Part I

Preprocessing

Chapter 1

Introduction

The combination of the `FOOBAR`-modified Miner and the `FOOBAR`-server yields *preimage files*, containing (counter, nonce) pairs. In this document, we call this a **preimage**. Each preimage requires 12 bytes, and we expect to have 3 lists of $2^{31.7}$ entries each, for a total storage space of 117Gbyte.

The preimage files must be checked (i.e. the corresponding hashes must have 33 zero bits). They must then be hashed, and 64 bits of each hash stored to disk. This yields three hash lists, of 25.4Gbyte each. These lists have to be sorted and checked for duplicates. Thus in total, we expect to need ≈ 150 Gbyte of storage.

Each preimage is associated to a 256-bit **full hash**, of which 33 bits at least must be zero. More precisely, bits $[0:33]$ are zero. From this full hash, we extract a subset of 64 (uniformly distributed) bits (bits $[33:97]$) that we call the **hash**. Furthermore, we extract a k-bit *partionning key* from the full hash by taking bits $[97:97+k]$.

Each input list is partitionned according the the partitionning key into 2^k sublists. This serves two purposes

1. The sublists are smaller and thus more manageable.
2. All triplets in $A^{[i]} \times B^{[j]} \times C^{[i \oplus j]}$ are known to have a zero-sum on the bits of the partitionning key.

With $k = 10$, for instance, the sublists are ≈ 32 Mbyte and there are $k^2 = 1$ million subtasks to solve, which is probably enough. Solving one such subtask should require less than 10 CPU-hours.

On the other hand, we do not strictly require that a "task" in the task distribution scheme maps exactly to sublists. We may want to keep sublists of RAM-size (say 1 Gbyte), and split processing of sublists into smaller "tasks".

For various reasons, we enforce that the 64-bit hashes are unique in each sublist. For each sublist, we actually maintain a dictionnary `hash` $\rightarrow$ `preimage`.

The miner generates ≈ 550 preimages per second, which means roughly 550 Mbyte per day of fresh data. Ideally, we would like the total amount of work required to integrate this much data to the

dictionnary to be as low as possible.

We use the following strategy :

1. When a new preimage file is ready, split it into 2^k **sub-dictionnaries** by partitionning according to the partitionning key. Potential duplicate preimages are now confined into a single sub-dictionnary.
2. Sort the sub-dictionnaries by hash. They must fit in RAM in order to do this easily.
3. For each partitionning key, perform a multiway merge on all sub-dictionnaries. Detect duplicate hashes and write down the hashes in order into a **hash file**.

Concretely, a preimage file contains a sequence of `struct preimage_t`. A split dictionnary contains a sequence of `dict_t`. These types are described in `preprocessing.h`.

Filesystem-wise, we use the following conventions :

- Preimage files are named `preimage/<kind>.<anything>`, where `<kind>` is one of `foo`, `bar` or `foobar`.
- Unsorted sub-dictionnaries extracted from `preimage/<name>` are named `dict/<key>/<name>.unsorted`, where `<key>` is the hex representation of the partitionning key (preferably left-padded with zeroes).
- The sorted version of `dict/<key>/<name>` is `dict/<key>/<name>.sorted`.
- The (merged) hashfiles are in `hash/<kind>.<key>`

All programs may assume that the directories already exist.

As stated above, the *kind* of a preimage/dictionary file can be inferred from its name.

4 ⟨*preprocessing.h* 4⟩≡

```
#ifndef _PREPROCESSING_H
#define _PREPROCESSING_H
#include "../types.h"

enum kind_t {
        FOO,
        BAR,
        FOOBAR
};

struct preimage_t {
        i64 counter;
        u32 nonce;
} __attribute__((packed));

struct dict_t {
        u64 hash;
        struct preimage_t preimage;
} __attribute__((packed));
```

⟨*More stuff* 7⟩

```
enum kind_t file_get_kind(const char *filename);
u32 file_get_partition(const char *filename);
#endif
```

Obtaining parts of the filename is easy thanks to the POSIX functions `basename` and `dirname`... except that these modify their argument. We thus have to make a copy first and clean it up afterwards.

