry in the generator matrix and fills it with values derived from the buffer. Each entry is constructed by combining 4 bytes from the buffer into a single uint32_t value, which is then reduced modulo MOD to fit within the required range.
4. Memory Cleanup: The allocated buffer is freed to avoid memory leaks.

Note

The function does not return any value; it directly modifies the provided gen_matrix object. It also requires a seed to ensure deterministic behavior, which is passed as an argument.

Chacha20:- It is a stream cipher that is designed to be fast and secure. It operates on 64-byte blocks and uses a 256-bit key, which is suitable for generating random numbers in this context. It takes streams of bytes, a key and nonce/count as input and produces a key stream as output. This is particularly useful for generating random matrices in cryptographic applications, ensuring that the same seed will always produce the same matrix. In this implementation, the seed is used as key ,stream as the input stream and the nonce is fixed(0). So evevrytime the function is called with the same seed, it will produce the same output stream.

Parameters

n	The total number of columns in the generator matrix. It represents the length of the codewords generated by the matrix.
k	The number of rows in the generator matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
gen_matrix	A pointer to an nmod_mat_t type, which represents the generator matrix to be created. The function initializes this matrix with random entries modulo MOD.
seed	A pointer to an unsigned char array that serves as the seed for the deterministic random number generation. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.
output_file	A pointer to a FILE object where any output or debug information can be written.

Returns

void This function does not return any value. It modifies the provided gen_matrix object directly and fills it with values derived from the seed.

4.15.2.3 generate_keys()

```
void generate_keys (
            struct code * C_A,
            struct code * C1,
            struct code * C2,
            nmod_mat_t H_A,
            nmod_mat_t G1,
            nmod_mat_t G2,
            bool use_seed_mode,
            bool regenerate,
            FILE * output_file,
            unsigned char * h_a_seed,
            unsigned char * g1_seed,
            unsigned char * g2_seed)
```

A function that generates keys for a signature scheme based on the provided parameters and options.

Generates keys for a signature scheme.

This function generates keys for a signature scheme by creating a parity check matrix (H_A) and two generator matrices (G1 and G2) based on the provided code parameters (C_A, C1, C2). It supports both random generation and seed-based generation of matrices. The generated matrices are saved to files, and if seed-based generation is used, the seeds are also saved.

Parameters

C_A	A pointer to a struct code object that contains the parameters for the first code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the parity check matrix H_A.
C1	A pointer to a struct code object that contains the parameters for the first generator code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the first generator matrix G1.
C2	A pointer to a struct code object that contains the parameters for the second generator code, including its length (n), dimension (k), and minimum distance (d). This code is used to generate the second generator matrix G2.
H_A	A pointer to an nmod_mat_t type, which represents the parity check matrix H_A to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
G1	A pointer to an nmod_mat_t type, which represents the first generator matrix G1 to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
G2	A pointer to an nmod_mat_t type, which represents the second generator matrix G2 to be generated. The function initializes this matrix with random entries or from a seed, depending on the options provided.
use_seed_mode	A boolean flag that indicates whether to use seed-based generation for the matrices. If set to true, the function will generate the matrices using a seed; if false, it will generate the matrices randomly.
regenerate	A boolean flag that indicates whether to regenerate the matrices even if they already exist. If set to true, the function will always generate new matrices; if false, it will load the existing matrices if available.

Parameters

output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.
h_a_seed	A pointer to an unsigned char array where the generated seed for the parity check matrix H_A will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.
g1_seed	A pointer to an unsigned char array where the generated seed for the first generator matrix G1 will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.
g2_seed	A pointer to an unsigned char array where the generated seed for the second generator matrix G2 will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.

Returns

void This function does not return any value. It directly modifies the provided matrices (H_A, G1, G2) and saves them to files if necessary. If seed-based generation is used, it also saves the generated seeds to files.

4.15.2.4 generate_parity_check_matrix()

```
void generate_parity_check_matrix (
            slong n,
            slong k,
            slong d,
            nmod_mat_t H,
            FILE * output_file)
```

This function generates a parity check matrix with random entries modulo MOD.

The function initializes a parity check matrix of size (n-k) x n with random entries modulo MOD. It uses the FLINT library's nmod_mat_t type to represent the matrix and fills it with random values using the nmod_mat_randtest function. The matrix is initialized with dimensions (n-k) (number of rows) and n (number of columns), and the entries are generated randomly in the range of 0 to MOD-1. The process involves the following steps:

1. Random State Initialization: A FLINT random state (flint_rand_t) is initialized using flint_randinit. This state is required for generating random numbers in FLINT.
2. Matrix Initialization: The parity check matrix (H) is initialized with n-k rows and n columns, and modulus MOD, using nmod_mat_init.
3. Random Entry Generation: The matrix is filled with random values using nmod_mat_randtest, which uses the previously initialized random state to populate each entry.
4. Random State Cleanup: The random state is cleared with flint_randclear to free any resources associated with it.

Note

The function does not return any value; it directly modifies the provided H object. It also initializes a random state using flint_rand_t to ensure that the random values are generated correctly.

Parameters

n	The total number of columns in the parity check matrix. It represents the length of the codewords that the matrix checks for validity.
k	The number of rows in the parity check matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
H	A pointer to an nmod_mat_t type, which represents the parity check matrix to be created. The function initializes this matrix with random entries modulo MOD.
output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.

Returns

void This function does not return any value. It modifies the provided H object directly and fills it with random values.

4.15.2.5 generate_parity_check_matrix_from_seed()

```
void generate_parity_check_matrix_from_seed (
            slong n,
            slong k,
            slong d,
            nmod_mat_t H,
            const unsigned char * seed,
            FILE * output_file)
```

This function generates a parity check matrix from a seed.

Generates a parity check matrix from a seed.

The function initializes a parity check matrix of size (n-k) x n using a deterministic approach based on a provided seed. The seed is used to generate random entries in the matrix, ensuring that the same seed will always produce the same matrix. The entries are generated modulo MOD. The process involves the following steps:

1. Stream Buffer Allocation: A buffer (stream) is allocated to hold the random bytes needed for the matrix entries. The size of this buffer is determined by the number of entries in the matrix ((n-k) ∗ n) multiplied by the size of each entry (4 bytes for a uint32_t).
2. Deterministic Random Generation: The randombytes_buf_deterministic function from the Sodium library is used to fill the buffer with random bytes based on the provided seed. This ensures that the same seed will always produce the same sequence of random bytes. It's important to note that this function uses the ChaCha20 algorithm under the hood for secure random number generation.
3. Matrix Entry Population: The function iterates over each entry in the parity check matrix and fills it with values derived from the buffer. Each entry is constructed by combining 4 bytes from the buffer into a single uint32_t value, which is then reduced modulo MOD to fit within the required range.
4. Memory Cleanup: The allocated buffer is freed to avoid memory leaks.

Note

The function does not return any value; it directly modifies the provided H object. It also requires a seed to ensure deterministic behavior, which is passed as an argument.

Parameters

n	The total number of columns in the parity check matrix. It represents the length of the codewords that the matrix checks for validity.
k	The number of rows in the parity check matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
H	A pointer to an nmod_mat_t type, which represents the parity check matrix to be created. The function initializes this matrix with random entries modulo MOD.
seed	A pointer to an unsigned char array that serves as the seed for the deterministic random number generation. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.
output_file	A pointer to a FILE object where any output or debug information can be written.

Returns

> void This function does not return any value. It modifies the provided H object directly and fills it with values derived from the seed.

4.15.2.6 generate_random_seed()

```
void generate_random_seed (
            unsigned char * seed)
```

This function generates a random seed of a fixed size.

The function uses the randombytes_buf function from the Sodium library to fill the provided seed buffer with random bytes. The size of the seed is defined by the constant SEED_SIZE, which is set to 32 bytes.

Parameters

seed	A pointer to an unsigned char array where the generated random seed will be stored. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.

Returns

> void This function does not return any value. It directly modifies the provided seed buffer by filling it with random bytes.

