KECCAK ALGO (SHA3)

PROJET KECCAK REPORT

11 DÉCEMBRE 2024

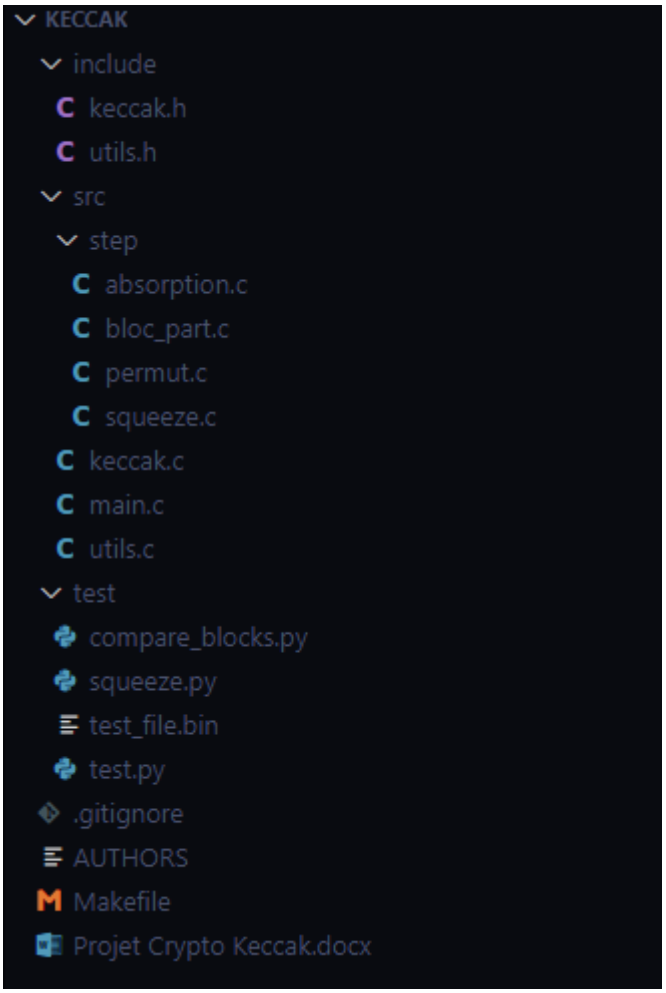
BOISSON Thomas
David Aime Oumbe

"Cryptography is the art of creating puzzles that are difficult to solve but easy to verify."
- Joan Daemen (co-créateur de Keccak)

1. Résumé du projet

Ce projet consiste à implémenter l'algorithme Keccak, plus précisément la variante SHA3-256, en binôme. L'objectif principal est de développer une implémentation fonctionnelle capable de traiter des fichiers binaires de taille raisonnable (environ 50 Mo) dans un temps acceptable et sans provoquer une surcharge de mémoire.

2. Architecture du projet



L'architecture du projet proposée se compose de 3 grands dossier :

- Le dossier include comprenant les prototypes des fonctions utilisées

- Le dossier src comprenant le main.c ainsi que keccak.c et ses sous fichiers représentants les étapes dans le sous dossier "step"

- Le dossier test est un dossier un peu brouillon… ne comprenant pas de test suites automatisé mais plutôt des test case individuel représentants les problèmes rencontrés lors du développement du projet, tel la comparaison de hash, ou le debug de N partie

Un makefile est défini possédant 4 règles dont la règle principale "all", deux tests et une règle clean.

3. Algorithme Keccak en C

Dans le développement du projet, voulant manipuler notre mémoire nous même, nous nous sommes penchés vers les langages bas level, nous avons donc choisi de développer le projet en C, nous permettant de gérer notre mémoire et avoir une vitesse d'exécution rapide (on hésitait avec le C++ pour la modernité de ses classes et librairies rendant le travail plus facile mais le C reste plus rapide)

4. main()

Nous commençons dans le main.c qui va prendre en argument un fichier (son nom), initialiser un tableau “hash” de la taille SHA3_256_HASH_SIZE défini dans le header keccak.h (=32) et nous appellerons la fonction keccak avec ces deux arguments.