6 ⟨ * 6⟩≡

```
#define _XOPEN_SOURCE  500  /* for strdup */
#include <string.h>
#include <strings.h>        /* strncasecmp */
#include <stdlib.h>
#include <libgen.h>
#include <err.h>
#include "preprocessing.h"

enum kind_t file_get_kind(const char *filename)
{
        enum kind_t out = -1;
        char *filename_ = strdup(filename);
        char *base = basename(filename_);
        if (strncasecmp(base, "foobar", 6) == 0)
                out = FOOBAR;
        else if (strncasecmp(base, "foo", 3) == 0)
                out = FOO;
        else if (strncasecmp(base, "bar", 3) == 0)
                out = BAR;
        else
                errx(1, "cannot determine kind of file %s", filename);
        free(filename_);
        return out;
}

u32 file_get_partition(const char *filename)
{
        char *filename_ = strdup(filename);
        char *dir = dirname(filename_);
        char *base = basename(dir);
        u32 out = strtol(base, NULL, 16);
        free(filename_);
        return out;
}
```

1.1 Additional Preprocessing for Joux's Algorithm

Lists A and B must not be sorted, so that it would make sense to randomly permute each hash file.

For C, it seems reasonable to precompute the slices (thus enabling more intensive precomputations in order to increase the size of slices). In Addition to slice boundaries, we could store : CM instead of C, as well as M and M^{-1}. Advantage: this drops the "compute-time" dependency on M4RI.

The C hash file could be replaced by a sequence of :

7 ⟨*More stuff* 7⟩≡ (4)

```
struct slice_t {
        u64 M[64];
        u64 Minv[64];
        u64 n;
        u64 l;
        u64 CM[];
} __attribute__((packed));

static inline u64 LEFT_MASK(u8 n)
{
        return ~((1ull << (64 - n)) - 1);
}
```

An important caveat is that for Joux's algorithm, the hash files **must not** be sorted. Otherwise, probabilistic reasonings based on Chernoff bounds fail badly.

If the "compute" phase happens on the BGQ, then hash files should be byte- swapped in advance. By the above type definition, a slice file is essentially a sequence of `u64`, so it is easy to byte-swap.

Chapter 2

Hasher

This file describes a shared function that computes the hash associated to a given preimage. This code is embedded into the (modified) code that runs on the miner. We use OpenSSL's implementation of SHA256.

The miner produces (counter, nonce) pairs (*preimages* in our parlance). The preprocessing step builds *dictionnaries* of hash → preimage.

Given a `struct preimage_t`, we compute the full hash (and check its validity). Given the full hash we extract a 64-bit substring, known to be uniformly distributed, as well as the partitionning key. The full hash fits into an array of $8 \times$ `u32`.

The public interface is thus:

1 ⟨*hasher.h* 1⟩≡

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

static inline void build_plaintext_block(int kind, struct preimage_t *preimage, char *buffer) {
        ⟨Assemble plaintext block 2b⟩
}
extern bool compute_full_hash(int kind, struct preimage_t *preimage, u32 *hash);
static inline u64 extract_partial_hash(u32 *hash) {
        ⟨Return partial hash 3d⟩
}
static inline u64 extract_partitioning_key(int k, u32 *hash) {
        ⟨Return partitionning key 3e⟩
}
```

The smallest functions are `static inline` for speed. Now we come to the implementation.

2a ⟨* 2a⟩≡