4.15.2.7 get_or_generate_matrix_with_seed()

```
void get_or_generate_matrix_with_seed (
            const char * prefix,
            int n,
            int k,
            int d,
            nmod_mat_t matrix,
            void(* generate_func )(slong, slong, slong, nmod_mat_t, FILE *),
            void(* generate_from_seed_func )(slong, slong, slong, nmod_mat_t, const unsigned
char *, FILE *),
            FILE * output_file,
            bool regenerate,
            bool use_seed_mode,
            unsigned char * seed_out)
```

Gets or generates matrix with seed object.

Gets or generates matrix with seed.

This function checks if a matrix file exists for the given parameters (prefix, n, k, d). If it does and regeneration is not requested, it loads the matrix from the file. If the file does not exist or regeneration is requested, it generates a new matrix either using a random generation function or a seed-based generation function. The generated matrix is then saved to a file. This function is designed to handle both random and seed-based generation of matrices, allowing for reproducibility when using the same seed.

Parameters

prefix	A string that serves as a prefix for the filename of the matrix. This prefix is used to create a unique filename based on the parameters n, k, and d.
n	The total number of columns in the matrix. It represents the length of the codewords generated by the matrix.
k	The number of rows in the matrix. It represents the dimension of the code, which is the number of information symbols that can be encoded.
d	The minimum distance of the code. This parameter is not directly used in the function but is typically relevant for the properties of the code being generated.
matrix	A pointer to an nmod_mat_t type, which represents the matrix to be generated or loaded. The function initializes this matrix with random entries or loads it from a file if it already exists.
generate_func	A pointer to a function that generates a matrix with random entries. This function should take parameters (n, k, d, matrix, output_file) and fill the matrix with random values.
generate_from_seed_func	A pointer to a function that generates a matrix from a seed. This function should take parameters (n, k, d, matrix, seed, output_file) and fill the matrix with values derived from the seed.
output_file	A pointer to a FILE object where any output or debug information can be written. This parameter is optional and can be used for logging purposes, but in this implementation, it is not used.
regenerate	A boolean flag that indicates whether to regenerate the matrix even if it already exists. If set to true, the function will always generate a new matrix; if false, it will load the existing matrix if available.
use_seed_mode	A boolean flag that indicates whether to use seed-based generation for the matrix. If set to true, the function will generate the matrix using a seed; if false, it will generate the matrix randomly.
seed_out	A pointer to an unsigned char array where the generated seed will be stored if seed-based generation is used. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file. If seed-based generation is not used, this parameter can be NULL.

Returns

void This function does not return any value. It modifies the provided matrix object directly and saves it to a file if necessary.

4.16 signature-scheme/src/main.c File Reference

Main entry point and core command handling for the signature scheme.

```
#include <string.h>
#include <flint/flint.h>
#include <flint/nmod_mat.h>
#include "params.h"
#include "time.h"
#include "keygen.h"
#include "signer.h"
#include "verifier.h"
#include "utils.h"
#include "constants.h"
```

Functions

- int keygen (int argc, char ∗argv[ ])

  Key generation function for the signature scheme.
- int sign (int argc, char ∗argv[ ])

  Responsible for signing a message using the signature scheme.
- int verify (int argc, char ∗argv[ ])

  Responsible for verifying a signature against a message using the signature scheme.
- int main (int argc, char ∗argv[ ])

  Main function of the signature scheme program.

4.16.1 Detailed Description

Main entry point and core command handling for the signature scheme.

This file implements the main function and the three primary commands for the signature scheme:

- keygen: Key generation
- sign: Message signing
- verify: Signature verification

Each command is handled by its own function, which parses command-line arguments, manages file I/O, and coordinates the use of supporting modules (matrix operations, parameter loading, etc).

4.16.2 Function Documentation

4.16.2.1 keygen()

```
int keygen (
            int argc,
            char * argv[ ])
```

Key generation function for the signature scheme.

It is the key generation function for the signature scheme.

First, it checks for command line arguments to determine if it should use seed mode or regenerate keys. It then opens the output file to write the generated keys. It retrieves user input for the parameters of the keys, initializes matrices for the codes, and generates the keys based on the specified parameters. The generated keys are written to the output file, and the matrices are cleared before closing the output file.

Parameters

argc	The number of command line arguments passed to the keygen function.
argv	An array of strings representing the command line arguments passed to the keygen function.

Returns

int 0 on success, or a non-zero value on failure.

See also

get_user_input
generate_keys
load_params
get_H_A_n
get_H_A_k
get_H_A_d
get_G1_n
get_G1_k
get_G1_d
get_G2_n
get_G2_k
get_G2_d

At first it checks for command line arguments to determine if it should use seed mode or regenerate keys. It then opens the output file to write the generated keys. It retrieves user input for the parameters of the keys, initializes matrices for the codes, and generates the keys based on the specified parameters. The generated keys are written to the output file, and the matrices are cleared before closing the output file.

Parameters

argc	It is the number of command line arguments passed to the keygen function.
argv	It is an array of strings representing the command line arguments passed to the keygen function.

Returns

int It returns 0 on success, or a non-zero value on failure.

See also

get_user_input
generate_keys
load_params
get_H_A_n
get_H_A_k
get_H_A_d
get_G1_n
get_G1_k
get_G1_d
get_G2_n
get_G2_k
get_G2_d

4.16.2.2 main()

```
int main (
            int argc,
            char * argv[ ])
```

Main function of the signature scheme program.

It handles command line arguments to either generate keys, sign a message, or verify a signature. It supports three main commands: keygen, sign, and verify.

• keygen: Generates the keys required for the signature scheme. • sign: Signs a message using the generated keys. • verify: Verifies a signature against a message using the public key. It also checks for the existence of necessary directories (matrix cache and output directory) and initializes them if they do not exist.

Parameters

argc	The number of command line arguments passed to the program. It is used to determine how many arguments were provided and to parse them accordingly.
argv	An array of strings representing the command line arguments passed to the program. Each element in the array corresponds to a command line argument, with argv[0] being the program name and subsequent elements being the actual arguments provided by the user. This array is used to determine the command to execute (key generation, signing, or verification) and to parse any additional options or parameters that may be required for those commands. The program expects specific commands and options to be provided, and it uses this array to handle those commands appropriately.

Returns

int 0 on success, or a non-zero value on failure. The return value indicates the success or failure of the operation performed by the main function. If the command is recognized and executed successfully (key generation, signing, or verification), it returns 0. If there are errors such as missing arguments, unrecognized commands, or file I/O issues, it returns a non-zero value to indicate failure. This allows the calling environment (such as a shell or another program) to determine whether the operation was successful or if there were errors that need to be addressed.

See also

ensure_matrix_cache

ensure_output_directory

4.16.2.3 sign()

```
int sign (
            int argc,
            char * argv[ ])
```

Responsible for signing a message using the signature scheme.

This function is responsible for signing a message using the signature scheme.

It processes command line arguments to get the message file and output signature file. First, it checks if the message file is provided, and if not, it prints usage instructions. Then it loads the parameters for the codes, reads the message from the file or generates it if not found, and normalizes the message length. It initializes matrices for the codes and generates the necessary matrices (H_A, G1, G2) using the specified parameters. The function then generates the signature by calling generate_signature, which computes the signature based on the message and the codes. Finally, it saves the generated signature, hash, and public key to the output directory, clears the matrices, and frees the allocated memory for the message.

Parameters

argc	The number of command line arguments passed to the sign function.
argv	An array of strings representing the command line arguments passed to the sign function.

Returns

int 0 on success, or a non-zero value on failure.

Note

This function uses Flint's nmod_mat_t for matrix operations, which are essential for the signature generation process.

See also

load_params

read_file_or_generate

normalize_message_length

get_or_generate_matrix_with_seed

generate_signature

save_matrix

load_matrix

It processes command line arguments to get the message file and output signature file. At first it checks if the message file is provided, and if not, it prints usage instructions. Then it loads the parameters for the codes, reads the message from the file or generates it if not found, and normalizes the message length. It initializes matrices for the codes and generates the necessary matrices (H_A, G1, G2) using the specified parameters. The function then generates the signature by calling generate_signature, which computes the signature based on the message and the codes. Finally, it saves the generated signature, hash, and public key to the output directory, clears the matrices, and frees the allocated memory for the message.