Le tableau hash sera le résultat final, le hash du fichier que nous aurons “squeeze” a travers le tableau state initialisé dans la fonction (nous partons du principe qu’il n’est pas besoin de résumé comment fonctionne l’algorithme)

Enfin, nous pourrons print le hash résulté.

```c
int main(int argc, char *argv[])
{
    if (TEST == 1)
    {
        printf("Test squeeze avec un state initialisé à 0\n");
        test_squeeze();
        return 0;
    }

    if (argc != 2)
    {       You, il y a 7 heures • final project, TODO pdf
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }

    const char *filename = argv[1];
    uint8_t hash[SHA3_256_HASH_SIZE];

    keccak(filename, hash);

    printf("hash SHA3-256 de '%s':\n", filename);
    print_hash(hash, SHA3_256_HASH_SIZE);

    return 0;
}
```

5. keccak()

```c
void keccak(const char *filename, uint8_t *hash)
{
    uint8_t *blocks;
    size_t num_blocks = bloc_part(filename, &blocks);

    uint64_t state[KECCAK_STATE_SIZE] = {0};

    absorption(blocks, num_blocks, state);

    squeeze(state, hash, SHA3_256_HASH_SIZE);

    free(blocks);
}       You, il y a 1 seconde • Uncommitted changes
```

La fonction keccak() fonctionne sur 3 steps :

- bloc_part() : Le but de cette fonction est de calculer et créer le nombre de blocs nécessaires pour la suite de l'exécution de l'algorithme, en fonction de la taille du fichier d'entrée. Elle segmente les données en morceaux compatibles avec la structure de Keccak et prépare ces blocs pour les étapes suivantes.

- absorption() : Cette étape consiste à intégrer les blocs de données dans l'état interne de l'algorithme Keccak. Chaque bloc est combiné avec l'état actuel à l'aide d'opérations bitwise (comme XOR), et des transformations non linéaires sont appliquées pour diffuser les données dans tout l'état.

- squeeze() : Cette dernière étape extrait les bits nécessaires pour produire le hash final. Après avoir absorbé tous les blocs, l'algorithme génère la sortie (le hash) en lisant les bits de l'état interne, souvent en plusieurs tours pour obtenir la longueur désirée.

Au début on initialise un pointeur sur les blocs sur la stack (que l'on ira ensuite allouera dynamiquement plus tard) et un tableau "state" de taille KECCAK_STATE_SIZE (=25) rempli de 0, après exécution complète de l'algorithme, on libère les blocs.

6. block_part()

```c
size_t bloc_part(const char *filename, uint8_t **blocks_out)
{
    printf("==== Bloc part loading... ====\n");

    uint8_t *file_content;
    size_t filesize = read_file(filename, &file_content);

    size_t block_size = KECCAK_RATE;
    size_t num_blocks = (filesize + block_size - 1) / block_size;

    *blocks_out = calloc(num_blocks, block_size); // On remplit les blocs de zéros
    if (!*blocks_out)
    {
        perror("Erreur d'allocation mémoire");
        free(file_content);
        exit(EXIT_FAILURE);
    }
    memcpy(*blocks_out, file_content, filesize); // On copie le contenu du fichier dans les blocs

    if (filesize % block_size != 0)
    {
        apply_padding(*blocks_out, filesize, block_size, num_blocks);
    }

    free(file_content);

    printf("Taille du fichier : %zu octets\n", filesize);
    printf("Nombre de blocs calculés : %zu\n", num_blocks);

    printf("==== Bloc part loaded ====\n\n");
    return num_blocks;
}
```

La fonction bloc_part() sert à préparer les blocs de données nécessaires pour l'algorithme Keccak. Elle fonctionne en trois étapes principales :

- Lecture du fichier : Elle lit le contenu du fichier spécifié en entrée et détermine sa taille (en octets).

- Calcul des blocs : Elle segmente le fichier en blocs de taille fixe, définie par la constante KECCAK_RATE. Si le fichier ne correspond pas exactement à un multiple de la taille d'un bloc, le nombre total de blocs est ajusté pour inclure un dernier bloc partiellement rempli.

- Remplissage et padding :
    1. Les blocs sont initialisés avec des zéros avec un calloc.
    2. Le contenu du fichier est copié dans les blocs.

3. Si le fichier ne remplit pas complètement le dernier bloc, un padding (selon la spécification Keccak) est appliqué pour marquer les débuts et fins de données.

```c
void apply_padding(uint8_t *blocks, size_t filesize, size_t block_size, size_t num_blocks)
{
    blocks[filesize] = 0x06; // On met le premier bit à 1 et le reste à 0
    blocks[num_blocks * block_size - 1] |= 0x80; // On met le dernier bit à 1
}
```

7. absorption()

```c
void absorption(uint8_t *blocks, size_t num_blocks, uint64_t state[KECCAK_STATE_SIZE])
{
    printf("==== Absorption loading... ====\n");
    size_t rate = KECCAK_RATE / 8;

    for (size_t i = 0; i < num_blocks; i++)
    {
        for (size_t j = 0; j < rate; j++)
            state[j] ^= ((uint64_t *)blocks)[i * rate + j];

        if (i == 0) // Test case
        {
            save_block_to_file("test/first_block.txt", &((uint64_t *)blocks)[i * rate], rate);
            printf("Taille du bloc extrait pour test : %zu entiers de 64 bits\n", rate);
        }
        permut(state);      You, il y a 10 heures • whole project
    }
    printf("==== Absorption loaded ====\n\n");
}
```

La fonction absorption() constitue une étape clé de l'algorithme Keccak, où les blocs de données sont intégrés dans l'état interne du hash.
Elle commence par parcourir chaque bloc, un à un, et applique une opération XOR entre chaque sous-partie du bloc et les mots correspondants de l'état interne.
Cette étape permet de mélanger progressivement les données d'entrée avec l'état interne.