```
#include "sha256.h"
#include "hasher.h"

bool compute_full_hash(int kind, struct preimage_t *preimage, u32 *hash)
{
        u32 *block[20];
        build_plaintext_block(kind, preimage, (char *) block);
        ⟨Compute full hash of plaintext block 3b⟩
        ⟨Return true if hash is valid 3c⟩
}
```

To hash a preimage, we must first build the 80-byte "plaintext block", i.e. the input of the hash function.

2b ⟨Assemble plaintext block 2b⟩≡ (1)

```
static const char *TEMPLATE[3] = {
"FOO-0x0000000000000000                                                    ",
"BAR-0x0000000000000000                                                    ",
"FOOBAR-0x0000000000000000                                                 "
};
static const u8 NIBBLE[16] = {48, 49, 50, 51, 52, 53, 54, 55,
                              56, 57, 65, 66, 67, 68, 69, 70};
⟨Start from the template given by kind 2c⟩
⟨Write down counter as ASCII hexadecimal 2d⟩
⟨Write the (byteswapped) nonce at the end 3a⟩
```

Let's do this.

2c ⟨Start from the template given by kind 2c⟩≡ (2b)

```
memcpy(buffer, TEMPLATE[kind], 80);
```

2d ⟨Write down counter as ASCII hexadecimal 2d⟩≡ (2b)

```
u64 counter = preimage->counter;
int j = (kind == 2) ? 25 : 22;
while (counter > 0) {
        u8 nibble = counter & 0x000f;
        counter >>= 4;
        buffer[j] = NIBBLE[nibble];
        j--;
}
```

3a ⟨*Write the (byteswapped)* `nonce` *at the end* 3a⟩≡ (2b)

```
u32 *block = (u32 *) buffer;
block[19] = __builtin_bswap32(preimage->nonce);
```

At this stage, the plaintext block is ready. We apply the SHA256 hash function *twice*, and that's it.

3b ⟨*Compute full hash of plaintext block* 3b⟩≡ (2a)

```
u8 md[32];
SHA256((u8 *) block, 80, md);
SHA256((u8 *) md, 32, (u8 *) hash);
```

A priori, the nonces are valid at difficulty 33. This means that the top 33 bits of the hash must be zero.

3c ⟨*Return* `true` *if hash is valid* 3c⟩≡ (2a)

```
return (hash[7] == 0x00000000) && ((hash[6] & 0x80000000) == 0x0000000000);
```

We extract 64 bits of the hash from `hash[5]` and `hash[6]`, excluding the most-significant bit of `hash[6]` which is always zero. We replace it with the most-significant bit of `hash[4]`.

3d ⟨*Return partial hash* 3d⟩≡ (1)

```
return (((u64) hash[5]) << 32) ^ hash[6] ^ (hash[4] & 0x80000000);
```

The partitioning key is taken from the high-order bits of `hash[4]`.

3e ⟨*Return partitionning key* 3e⟩≡ (1)

```
return (hash[4] & 0x7fffffff) >> (31 - k);
```

Chapter 3

Splitter

3.1 Global Structure

Splitting preimage files requires computing all the hashes, and this is CPU-bound. Therefore, we parallelize the hash computations.

The main strategy is as follows. A "reader" reads the input file and sends blocks of (counter, nonce) pairs to "mappers". Each mapper assembles the plaintext block, hash it, and dispatch it in one of its output buffers. When a buffer is full, it is flushed to the "writer". Each of these workers is an independent process. Processes interact through the MPI messaging library.

This program reads a preimage file, and compute all the hashes. It checks that the (counter, nonce) pairs indeed yield hashes with 33 leading zero bits, and dispatches the dictionnary items into sub-dictionnary files.

1 ⟨ * 1⟩≡
⟨*Header files to include* 2a⟩
⟨*Global variables* 2c⟩
⟨*The main program* 2b⟩

We need the usual standard headers.

2a ⟨*Header files to include* 2a⟩≡ (1)

```
#define _XOPEN_SOURCE 500
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <getopt.h>
#include <mpi.h>
#include "preprocessing.h"
#include "hasher.h"
```

2b ⟨*The main program* 2b⟩≡ (1)