Parameters

argc	It is the number of command line arguments passed to the sign function.
argv	It is an array of strings representing the command line arguments passed to the sign function.

Returns

int It returns 0 on success, or a non-zero value on failure.

Note

This function uses Flint's nmod_mat_t for matrix operations, which are essential for the signature generation process.

See also

load_params

read_file_or_generate

normalize_message_length

get_or_generate_matrix_with_seed

generate_signature

save_matrix

load_matrix

4.16.2.4 verify()

```
int verify (
            int argc,
            char * argv[])
```

Responsible for verifying a signature against a message using the signature scheme.

This function is responsible for verifying a signature against a message using the signature scheme.

It processes command line arguments to get the message file and signature file. First, it checks if both the message file and signature file are provided, and if not, it prints usage instructions. It loads the parameters for the codes, reads the message from the file, and initializes matrices for the codes. The function then loads the signature from the specified file and generates the parity check matrix (H_A) using the specified parameters. It also loads the hash of the message and the public key matrix (F) from the output directory. Finally, it verifies the signature by calling verify_signature, which checks if the signature is valid for the given message and parameters. The results of the verification are written to an output file.

Parameters

argc	The number of command line arguments passed to the verify function.
argv	An array of strings representing the command line arguments passed to the verify function.

Returns

int 0 on success, or a non-zero value on failure.

Note

These functions use Flint's nmod_mat_t for matrix operations, which are essential for the signature verification process.

See also

load_params

read_file

load_matrix

get_or_generate_matrix_with_seed

verify_signature

It processes command line arguments to get the message file and signature file. At first it checks if both the message file and signature file are provided, and if not, it prints usage instructions. It loads the parameters for the codes, reads the message from the file, and initializes matrices for the codes. The function then loads the signature from the specified file and generates the parity check matrix (H_A) using the specified parameters. It also loads the hash of the message and the public key matrix (F) from the output directory. Finally, it verifies the signature by calling verify_signature, which checks if the signature is valid for the given message and parameters. The results of the verification are written to an output file.

Parameters

argc	It is the number of command line arguments passed to the verify function.
argv	It is an array of strings representing the command line arguments passed to the verify function.

Returns

int It returns 0 on success, or a non-zero value on failure.

Note

These functions use Flint's nmod_mat_t for matrix operations, which are essential for the signature verification process.

See also

load_params

read_file

load_matrix

get_or_generate_matrix_with_seed

verify_signature

4.17 signature-scheme/src/matrix.c File Reference

Implementation of matrix operations for the signature scheme.

```
#include <stdio.h>
#include <sodium.h>
#include <flint/flint.h>
#include <flint/nmod_mat.h>
```

Functions

- void print_matrix (FILE ∗fp, nmod_mat_t matrix)

  The print_matrix function is designed to output the contents of a matrix to a specified file stream, such as a file or the console.
- void transpose_matrix (int rows, int cols, int matrix[rows][cols], int transpose[cols][rows])

  The function computes the transpose of a two-dimensional integer matrix.
- void multiply_matrices_gf2 (nmod_mat_t C, const nmod_mat_t A, const nmod_mat_t B)

  The function performs matrix multiplication over the finite field GF(2)
- static void swap_columns (size_t n, size_t k, size_t first, size_t second, nmod_mat_t H)

  The function swaps two(first and second) columns in the matrix H for a specified range of rows.
- void make_systematic (size_t n, size_t k, nmod_mat_t H)

  The function transforms a parity check matrix H into systematic form.
- void rref (int num_rows, int num_cols, int(∗H)[num_cols])

  The function transforms a binary matrix into its Reduced Row Echelon Form(RREF).

4.17.1 Detailed Description

Implementation of matrix operations for the signature scheme.

Implementation of matrix operations for the signature scheme. This file provides functions for matrix operations such as printing matrices, transposing matrices, multiplying matrices over GF(2), and transforming matrices into systematic form. It also includes functions for performing row reduction to echelon form. The operations are primarily used for handling parity check matrices and generator matrices in the context of error-correcting codes.

4.17.2 Function Documentation

4.17.2.1 make_systematic()

```
void make_systematic (
            size_t n,
            size_t k,
            nmod_mat_t H)
```

The function transforms a parity check matrix H into systematic form.

This function takes a parity check matrix H, which is represented as an nmod_mat_t type from the FLINT library, and transforms it into systematic form. Systematic form means that the first k columns of the matrix will be an identity matrix, and the remaining columns will contain the parity check bits. The function computes r = n - k and scans columns to find unit vectors (columns with a single 1 in the top r rows). When such a column is found, it is swapped into the correct position to form an identity matrix. This continues until r such columns are placed, at which point the matrix is in (partial) systematic form. It's a greedy approach that works well if the matrix is already close to systematic.

Note

> The function assumes that the matrix H is well-formed and that the number of rows n and columns k are correctly defined such that n >= k. It does not perform any bounds checking on the indices.

Parameters

n	The total number of rows in the matrix H, which is used to determine the range of rows that will be affected by the transformation.
k	The number of columns in the matrix H, which is used to determine the number of rows that will be transformed into an identity matrix.

The function uses the nmod_mat_get_entry and nmod_mat_set_entry functions from the FLINT library to access and modify matrix entries. It iterates through each column of the matrix, checking for unit vectors and swapping them into the correct position.

Parameters

H	The parity check matrix that will be transformed into systematic form, represented as an nmod_mat_t type from the FLINT library.

Returns

> void This function does not return a value; it modifies the matrix H in place to transform it into systematic form.

4.17.2.2 multiply_matrices_gf2()

```
void multiply_matrices_gf2 (
            nmod_mat_t C,
            const nmod_mat_t A,
            const nmod_mat_t B)
```

The function performs matrix multiplication over the finite field GF(2)

Performs matrix multiplication over GF(2).

This function multiplies two matrices A and B, both defined over the finite field GF(2), and stores the result in matrix C. The multiplication is performed using bitwise operations, where addition is equivalent to XOR and multiplication is equivalent to AND. The function multiplies matrices over GF(2) using three nested loops. The outer loops iterate over rows of A and columns of B to compute each entry of the result matrix C. For each (i, j) entry, it initializes C[i][j] to 0, then uses the inner loop to XOR the bitwise AND of A[i][k] and B[k][j] into C[i][j]. This performs matrix multiplication using bitwise operations, with & as multiplication and ^ as addition in GF(2). The function assumes the matrices are properly initialized and dimensionally compatible.

Parameters

A	The first matrix to be multiplied, represented as an nmod_mat_t type from the FLINT library.
B	The second matrix to be multiplied, also represented as an nmod_mat_t type from the FLINT library.

Note

> The function assumes that the matrices A and B are compatible for multiplication, meaning the number of columns in A must equal the number of rows in B. It also assumes that the result matrix C has been initialized with appropriate dimensions to hold the product of A and B.

The function uses the nmod_mat_get_entry and nmod_mat_set_entry functions from the FLINT library to access and modify matrix entries. It iterates through each row of A and each column of B, computing the product for each entry in C.

Parameters

C	The result matrix where the product of A and B will be stored, also represented as an nmod_mat_t type from the FLINT library.

Returns

> void This function does not return a value; it modifies the matrix C in place to store the result of the multiplication.

4.17.2.3 print_matrix()

```
void print_matrix (
            FILE * fp,
            nmod_mat_t matrix)
```

The print_matrix function is designed to output the contents of a matrix to a specified file stream, such as a file or the console.

Prints the contents of a matrix to a file stream.

It takes two parameters: a FILE ∗fp, which is the output stream, and an nmod_mat_t matrix, which represents the matrix to be printed. The function begins by printing the dimensions of the matrix in the format <rows x columns matrix>, using the r and c fields of nmod_mat_t matrix.

It then iterates through each row and column of the matrix, printing each entry in a formatted manner. Each row is enclosed in square brackets, and entries are separated by spaces. After printing all entries in a row, it moves to the next line for the next row.