Une fois qu'un bloc est absorbé, une permutation est appliquée à l'état interne.
Cette permutation repose sur l'application successive des quatre transformations fondamentales de Keccak : θ (Theta), ρ (Rho), π (Pi) et χ (Chi), chacune contribuant à la diffusion et au mélange des bits.

Ce cycle de transformations est répété 24 fois, comme spécifié dans l'algorithme Keccak.

```c
void permut(uint64_t state[KECCAK_STATE_SIZE])
{
    for (int round = 0; round < 24; round++)
    {
        // printf("\n=== Début du tour %d ===\n", round);
        // print_state(state, "Initialisation", round);

        theta(state);
        // print_state(state, "Theta", round);

        rho(state);
        // print_state(state, "Rho", round);

        pi(state);
        // print_state(state, "Pi", round);

        chi(state);
        // print_state(state, "Chi", round);

        iota(state, RC[round]);
        // print_state(state, "Iota", round);
    }
}
```

Pour des tests et validations, un extrait du premier bloc est également sauvegardé dans un fichier, afin de vérifier son contenu et sa taille.

L'ensemble de ces opérations garantit que l'état interne est modifié de manière à rendre les données d'entrée irréversiblement mélangées, préparant ainsi l'algorithme pour l'étape finale de génération du hash.

8. squeeze()

```c
void squeeze(uint64_t state[KECCAK_STATE_SIZE], uint8_t *hash, size_t hash_size)
{
    printf("==== Squeeze loading... ====\n");
    size_t rate_bytes = KECCAK_RATE;
    size_t bytes_extracted = 0;

    while (bytes_extracted < hash_size)
    {
        size_t to_copy = (hash_size - bytes_extracted < rate_bytes)
                       ? hash_size - bytes_extracted
                       : rate_bytes;

        memcpy(hash + bytes_extracted, state, to_copy);

        bytes_extracted += to_copy;
        printf("Bytes extracted: %zu\n", bytes_extracted);
        printf("Hash size: %zu\n", hash_size);
        if (bytes_extracted < hash_size)
            permut(state);
    }
    printf("==== Squeeze loaded ====\n\n");
}
```

La fonction squeeze() est responsable de générer le hash final à partir de l'état interne modifié durant l'étape d'absorption. Elle procède en extrayant des portions de l'état interne, appelées "bytes", et en les ajoutant au buffer de sortie qui contiendra le hash.

La fonction commence par déterminer combien de bytes peuvent être extraits à chaque itération, en fonction de la taille du hash demandé (hash_size) et de la constante KECCAK_RATE. À chaque tour, elle copie les bytes nécessaires depuis l'état interne dans le buffer de hash, puis met à jour le compteur des bytes extraits.

Si la taille du hash n'est pas entièrement couverte par l'état interne lors d'une itération, la fonction applique une permutation sur l'état interne pour générer une nouvelle portion de bytes. Ce processus est répété jusqu'à ce que le nombre total de bytes extraits corresponde à la taille du hash demandé.

À la fin, la fonction garantit que le buffer contient un hash conforme, prêt à être utilisé. La logique d'extraction et de permutation assure que l'ensemble des bits de l'état interne est utilisé pour produire un résultat unique et sécurisé.

9. Tests case

Les tests case crée lors de l'élaboration du projet sont complètement indépendants, ou le but était de débugger certaines parties de l'algo dans le but d'obtenir le hash valide

test.py, le but de ce fichier est de lancer sur le fichier "test_file.bin" l'algo keccak par le code python et le binaire compilé pour enfin comparé les hash obtenu (le bon hash par définition est celui du code python)

compare_blocks.py, ce fichier de test sert à comparer le premier bloc créé lors de l'étape d'absorption dans le code C avec l'équivalent créé dans le script pour comparer ces blocs et voir si l'étape a bien XORé le bloc.

squeeze.py, ce fichier avait pour but de tester l'absorption et la permutation sur un tableau state rempli de 0, donnant ainsi un hash rempli de 0 aussi, le but de ce fichier était d'étudier le comportement de l'algo et le comparé avec celui du code source C.