```
int main(int argc, char **argv)
{
        ⟨Initialization 3c⟩
        ⟨Process the command line 3a⟩
        ⟨Finish setup 5c⟩
        ⟨Start reader, mappers and writers 6a⟩
        MPI_Finalize();
        exit(EXIT_SUCCESS);
}
```

The possible command-line arguments are the size k of the partitionning key, the output directory (`dict/`) and the preimage file. We enforce that these arguments are actually present. The "kind" of the preimage file is inferred from its name.

2c ⟨*Global variables* 2c⟩≡ (1) 3b▷

```
struct option longopts[3] = {
        {"partitioning-bits", required_argument, NULL, 'b'},
        {"output-dir", required_argument, NULL, 'd'},
        {NULL, 0, NULL, 0}
};
int bits = -1;
char *output_dir = NULL;
```

3a ⟨*Process the command line* 3a⟩≡ (2b)

```
signed char ch;
while ((ch = getopt_long(argc, argv, "", longopts, NULL)) != -1) {
        switch (ch) {
        case 'b':
                bits = atoi(optarg);
                break;
        case 'd':
                output_dir = optarg;
                break;
        default:
                errx(1, "Unknown option\n");
        }
}
if (bits == -1)
        errx(1, "missing required option --partitioning-bits");
if (optind != argc - 1)
        errx(1, "missing (or extra) filenames");
if (output_dir == NULL)
        errx(1, "missing required option --output-dir");
char *in_filename = argv[optind];
enum kind_t kind = file_get_kind(in_filename);
```

Before starting to dwelve into the code of the threads, we must set up a bit of global context. The number of output files is $2^{\texttt{bits}}$. The number of mappers is the total number of processes minus 2.

3b ⟨*Global variables* 2c⟩+≡ (1) ◁2c 5b▷

```
i32 rank, size;
```

3c ⟨*Initialization* 3c⟩≡ (2b)

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
⟨Setup MPI datatype for preimages 4⟩
⟨Setup MPI datatype for dictionnary entries 5a⟩
```

To easily send / receive arrays of this though MPI, we setup a custom MPI datatype. Please refer to the MPI guide for details.

4 ⟨*Setup MPI datatype for preimages* 4⟩≡ (3c)

```
struct preimage_t sample[2];
MPI_Datatype PreimageStruct, PreimageType;
MPI_Datatype type[2] = {MPI_UINT64_T, MPI_UINT32_T};
int blocklen[2] = {1, 1};
MPI_Aint disp[2];
MPI_Aint base, sizeofentry;

/* compute displacements of structure components */
MPI_Get_address(&sample[0].counter, &disp[0]);
MPI_Get_address(&sample[0].nonce, &disp[1]);
MPI_Get_address(sample, &base);
disp[0] -= base;
disp[1] -= base;
MPI_Type_create_struct(2, blocklen, disp, type, &PreimageStruct);
MPI_Type_commit(&PreimageStruct);

/* If compiler does padding in mysterious ways, the following may be safer */
MPI_Get_address(sample + 1, &sizeofentry);
sizeofentry -= base;
MPI_Type_create_resized(PreimageStruct, 0, sizeofentry, &PreimageType);

/* quick safety check */
int x;
MPI_Type_size(PreimageType, &x);
if ((x != sizeof(struct preimage_t)) || (x != 12))
        errx(1, "data types size mismatch");
```

5a ⟨*Setup MPI datatype for dictionnary entries* 5a⟩≡ (3c)

```
struct dict_t sample2[2];
MPI_Datatype type2[2] = {MPI_UINT64_T, PreimageType};
MPI_Datatype DictStruct, DictType;

/* compute displacements of structure components */
MPI_Get_address(&sample2[0].hash, &disp[0]);
MPI_Get_address(&sample2[0].preimage, &disp[1]);
MPI_Get_address(sample2, &base);
disp[0] -= base;
disp[1] -= base;
MPI_Type_create_struct(2, blocklen, disp, type2, &DictStruct);
MPI_Type_commit(&DictStruct);

MPI_Get_address(sample2 + 1, &sizeofentry);
sizeofentry -= base;
MPI_Type_create_resized(DictStruct, 0, sizeofentry, &DictType);
```

MPI processes are numbered starting at zero. The reader is process 0. The writer is process 1. The remaining processes are mappers. To easily distinguish between messages, we use tags.