Parameters

fp	The file pointer to which the matrix will be printed. This can be a file opened in write mode or stdout for console output.

Note

The function assumes that the matrix is non-empty and that the nmod_mat_t structure is properly initialized. It does not handle any errors related to file operations or matrix initialization.

Parameters

matrix	The matrix to be printed, represented as an nmod_mat_t type from the FLINT library. This structure allows efficient access and manipulation of matrix data under modular arithmetic.

Returns

void This function does not return a value; it performs output operations directly to the specified file stream.

4.17.2.4 rref()

```
void rref (
             int num_rows,
             int num_cols,
             int(*) H[num_cols])
```

The function transforms a binary matrix into its Reduced Row Echelon Form(RREF).

Transforms a matrix into Reduced Row Echelon Form.

This function takes a binary matrix H, represented as a two-dimensional array of integers, and transforms it into its Reduced Row Echelon Form (RREF). The RREF is a form where each leading entry in a row is 1, and all entries in the column above and below each leading 1 are 0. The function performs forward and back substitution to achieve this form. It iterates through the columns of the matrix, finding non-zero elements to use as pivot points, and then eliminates other entries in the same column by XORing rows.

Parameters

num_rows	The number of rows in the matrix H, which is used to determine the range of rows that will be affected by the transformation.
num_cols	The number of columns in the matrix H, which is used to determine the range of columns that will be affected by the transformation.
H	The binary matrix that will be transformed into Reduced Row Echelon Form, represented as a two-dimensional array of integers. Each entry in the matrix is either 0 or 1, representing elements in GF(2).

Note

The notation (∗H)[num_cols] in the function parameter list means that H is a pointer to an array of num_↩ cols integers. In other words, H points to the first element of a 2D array where each row contains num_cols elements. This allows you to use H[i][j] inside the function to access the element at row i and column j, just like with a regular 2D array. The compiler needs to know the size of each row (num_cols) to correctly compute the memory offset for each element. This is why num_cols must be specified in the parameter type.

4.17.2.5 swap_columns()

```
void swap_columns (
             size_t n,
             size_t k,
             size_t first,
             size_t second,
             nmod_mat_t H)  [static]
```

The function swaps two(first and second) columns in the matrix H for a specified range of rows.

This function is designed to swap two columns in a matrix H, specifically for the first n-k rows of the matrix. The function takes the number of rows n, the number of columns k, and the indices of the two columns to be swapped (first and second). It iterates through each row from 0 to n-k-1 and swaps the entries in the specified columns.

Parameters

n	The total number of rows in the matrix H.
k	The number of columns in the matrix H, which is used to determine the range of rows that will be affected by the column swap.
first	The index of the first column to be swapped.
second	The index of the second column to be swapped.

Note

> The function assumes that the indices first and second are valid column indices within the range of the matrix H, and that n and k are correctly defined such that n $>=$ k. It does not perform any bounds checking on the indices.

Parameters

H	The matrix in which the columns will be swapped, represented as an nmod_mat_t type from the FLINT library.

The function uses the nmod_mat_get_entry and nmod_mat_set_entry functions from the FLINT library to access and modify matrix entries. It iterates through each row from 0 to n-k-1, swapping the entries in the specified columns.

4.17.2.6 transpose_matrix()

```
void transpose_matrix (
             int rows,
             int cols,
             int matrix[rows][cols],
             int transpose[cols][rows])
```

The function computes the transpose of a two-dimensional integer matrix.

Computes the transpose of a matrix.

This function takes a matrix defined by its number of rows and columns, and fills a new matrix with the transposed values. The transpose of a matrix is obtained by swapping its rows and columns, meaning that the element at position (i, j) in the original matrix becomes the element at position (j, i) in the transposed matrix. So that the elements of the firsr row in the original matrix becomes the elements at the first column in the transposed matrix.

Parameters

rows	The number of rows in the original matrix.
cols	The number of columns in the original matrix.

Note

> The function assumes that the input matrix is well-formed and that the transpose matrix has been allocated with appropriate dimensions.

Parameters

matrix	The original matrix to be transposed, represented as a two-dimensional array of integers.
transpose	The transposed matrix, which will be filled with the transposed values of the original matrix.

Returns

> void This function does not return a value; it modifies the transpose matrix in place.

4.18 signature-scheme/src/params.c File Reference

Implementation of parameter handling functions for the signature scheme.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sodium.h>
#include "params.h"
#include "constants.h"
```

Functions

- void init_params (void)

  Initializes the libsodium library.
- uint32_t random_range (uint32_t min, uint32_t max)

  It generates a random number in the range [min, max].
- static void generate_random_params (Params ∗p)

  Generates random parameters for the Params structure.
- bool get_yes_no_input (const char ∗prompt)

  Get the yes no input object.
- static void get_param_input (Params ∗p, const char ∗name)

  Get the param input object.
- void get_user_input (Params ∗g1, Params ∗g2, Params ∗h)

  Get user input for parameters.
- uint32_t get_H_A_n (void)

  Returns the n parameter of the concatenated code H_A.
- uint32_t get_H_A_k (void)

  Returns the k parameter of the concatenated code H_A.

- uint32_t get_H_A_d (void)

  Returns the d parameter of the concatenated code H_A.
- uint32_t get_G1_n (void)

  Returns the n parameter of the generator matrix G1.
- uint32_t get_G1_k (void)

  Returns the k parameter of the generator matrix G1.
- uint32_t get_G1_d (void)

  Returns the d parameter of the generator matrix G1.
- uint32_t get_G2_n (void)

  Returns the n parameter of the generator matrix G2.
- uint32_t get_G2_k (void)

  Returns the k parameter of the generator matrix G2.
- uint32_t get_G2_d (void)

  Returns the d parameter of the generator matrix G2.

Variables

- static char ∗ MESSAGE = NULL

  Static pointer to a message string.
- static size_t MESSAGE_LEN = 0

  It's the length of the message.

Static Global Parameters

Parameters used for the generator and concatenated codes.

- static Params G1
- static Params G2
- static Params H_A

4.18.1 Detailed Description

Implementation of parameter handling functions for the signature scheme.

This file provides functions for initializing, generating, reading, and handling parameters used in the signature scheme. It manages parameters for generator matrices and concatenated codes, supports random and user-defined parameter generation, and handles persistent storage of parameters.

Main functionalities include:

- Initialization of the libsodium library for cryptographic randomness.
- Generation of random parameters within specified ranges.
- Interactive user input for code parameters, including support for BCH code parameters.
- Reading and writing parameter sets to a persistent file.
- Accessor functions for retrieving current parameter values.

The parameters managed include those for two generator matrices (G1, G2) and a concatenated code (H_A).

4.18.2 Function Documentation

4.18.2.1 generate_random_params()

```
void generate_random_params (
            Params * p)  [static]
```

Generates random parameters for the Params structure.

This function generates random values for n, k, and d within specified ranges. It ensures that n is greater than both k and d. here the n, k, and d values are generated in the range of 16 to 17 for n, 6 to 7 for k, and 3 to 4 for d.

Parameters

p	Pointer to the Params structure to be filled with random values.

4.18.2.2 get_G1_d()

```
uint32_t get_G1_d (
            void )
```

Returns the d parameter of the generator matrix G1.

Returns

The value of G1.d

4.18.2.3 get_G1_k()

```
uint32_t get_G1_k (
            void )
```

Returns the k parameter of the generator matrix G1.

Returns

The value of G1.k

4.18.2.4 get_G1_n()

```
uint32_t get_G1_n (
            void )
```

Returns the n parameter of the generator matrix G1.

Returns

The value of G1.n

4.18.2.5 get_G2_d()

```
uint32_t get_G2_d (
            void )
```

Returns the d parameter of the generator matrix G2.

Returns

The value of G2.d

4.18.2.6 get_G2_k()

```
uint32_t get_G2_k (
            void )
```

Returns the k parameter of the generator matrix G2.

Returns

The value of G2.k

4.18.2.7 get_G2_n()