5b ⟨*Global variables* 2c⟩+≡ (1) ◁3b 5d▷

```
static const int READER_REQUEST_TAG = 0;
static const int NONCE_BLOCK_TAG = 1;
static const int HASH_BLOCK_TAG = 2;
static const int EOF_TAG = 3;
static const int KEY_TAG = 4;
u32 n_mapper, n_slots;
```

5c ⟨*Finish setup* 5c⟩≡ (2b)

```
n_mapper = size - 2;
n_slots = 1 << bits;
if (rank == 0 && n_mapper <= 0)
        errx(1, "not enough MPI processes. Need 3, have %d", size);
```

Nonces and hashes are processed in batches. The output buffer size is much smaller, because with $k = 10$, each mapper will have 1024 of these.

5d ⟨*Global variables* 2c⟩+≡ (1) ◁5b

```
static const u32 READER_BUFFER_SIZE = 65536;
static const u32 WRITER_BUFFER_SIZE = 1024;
```

Thanks to the MPI programming model, starting everything is easy.

6a ⟨*Start reader, mappers and writers* 6a⟩≡ (2b)

```
if (rank == 0) {
        ⟨Reader 6b⟩
} else if (rank == 1) {
        ⟨Writer 10b⟩
} else {
        ⟨Mapper 8a⟩
}
```

3.2 Reading the Preimage Files

The mapper threads will request preimage blocks from the reader. The reader will reply with either a preimage block or an `EOF` message.

6b ⟨*Reader* 6b⟩≡ (6a)

```
printf("Reader started. %d mappers.\n", n_mapper);
u32 preimages_read = 0;
double start = MPI_Wtime();
double wait = 0;
FILE *f = fopen(in_filename, "r");
if (f == NULL)
        err(1, "fopen on %s", in_filename);
while (1) {
        ⟨Read a preimage block from f in buffer 7a⟩
        ⟨Wait for a request and send back buffer 7b⟩
        ⟨Print status report 7d⟩
}
fclose(f);
for (u32 i = 0; i < n_mapper; i++) {
        ⟨Wait for a request and send back EOF 7c⟩
}
printf("\nReader finished. %d preimages read, total wait = %.1f s\n", preimages_read, wait);
```

Reading the file is straightforward. We use a buffer of `struct preimage_t`.

7a ⟨*Read a preimage block from* `f` *in* `buffer` 7a⟩≡ (6b)

```
struct preimage_t buffer[READER_BUFFER_SIZE];
size_t n_items = fread(buffer, sizeof(struct preimage_t), READER_BUFFER_SIZE, f);
if (ferror(f))
        err(1, "fread in reader");
if (n_items == 0 && feof(f))
        break;
```

Sending the block to the mapper is also quite simple. Because we accept requests from anyone, we must be able to tell who asked us for a block. We also use a specific "tag" for block requests. We use `MPI_Bsend`, because it allows us to get back to reading the file faster (at the expense of using a bit more memory).

7b ⟨*Wait for a request and send back* `buffer` 7b⟩≡ (6b)

```
MPI_Status status;
double wait_start = MPI_Wtime();
MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, READER_REQUEST_TAG, MPI_COMM_WORLD, &status);
wait += MPI_Wtime() - wait_start;
MPI_Send(buffer, n_items, PreimageType, status.MPI_SOURCE, NONCE_BLOCK_TAG, MPI_COMM_WORLD);
preimages_read += n_items;
```

Once all the files have been processed, the mappers must be told to stop sending requests. We use an empty block with the EOF tag.