```
uint32_t get_G2_n (
            void )
```

Returns the n parameter of the generator matrix G2.

Returns

The value of G2.n

4.18.2.8 get_H_A_d()

```
uint32_t get_H_A_d (
            void )
```

Returns the d parameter of the concatenated code H_A.

Returns

The value of H_A.d

4.18.2.9 get_H_A_k()

```
uint32_t get_H_A_k (
            void )
```

Returns the k parameter of the concatenated code H_A.

Returns

The value of H_A.k

4.18.2.10 get_H_A_n()

```
uint32_t get_H_A_n (
             void )
```

Returns the n parameter of the concatenated code H_A.

Returns

The value of H_A.n

4.18.2.11 get_param_input()

```
void get_param_input (
             Params * p,
             const char * name)  [static]
```

Get the param input object.

This function prompts the user to input parameters for a given code (G1, G2 and H_A). It takes a pointer to a Params structure and a name string as arguments. The function will repeatedly ask the user for input until valid parameters are provided (i.e., n $>$ k and n $>$ d). If the user inputs invalid parameters, it will prompt them to try again. the name param is used to identify which code the parameters are for (e.g., "G1", "G2", or "H_A"). The validity of the parameters is checked to ensure that `n` is either greater than `k` or `d`. If the input is invalid, it will prompt the user to try again.

Parameters

p	is the pointer to the Params structure where the parameters will be stored.
name	is the name of the code for which parameters are being input (e.g., "G1", "G2", or "H_A").

4.18.2.12 get_user_input()

```
void get_user_input (
             Params * g1,
             Params * g2,
             Params * h)
```

Get user input for parameters.

This function checks if a saved parameter file exists and prompts the user to use it. If not, it asks the user whether they want to use BCH code or input G1 and G2 parameters manually. It generates random parameters if the user chooses not to input them. The function also saves the parameters to a file for future use. At first it checks if a saved parameter file exists. If it does, it prompts the user to use it. If the user chooses not to use the saved parameters, it asks whether they want to use BCH code or input G1 and G2 parameters manually.

- For the BCH code, it calculates the parameters based on user input for `m` and `t`, ensuring that the derived values for `n`, `k`, and `d` are consistent across G1 and G2. The parameters for H_A are derived from G1 and G2. `m` is the degree of the BCH code, and `t` is the error-correcting capability. The parameters are calculated as follows:
- - n = 2^m - 1 (length of the codeword)

- - k = m $*$ t (length of the message)
- - d = 2 $*$ t + 1 (minimum distance of the code)
- If the user chooses to input G1 and G2 parameters manually, it prompts for each parameter (n, k, d) and checks their validity. If the user does not want to input parameters, it generates random parameters for G1 and G2.

After gathering the parameters, it saves them to a file named Defined as PARAM_PATH for future use. The parameters for G1, G2, and H_A are printed to the console for confirmation.

Parameters

g1	Pointer to Params structure for G1 parameters.
g2	Pointer to Params structure for G2 parameters.
h	Pointer to Params structure for H_A parameters.

4.18.2.13 get_yes_no_input()

```
bool get_yes_no_input (
             const char * prompt)
```

Get the yes no input object.

Gets yes/no input from user.

Parameters

prompt	It is a string that will be displayed to the user as a prompt.

This function prompts the user for a yes or no response. It reads the user's input and checks for the 1st character of it. It returns true for 'y' or 'Y', and false for 'n' or 'N'. If the input is invalid, it will terminate the program with failure.

Returns

true if the input's 1st character is 'y' or 'Y'.

false if the input's 1st character is 'n' or 'N'.

4.18.2.14 init_params()

```
void init_params (
             void )
```

Initializes the libsodium library.

This function should be called before using any other libsodium functions. The libsodium library is used for generating random numbers in this implementation.

Note

If libsodium fails to initialize, the program will exit with an error message.

See also

sodium_init

4.18.2.15 random_range()

```
uint32_t random_range (
             uint32_t min,
             uint32_t max)
```

It generates a random number in the range [min, max].

This function uses the libsodium library to generate a uniform random unsigned 32bit number. Basically, it generates a random number in the range [0, max-min] and adds with min.

Parameters

min	The minimum value of the range (inclusive).
max	The maximum value of the range (inclusive).

Returns

uint32_t The generated random number.

Note

This function assumes that `max` is greater than or equal to `min`.

See also

randombytes_uniform

4.18.3 Variable Documentation

4.18.3.1 G1

```
Params G1 [static]
```

Parameters of the generator matrix for the first code (C1).

4.18.3.2 G2

```
Params G2 [static]
```

Parameters for the generator matrix for the second code (C2).

4.18.3.3 H_A

```
Params H_A [static]
```

Parameters for the concatenated code (C_A).

4.18.3.4 MESSAGE

```
char* MESSAGE = NULL [static]
```

Static pointer to a message string.

This variable is used to store a message as a dynamically allocated string. It is initialized to NULL and should be assigned before use. Being static, it has internal linkage and is only accessible within this source file.

4.18.3.5 MESSAGE_LEN

```
size_t MESSAGE_LEN = 0  [static]
```

It's the length of the message.

This is used to allocate memory for the message and to ensure that the message is processed correctly. This is used to determine the size of the message. It is initialized to 0 and will be set when the message is read or generated.

4.19 signature-scheme/src/signer.c File Reference

This file contains the implementation of the signature generation function.

```
#include <sodium.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include "signer.h"
#include "utils.h"
#include "matrix.h"
#include "constants.h"
```

Functions

- void generate_signature (nmod_mat_t bin_hash, const unsigned char ∗message, size_t message_len, struct code C_A, struct code C1, struct code C2, nmod_mat_t H_A, nmod_mat_t G1, nmod_mat_t G2, nmod_↩ mat_t F, nmod_mat_t signature, FILE ∗output_file)

  This function generates a signature based on the provided parameters.

4.19.1 Detailed Description

This file contains the implementation of the signature generation function.

This function generates a signature based on the provided parameters, including the binary hash of the message, the code parameters, and the generator matrices. It uses the Sodium library for cryptographic operations and matrix operations for handling the codewords. The signature is generated by combining the binary hash with the generator matrix and ensuring that the weight of the signature meets the minimum distance requirement of the code. The function also prints debug information if the PRINT flag is set.

4.19.2 Function Documentation

4.19.2.1 generate_signature()

```
void generate_signature (
            nmod_mat_t bin_hash,
            const unsigned char * message,
            size_t message_len,
            struct code C_A,
            struct code C1,
            struct code C2,
            nmod_mat_t H_A,
            nmod_mat_t G1,
            nmod_mat_t G2,
            nmod_mat_t F,
            nmod_mat_t signature,
            FILE * output_file)
```

This function generates a signature based on the provided parameters.

Generates a signature based on the provided parameters.

This function generates a digital signature for a given message using a code-based cryptographic approach. The signature is created by constructing a hybrid generator matrix from two codes (C1 and C2), salting the message, hashing it using SHA-256, and encoding the resulting binary hash vector as a codeword. The process ensures that the signature meets the minimum weight requirement specified by code C_A. It uses matrix operations over finite fields via the FLINT library (`nmod_mat_t`) and cryptographic hashing and randomness via Libsodium.

The function performs the following steps:

1. Allocate an array J to hold a random selection of indices from [0, C_A.n - 1].
2. Generate a random permutation of size C1.n and store it in J.
3. Initialize the matrix G_star (size C1.k × C_A.n) which will hold the combined generator matrix.
4. For each column index i from 0 to C_A.n - 1: a. If i is in J (i.e., selected for G1), copy the corresponding column from G1 to G_star. b. Otherwise, copy the next available column from G2 to G_star.
5. Free the random index array J as it's no longer needed.
6. If PRINT is enabled, print the contents of G_star to the output file.
7. Transpose G_star to obtain G_star_T.
8. Compute F = H_A × G_star_T. This may be used for constraint checking or debugging.
9. Begin a loop to compute a valid signature: a. Allocate and fill a buffer with the original message followed by a random salt of the same length. b. Hash the salted message using SHA-256. c. Convert the resulting hash into a binary vector (0s and 1s) to fill the bin_hash matrix. d. Multiply bin_hash with G_star to produce the signature matrix. e. If the weight (Hamming weight) of the signature is less than C_A.d (minimum distance), repeat the loop.
10. If PRINT is enabled, print the binary hash matrix to the output file.
11. Clear memory used by G_star and G_star_T.

    Note

    > This function assumes that the input matrices and codes are properly initialized and that the MOD constant is defined for finite field operations.

Parameters

bin_hash	It is a matrix that will hold the binary hash of the message.
message	It is the input message for which the signature is being generated.
message_len	It is the length of the input message in bytes.
C_A	It is the derived code that defines the parameters for the signature generation, including the length of the code (n), the length of the message (k), and the minimum distance (d).
C1	It is the first code used in the signature generation process, which provides part of the generator matrix.
C2	It is the second code used in the signature generation process, which provides the remaining part of the generator matrix.
H_A	It is the parity-check matrix for the derived code C_A, which is used to ensure that the generated signature meets the required properties.
G1	It is the generator matrix for the first code C1, which is used to construct part of the hybrid generator matrix.
G2	It is the generator matrix for the second code C2, which is used to construct the remaining part of the hybrid generator matrix.
F	It is a matrix that will hold the product of the parity-check matrix H_A and the transposed hybrid generator matrix G_star_T. This is used for debugging or verification purposes.
signature	It is the output matrix that will hold the generated signature for the input message.
output_file	It is a file pointer to the output file where debug information will be printed if the PRINT flag is set.

Note

The function uses the Sodium library for cryptographic operations and the FLINT library for matrix operations.

4.20 signature-scheme/src/utils.c File Reference

implementation of utility functions for file and directory management, matrix operations, and random number generation used in the signature scheme.

```
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "params.h"
#include "utils.h"
#include "constants.h"
```

Functions

- void ensure_matrix_cache ()

  checks the existence of the matrix cache directory and creates it if it does not exist.
- void ensure_output_directory ()

  checks the existence of the output directory and creates it if it does not exist.

- static int compare_ints (const void ∗a, const void ∗b)

  This function compares two integers for sorting purposes.
- long weight (nmod_mat_t array)

  Calculates the Hamming weight of a given matrix row.
- double binary_entropy (double p)

  Calculates the binary entropy of a given probability.
- void generate_random_set (unsigned long upper_bound, unsigned long size, unsigned long set[size])

  Generates a random set of unique integers within a specified range and sorts the original set in ascending order.
- char ∗ generate_matrix_filename (const char ∗prefix, int n, int k, int d)

  Generates a filename for a matrix based on a prefix and its dimensions.
- void save_matrix (const char ∗filename, const nmod_mat_t matrix)

  Saves a matrix to a text file in a specific format (FLINT matrix format).
- int load_matrix (const char ∗filename, nmod_mat_t matrix)

  Loads a matrix from a text file in a specific format (FLINT matrix format).
- int file_exists (const char ∗filename)

  Checks if a file exists by attempting to open it in read mode.
- char ∗ generate_seed_filename (const char ∗prefix, int n, int k, int d)

  Generates a filename for a seed based on a prefix and its parameters.
- bool save_seed (const char ∗filename, const unsigned char ∗seed)

  Saves a seed to a binary file.
- bool load_seed (const char ∗filename, unsigned char ∗seed)

  Loads a seed from a binary file.
- char ∗ read_file (const char ∗filename)

  Reads the contents of a file into a dynamically allocated string.
- char ∗ read_file_or_generate (const char ∗filename, int msg_len)

  Reads a message from a file or generates a random message if the file is empty or does not exist.
- bool load_params (struct code ∗C_A, struct code ∗C1, struct code ∗C2)

  Loads parameters for the codes from a file.
- char ∗ normalize_message_length (const char ∗msg, size_t msg_len, size_t target_len, size_t ∗final_len_out)

  Normalizes the length of a message to a target length by padding or truncating it.

4.20.1 Detailed Description

implementation of utility functions for file and directory management, matrix operations, and random number generation used in the signature scheme.

implementation of utility functions for file and directory management, matrix operations, and random number generation used in the signature scheme. This file provides functions to ensure the existence of necessary directories, create and manage matrix files, randomly generate sets, perform various matrix operations and message manipulations. It begins with standard and project-specific headers, and includes functions like ensure_matrix_cache and ensure_output_directory to create necessary directories if missing. It defines compare_ints for sorting, weight for computing the Hamming weight of a matrix row, and binary_entropy for entropy calculation. Random subsets are generated using generate_random_set, which applies the Fisher-Yates shuffle and sorting. Matrix-related tasks such as naming, saving, and loading are handled using FLINT matrix functions through generate_matrix_↩ filename, save_matrix, and load_matrix. File handling includes checking for existence, saving seeds, and managing input/output files. read_file_or_generate ensures a message is available by reading it from a file or generating one if missing, while normalize_message_length ensures proper length. Finally, load_params loads code parameters from a file into program structures. The module emphasizes robustness and reusability, streamlining I/O, randomness, and matrix handling for the broader cryptographic system.

4.20.2 Function Documentation

4.20.2.1 binary_entropy()

```
double binary_entropy (
            double p)
```

Calculates the binary entropy of a given probability.

Calculates the binary entropy of a probability.

The binary entropy function computes the entropy of a binary random variable with probability p of being 1. It uses the formula: H(p) = -p ∗ log2(p) - (1 - p) ∗ log2(1 - p). The function checks if p is within the valid range (0, 1) and returns 0 if p is 0 or 1, as there is no uncertainty in those cases. The logarithm is computed using the log2 function from the math library, which calculates the base-2 logarithm.

Parameters

p	A double representing the probability of a binary event occurring, where 0 < p < 1.

Returns

double The binary entropy of the given probability p, which is a measure of uncertainty in bits.

Note

The function assumes that the input probability p is a valid value between 0 and 1 (exclusive). If p is outside this range, the function will return 0, indicating no uncertainty. It does not handle cases where p is NaN or infinite.

4.20.2.2 compare_ints()

```
int compare_ints (
            const void * a,
            const void * b)  [static]
```

This function compares two integers for sorting purposes.

This function is used as a comparison function for sorting arrays of integers. It takes two pointers to integers, dereferences them to get the actual integer values, and then compares them. It returns -1 if the first integer is less than the second, 1 if the first integer is greater than the second, and 0 if they are equal. This function is typically used with the qsort function from the C standard library to sort arrays of integers in ascending order.

Parameters

a	Pointer to the first element (of type const void∗) to compare.
b	Pointer to the second element (of type const void∗) to compare.

Returns

int Comparison result: -1, 0, or 1.

4.20.2.3 ensure_matrix_cache()

```
void ensure_matrix_cache ()
```

checks the existence of the matrix cache directory and creates it if it does not exist.

Ensures matrix cache directory exists.

This function checks if the directory "matrix_cache" exists. If it does not, it creates the directory with permissions set to 0700 (read, write, and execute permissions for the owner only). This is useful for storing cached matrices used in the signature scheme. It uses the stat function to check for the directory's existence and mkdir to create it if necessary. The function does not return any value; it simply ensures that the directory is present before any matrix operations are performed.

Returns

It does not return any value; it simply ensures that the matrix cache directory is present before any matrix operations are performed.

4.20.2.4 ensure_output_directory()

```
void ensure_output_directory ()
```

checks the existence of the output directory and creates it if it does not exist.

Ensures output directory exists.

This function checks if the directory "output" exists. If it does not, it creates the directory with permissions set to 0700 (read, write, and execute permissions for the owner only). This is useful for storing output files generated by the signature scheme. It uses the stat function to check for the directory's existence and mkdir to create it if necessary. The function does not return any value; it simply ensures that the output directory is present before any output operations are performed.

Returns

It does not return any value; it simply ensures that the output directory is present before any output operations are performed.

4.20.2.5 file_exists()