7c ⟨*Wait for a request and send back* `EOF` 7c⟩≡ (6b)

```
MPI_Status status;
MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, READER_REQUEST_TAG, MPI_COMM_WORLD, &status);
MPI_Send(NULL, 0, MPI_INT, status.MPI_SOURCE, EOF_TAG, MPI_COMM_WORLD);
```

We implement a simple form of verbosity.

7d ⟨*Print status report* 7d⟩≡ (6b)

```
double megabytes = preimages_read * 1.1444091796875e-05;
double rate = megabytes / (MPI_Wtime() - start);
printf("\rPreimages read: %d (%.1f Mb, %.1f Mb/s)", preimages_read, megabytes, rate);
fflush(stdout);
```

3.3 Hashing and Dispatching the Preimages

Now come the mapper threads. They split input blocks into $2^{\texttt{bits}}$ output buffers. When an output buffer is full, it is flushed to the writer.

8a ⟨*Mapper* 8a⟩≡ (6a)

```
int id = rank - 2;
⟨Initialize output buffers 8b⟩
u32 n_processed = 0, n_invalid = 0;
while (1) {
        ⟨Request preimages from reader; if EOF, then break 9a⟩
        for (int i = 0; i < n_preimages; i++) {
                ⟨Compute the hash; if invalid then continue 9b⟩
                ⟨Push the dictionnary entry to the output buffer slot 9c⟩
                if (output_size[slot] == WRITER_BUFFER_SIZE) {
                        ⟨flush output[slot] to the writer 9d⟩
                }
        }
}
⟨Flush all buffers and send EOF message to the writer 10a⟩
printf("Mapper %d finished. %d dictionnary entries transmitted. %d invalid.\n", id, n_processe
```

Dealing with the buffer is as simple as in the reader. So is communication with the reader.

8b ⟨*Initialize output buffers* 8b⟩≡ (8a)

```
struct dict_t * output[n_slots];
u32 output_size[n_slots];
for (u32 i = 0; i < n_slots; i++) {
        output_size[i] = 0;
        output[i] = malloc(WRITER_BUFFER_SIZE * sizeof(struct dict_t));
        if (output[i] == NULL)
                err(1, "cannot alloc mapper output buffer");
}
```

When receiving a block from the reader, we only know an upper bound on its size. We must then query its actual size from MPI.

9a ⟨*Request* `preimages` *from reader; if* `EOF`*, then* `break` 9a⟩≡ (8a)

```
struct preimage_t preimages[READER_BUFFER_SIZE];
MPI_Status status;
MPI_Send(NULL, 0, MPI_INT, 0, READER_REQUEST_TAG, MPI_COMM_WORLD);
MPI_Recv(preimages, READER_BUFFER_SIZE, PreimageType, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
if (status.MPI_TAG == EOF_TAG)
        break;
i32 n_preimages;
MPI_Get_count(&status, PreimageType, &n_preimages);
```

A priori, the nonces are valid at difficulty 33. This means that the top 33 bits of the hash must be zero. This is checked by `compute_full_hash`.

9b ⟨*Compute the hash; if invalid then* `continue` 9b⟩≡ (8a)

```
u32 full_hash[8];
if (!compute_full_hash(kind, preimages + i, full_hash)) {
        n_invalid++;
        continue;
}
```

9c ⟨*Push the dictionnary entry to the output buffer* `slot` 9c⟩≡ (8a)

```
u64 x = extract_partial_hash(full_hash);
u32 slot = extract_partitioning_key(bits, full_hash);
output[slot][output_size[slot]].hash = x;
output[slot][output_size[slot]].preimage.counter = preimages[i].counter;
output[slot][output_size[slot]].preimage.nonce = preimages[i].nonce;
output_size[slot] += 1;
```

To push the data to the writer, we first send `slot` and then only the actual data.