```
int file_exists (
             const char * filename)
```

Checks if a file exists by attempting to open it in read mode.

Checks if a file exists.

It takes a filename as input and tries to open the file using fopen with the "r" mode, which is for reading. If the file cannot be opened (for example, if it does not exist), fopen returns NULL. In this case, the function returns 0 to indicate that the file does not exist. If the file is successfully opened, it is immediately closed using fclose, and the function returns 1 to indicate that the file exists.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to check for existence. The function will attempt to open this file in read mode.

Returns

int 1 if the file exists (i.e., it can be opened in read mode), or 0 if the file does not exist (i.e., it cannot be opened).

Note

The function does not perform any additional checks on the file, such as verifying its contents or permissions. It simply checks for the existence of the file by trying to open it. If the file is successfully opened, it is closed immediately after checking.

4.20.2.6 generate_matrix_filename()

```
char * generate_matrix_filename (
            const char * prefix,
            int n,
            int k,
            int d)
```

Generates a filename for a matrix based on a prefix and its dimensions.

Generates a filename for a matrix based on prefix and dimensions.

The function constructs a filename for a matrix by concatenating a predefined cache directory with a prefix and the dimensions of the matrix (n, k, d). The resulting filename is formatted as "cache_dir/prefix_n_k_d.txt", where cache_dir is defined as "matrix_cache/", and prefix, n, k, and d are provided as parameters. The function allocates memory for the filename string, formats it using sprintf, and returns the pointer to the generated filename.

Parameters

prefix	A pointer to a constant character string that serves as a prefix for the filename. This prefix is typically used to identify the type of matrix or its specific characteristics. Example prefixes could be "H_A", "G1", or "G2", depending on the context of the matrix being generated or stored.
n	The length of the code.
k	The dimension of the code.
d	The minimum distance of the code.

Returns

char∗ A pointer to a dynamically allocated string containing the generated filename. If memory allocation fails, it returns NULL.

Note

The function allocates memory for the filename string, so it is the caller's responsibility to free this memory when it is no longer needed. The maximum length of the generated filename is defined by MAX_FILENAME↩_LENGTH, which should be set appropriately to accommodate the longest expected filename.

4.20.2.7 generate_random_set()

```
void generate_random_set (
            unsigned long upper_bound,
            unsigned long size,
            unsigned long set[size])
```

Generates a random set of unique integers within a specified range and sorts the original set in ascending order.

Generates a random set of unique integers within a specified range.

The function generates a random set of unique integers from 0 to upper_bound - 1, ensuring that the size of the set is equal to size. It uses the modern Fisher-Yates shuffle algorithm to randomly permute an array of integers from 0 to upper_bound - 1, and then selects the first size elements from this shuffled array. The resulting set is sorted in ascending order using the qsort function with a custom comparison function.

Parameters

upper_bound	An unsigned long integer representing the upper limit of the range from which unique integers will be selected. The function will generate integers in the range [0, upper_bound).
size	An unsigned long integer representing the number of unique integers to be generated in the set. The function will ensure that the size of the generated set is equal to this value.
set	A pointer to an array of unsigned long integers where the generated unique integers will be stored. The size of this array should be at least size elements to hold the generated set.

Note

Algorithm:

1. Initialize an array arr containing all integers from 0 to upper_bound - 1.
2. Shuffle the array in-place using the modern Fisher-Yates shuffle: ○ Iterate from the last element to the second element. ○ In each iteration, generate a random index j such that 0 <= j <= i. ○ Swap the elements at indices i and j.
3. Copy the first size elements from the shuffled array into set.
4. Sort the set array in ascending order.

4.20.2.8 generate_seed_filename()

```
char * generate_seed_filename (
            const char * prefix,
            int n,
            int k,
            int d)
```

Generates a filename for a seed based on a prefix and its parameters.

Generates a filename for a seed based on prefix and parameters.

The function constructs a filename for a seed by concatenating a predefined cache directory with a prefix and the parameters n, k, and d. The resulting filename is formatted as "cache_dir/prefix_n_k_d_seed.bin", where cache_dir is defined as "matrix_cache/", and prefix, n, k, and d are provided as parameters. The function allocates memory for the filename string, formats it using snprintf, and returns the pointer to the generated filename.

Parameters

prefix	A pointer to a constant character string that serves as a prefix for the filename. This prefix is typically used to identify the type of seed or its specific characteristics, such as "H_A", "G1", or "G2", depending on the context of the seed being generated or stored.
n	An integer representing the length of the code. This value is used to uniquely identify the seed associated with a specific code length.
k	An integer representing the dimension of the code. This value is used to uniquely identify the seed associated with a specific code dimension.
d	An integer representing the minimum distance of the code. This value is used to uniquely identify the seed associated with a specific code minimum distance.

Returns

> char∗ A pointer to a dynamically allocated string containing the generated filename. If memory allocation fails, it returns NULL. The caller is responsible for freeing this memory when it is no longer needed.

4.20.2.9 load_matrix()

```
int load_matrix (
            const char * filename,
            nmod_mat_t matrix)
```

Loads a matrix from a text file in a specific format (FLINT matrix format).

Loads a matrix from a text file in FLINT format.

It opens a file with the specified filename for reading. If it fails to open the file, it returns 0. It then reads the dimensions of the matrix (number of rows and columns) from the file. If it fails to read these dimensions, it closes the file and returns 0. The function clears any existing data in the provided matrix, initializes it with the specified dimensions, and then reads each entry of the matrix from the file, setting the corresponding entry in the matrix using nmod_mat_set_entry. If it fails to read any value, it closes the file and returns 0. Finally, it closes the file and returns 1 to indicate success.

Parameters

filename	A pointer to a constant character string that specifies the name of the file from which the matrix will be loaded.
matrix	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library.

Returns

> int 1 if the matrix was successfully loaded from the file, or 0 if there was an error (e.g., file not found, failed to read dimensions or values).

Note

> The file should be in a specific text format that includes the dimensions of the matrix followed by its entries.

4.20.2.10 load_params()

```
bool load_params (
            struct code * C_A,
            struct code * C1,
            struct code * C2)
```

Loads parameters for the codes from a file.

Loads parameters for codes from a file.

It opens a file named "params.txt" in read mode and reads key-value pairs from it. The keys correspond to parameters of the codes, such as "H_A_n", "H_A_k", "H_A_d", "G1_n", "G1_k", "G1_d", "G2_n", "G2_k", and "G2_d". For each key, it assigns the corresponding value to the appropriate field in the provided code structures (C_A, C1, and C2). If the file cannot be opened, it prints an error message and returns false. If all parameters are successfully loaded, it returns true.

Parameters

C_A	A pointer to the concatenated code structure, which contains parameters for the concatenated code (C_A).
C1	A pointer to the first generator code structure, which contains parameters for the first code (C1).
C2	A pointer to the second generator code structure, which contains parameters for the second code (C2).

Returns

true If the parameters were successfully loaded from the file, meaning that the file was opened, all key-value pairs were read, and the corresponding fields in the code structures were set.

false If there was an error opening the file or if any key-value pair could not be read, indicating that the parameters were not loaded successfully.

Note

The function assumes that the file "params.txt" exists and is formatted correctly with key-value pairs. If any key is missing or if the file cannot be read, the function will not set the corresponding fields in the code structures.

4.20.2.11 load_seed()

```
bool load_seed (
            const char * filename,
            unsigned char * seed)
```

Loads a seed from a binary file.

The function takes a filename and a pointer to an unsigned char array (seed) as input. It opens the specified file in binary read mode ("rb"). If the file cannot be opened, it returns false. It then reads SEED_SIZE bytes from the file into the seed array using fread. After reading, it closes the file and checks if the number of bytes read matches SEED_SIZE. If they match, it returns true, indicating that the seed was successfully loaded; otherwise, it returns false.

Parameters

filename	A pointer to a constant character string that specifies the name of the file from which the seed will be loaded. The file should contain binary data representing the seed.
seed	A pointer to an unsigned char array where the loaded seed data will be stored. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.

Returns

true If the seed was successfully loaded from the file, meaning that the file was opened, the seed data was read, and the correct number of bytes was read.

false If there was an error opening the file or if the number of bytes read does not match SEED_SIZE, indicating that the seed was not loaded successfully.

4.20.2.12 normalize_message_length()

```
char * normalize_message_length (
            const char * msg,
            size_t msg_len,
            size_t target_len,
            size_t * final_len_out)
```

Normalizes the length of a message to a target length by padding or truncating it.

Normalizes message length by padding or truncating.

The function takes a message, its length, a target length, and an optional pointer to store the final length. It allocates memory for a new message of the target length. If the original message is shorter than the target length, it copies the original message and fills the remaining space with random uppercase letters (A-Z). If the original message is longer than the target length, it truncates it to fit. If the lengths match, it simply copies the original message. The function returns the newly created message and updates the final length if requested.

Parameters

msg	A pointer to a constant character string that represents the original message to be normalized. The message can be of any length, and the function will either pad it with random characters or truncate it to fit the target length.
msg_len	The length of the original message in bytes. This value is used to determine whether the message needs to be padded or truncated to match the target length.
target_len	The desired length of the normalized message. The function will ensure that the final message has this exact length by either padding it with random characters or truncating it if necessary.
final_len_out	A pointer to a size_t variable where the final length of the normalized message will be stored. This parameter is optional; if it is NULL, the function will not update the final length.

Returns

char∗ A pointer to a dynamically allocated string containing the normalized message.

Note

The function allocates memory for the new message, so it is the caller's responsibility to free this memory when it is no longer needed. If memory allocation fails, the function will print an error message and return NULL.

4.20.2.13 read_file()

```
char * read_file (
            const char * filename)
```

Reads the contents of a file into a dynamically allocated string.

Reads the contents of a file into a string.

It opens the specified file in read mode, checks if the file was opened successfully, and then reads its contents into a buffer. The function first seeks to the end of the file to determine its length, rewinds to the beginning, allocates memory for the buffer, and reads the file's contents into it. Finally, it closes the file and returns the buffer containing the file's contents as a null-terminated string. If any step fails (e.g., file not found, memory allocation failure), it prints an error message and returns NULL.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to be read. The function will attempt to open this file in read mode and read its contents.

Returns

> char∗ A pointer to a dynamically allocated string containing the contents of the file. If the file cannot be opened or is empty, it returns NULL.

Note

> The caller is responsible for freeing the returned buffer after use to avoid memory leaks.

4.20.2.14 read_file_or_generate()

```
char * read_file_or_generate (
            const char * filename,
            int msg_len)
```

Reads a message from a file or generates a random message if the file is empty or does not exist.

Reads a message from file or generates random message.

It attempts to open the specified file in read mode. If the file is successfully opened, it checks its length. If the length is zero or less, it generates a random message of a specified length and saves it to the file. If the file contains valid data, it reads the contents into a dynamically allocated string and returns it. If the file cannot be opened, it generates a random message, saves it to the file, and returns that message.

Parameters

filename	A pointer to a constant character string that specifies the name of the file to read the message from. If the file does not exist or is empty, a random message will be generated and saved to this file.
msg_len	An integer representing the length of the message to be generated if the file is empty or does not exist. The function will generate a random message of this length using uppercase letters (A-Z).

Returns

> char∗ A pointer to a dynamically allocated string containing the message read from the file or the generated random message. If memory allocation fails or if there is an error reading the file, it returns NULL. The caller is responsible for freeing the returned string after use.

4.20.2.15 save_matrix()

```
void save_matrix (
            const char * filename,
            const nmod_mat_t matrix)
```

Saves a matrix to a text file in a specific format (FLINT matrix format).

Saves a matrix to a text file in FLINT format.

The function opens a file with the specified filename for writing. If it fails to open the file, it prints an error message and returns. It then retrieves the number of rows and columns of the matrix using nmod_mat_nrows and nmod↩
_mat_ncols, respectively, and writes these dimensions to the file. After that, it iterates through each entry of the matrix, retrieves its value using nmod_mat_entry, and writes it to the file in a space-separated format. Finally, it closes the file.

Parameters

filename	A pointer to a constant character string that specifies the name of the file where the matrix will be saved. The file will be created if it does not exist, or overwritten if it does.
matrix	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library. The matrix should be initialized and populated with values before calling this function. The function will save the matrix in a specific text format that includes its dimensions followed by its entries.

Note

> The function does not perform any error checking on the matrix itself, such as ensuring it is initialized or has valid dimensions. It assumes that the matrix is properly set up before calling this function. The file will be created in the current working directory, and if the file already exists, it will be overwritten.

4.20.2.16 save_seed()

```
bool save_seed (
            const char * filename,
            const unsigned char * seed)
```

Saves a seed to a binary file.

The function takes a filename and a pointer to an unsigned char array (seed) as input. It opens the specified file in binary write mode ("wb"). If the file cannot be opened, it returns false. It then writes the seed data to the file using fwrite, which writes SEED_SIZE bytes from the seed array to the file. After writing, it closes the file and checks if the number of bytes written matches SEED_SIZE. If they match, it returns true, indicating that the seed was successfully saved; otherwise, it returns false.

Parameters

filename	A pointer to a constant character string that specifies the name of the file where the seed will be saved. The file will be created if it does not exist, or overwritten if it does.
seed	A pointer to an unsigned char array that contains the seed data to be saved. The size of this array should be equal to SEED_SIZE, which is defined in the constants header file.

Returns

> true If the seed was successfully saved to the file, meaning that the file was opened, the seed data was written, and the correct number of bytes was written.
>
> false If there was an error opening the file or if the number of bytes written does not match SEED_SIZE, indicating that the seed was not saved successfully.

4.20.2.17 weight()

```
long weight (
            nmod_mat_t array)
```

Calculates the Hamming weight of a given matrix row.

Calculates the Hamming weight of a matrix row.

The Hamming weight is the number of non-zero elements in a row of a matrix. This function iterates through the first row of the provided matrix and counts how many entries are equal to 1, which corresponds to the Hamming weight. It assumes that the matrix is represented as an nmod_mat_t type from the FLINT library, which allows for efficient access to matrix entries.

Parameters

array	A pointer to an nmod_mat_t type, which represents a matrix in the FLINT library.

Returns

long The Hamming weight of the first row of the matrix, which is the count of entries equal to 1.

Note

The function assumes that the matrix has at least one row and that the entries are in the range of 0 to MOD-1, where MOD is defined in the FLINT library. It does not handle cases where the matrix is empty or has no rows.

4.21 signature-scheme/src/verifier.c File Reference

Contains the implementation of the signature verification function.

```
#include <stdio.h>
#include <stdbool.h>
#include "verifier.h"
#include "matrix.h"
#include "utils.h"
#include "constants.h"
```

Functions

- void verify_signature (nmod_mat_t bin_hash, size_t message_len, unsigned long sig_len, nmod_mat_t signature, nmod_mat_t F, struct code C_A, nmod_mat_t H_A, FILE ∗output_file)

  Verifies a digital signature.

4.21.1 Detailed Description

Contains the implementation of the signature verification function.

This file contains the implementation of the signature verification function. It verifies a digital signature by checking if the product of the binary hash of the message and the generator matrix equals the product of the parity-check matrix and the transposed signature. It uses the FLINT library for matrix operations and prints debug information.

4.21.2 Function Documentation

4.21.2.1 verify_signature()

```
void verify_signature (
            nmod_mat_t bin_hash,
            size_t message_len,
            unsigned long sig_len,
            nmod_mat_t signature,
            nmod_mat_t F,
            struct code C_A,
            nmod_mat_t H_A,
            FILE * output_file)
```

Verifies a digital signature.

This function verifies a digital signature by checking if the product of the binary hash of the message and the generator matrix equals the product of the parity-check matrix and the transposed signature. It uses the FLINT library for matrix operations and prints debug information.

Parameters

bin_hash	Binary hash matrix for the message
message_len	Length of the message
sig_len	Length of the signature
signature	Signature matrix to verify
F	Combined matrix F = H_A * G*^T
C_A	Code parameters for the concatenated code
H_A	Parity check matrix for the concatenated code
output_file	File to write verification output to