9d ⟨*flush* `output[slot]` *to the writer* 9d⟩≡ (8a 10a)

```
MPI_Send(&slot, 1, MPI_INT, 1, KEY_TAG, MPI_COMM_WORLD);
MPI_Send(output[slot], output_size[slot], DictType, 1, HASH_BLOCK_TAG, MPI_COMM_WORLD);
n_processed += output_size[slot];
output_size[slot] = 0;
```

When we receive the `EOF` mark from the reader, we must flush incomplete output buffers to the writer.

10a ⟨*Flush all buffers and send* `EOF` *message to the writer* 10a⟩≡ (8a)
```
for (u32 slot = 0; slot < n_slots; slot++) {
        ⟨flush output[slot] to the writer 9d⟩
}
MPI_Send(NULL, 0, MPI_INT, 1, EOF_TAG, MPI_COMM_WORLD);
```

3.4 Writing sub-Dictionnaries to Disk

It remains to describe the writer. The writer sequentialize writes, and deals with all the files. We just open the 2^k files simultaneously. For somewhat largish value of k, this may hit the OS restrictions. On my linux laptop, the default limit is 1024. Using `ulimit -n 2048` fixes the problem.

10b ⟨*Writer* 10b⟩≡ (6a)
```
u32 n_eof = 0;
FILE * f[n_slots];
⟨Open all output files 10c⟩
printf("Writer ready\n");
while (n_eof < n_mapper) {
        ⟨Receive block from a mapper; if EOF then continue 11a⟩
        ⟨Write block to the correct file 11b⟩
}
⟨Close all files 11c⟩
printf("Writer done.\n");
```

We derive the output filename from the input filename, the (optional) output directory and its `slot`. The output files are opened in write mode, so they are truncated to size zero.

10c ⟨*Open all output files* 10c⟩≡ (10b)
```
for (u32 i = 0; i < n_slots; i++) {
        char out_filename[255];
        char *input_base = basename(in_filename);
        sprintf(out_filename, "%s/%03x/%s.unsorted", output_dir, i, input_base);
        f[i] = fopen(out_filename, "w");
        if (f[i] == NULL)
                err(1, "[writer] Cannot open %s for writing", out_filename);
}
```

When receiving a block from a mapper, we must distinguish between regular and EOF messages. We must also observe the size of the hash block.

11a ⟨*Receive* `block` *from a mapper; if* `EOF` *then* `continue` 11a⟩≡ (10b)

```
MPI_Status status;
u32 slot;
struct dict_t block[WRITER_BUFFER_SIZE];
MPI_Recv(&slot, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
if (status.MPI_TAG == EOF_TAG) {
        n_eof++;
        continue;
}
MPI_Recv(block, WRITER_BUFFER_SIZE, DictType, status.MPI_SOURCE,
                                    HASH_BLOCK_TAG, MPI_COMM_WORLD, &status);
i32 n_entries;
MPI_Get_count(&status, DictType, &n_entries);
```

11b ⟨*Write* `block` *to the correct file* 11b⟩≡ (10b)

```
size_t tmp = fwrite(block, sizeof(struct dict_t), n_entries, f[slot]);
if (tmp != (size_t) n_entries)
        err(1, "fwrite writer (file %03x): %zd vs %d", slot, tmp, n_entries);
```

11c ⟨*Close all files* 11c⟩≡ (10b)

```
for (u32 i = 0; i < n_slots; i++)
        if (fclose(f[i]))
                err(1, "fclose writer file %03x", i);
```

Chapter 4

Dictionnary Checker

This program checks that a **sub-dictionnary file** is correct (i.e. the preimages are valid and the hash match). It will also detect if it is sorted by increasing hashes, with or without duplicates.

This program takes the name of a single dictionnary file on the command-line. Its kind and partition key are inferred from its name.

1a ⟨ * 1a⟩≡
⟨*Header files to include* 1b⟩
⟨*The main program* 2a⟩

We need the usual standard headers.

1b ⟨*Header files to include* 1b⟩≡ (1a)

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <err.h>

#include "preprocessing.h"
#include "hasher.h"
```

2a ⟨*The main program* 2a⟩≡ (1a)

```
int main(int argc, char **argv)
{
        ⟨Process the command line 2b⟩
        ⟨Open input file and allocate buffer 3a⟩
        struct dict_t prev = {0, {0, 0}};
        u32 processed = 0, size = 0;
        bool in_order = true;
        u32 duplicates = 0, collisions = 0;
        while (!feof(f)) {
                ⟨Fill buffer from input file 3b⟩
                ⟨Verify buffer 3c⟩
        }
        processed += size;
        ⟨Report and exit 4⟩
}
```

The only possible command-line arguments are the size k of the partioning key and the name of the dictionnary file to check.

2b ⟨*Process the command line* 2b⟩≡ (2a)

```
struct option longopts[2] = {
        {"partitioning-bits", required_argument, NULL, 'b'},
        {NULL, 0, NULL, 0}
};
int bits = -1;
signed char ch;
while ((ch = getopt_long(argc, argv, "", longopts, NULL)) != -1) {
        switch (ch) {
        case 'b':
                bits = atoi(optarg);
                break;
        default:
                errx(1, "Unknown option\n");
        }
}
if (bits == -1)
        errx(1, "missing required option --partitioning-bits");
if (optind != argc - 1)
        errx(1, "missing (or extra) filenames");
char *in_filename = argv[optind];
enum kind_t kind = file_get_kind(in_filename);
u32 partition = file_get_partition(in_filename);
printf("Input has kind %d and partition key %03x.\n", kind, partition);
```

We use a simple buffering scheme to read the input file. `processed` counts the number of items in previously processed buffers. `size` is the size of the current buffer.

3a ⟨*Open input file and allocate* `buffer` 3a⟩≡ (2a)

```
FILE *f = fopen(in_filename, "r");
if (f == NULL)
        err(1, "cannot open %s for reading", in_filename);
static const u32 BUFFER_SIZE = 131072;
struct dict_t *buffer = malloc(BUFFER_SIZE * sizeof(*buffer));
if (buffer == NULL)
        err(1, "cannot allocate buffer");
```

3b ⟨*Fill* `buffer` *from input file* 3b⟩≡ (2a)

```
processed += size;
size = fread(buffer, sizeof(*buffer), BUFFER_SIZE, f);
if (ferror(f))
        err(1, "fread failed");
```

`slot` is -1 until the partitioning key has been identified. It shall not change afterwards.

3c ⟨*Verify* `buffer` 3c⟩≡ (2a)

```
for (u32 i = 0; i < size; i++) {
        u32 full_hash[8];
        if (!compute_full_hash(kind, &buffer[i].preimage, full_hash))
                errx(1, "invalid preimage %d", processed + i);
        if (buffer[i].hash != extract_partial_hash(full_hash))
                errx(1, "partial hash mismatch %d", processed + i);
        if (partition != extract_partitioning_key(bits, full_hash))
                errx(1, "partitioning key changed");
        if (prev.preimage.counter == buffer[i].preimage.counter
                  && prev.preimage.nonce == buffer[i].preimage.nonce)
                duplicates++;
        if (prev.hash == buffer[i].hash)
                collisions++;
        if (prev.hash > buffer[i].hash)
                in_order = false;
        prev = buffer[i];
}
```

4 ⟨*Report and exit* 4⟩≡ (2a)

```
printf("%d items processed.\n", processed);
printf("In order: %d\n", in_order);
printf("Hash collisions: %d\n", collisions - duplicates);
printf("Duplicate preimages: %d\n", duplicates);
fclose(f);
exit(EXIT_SUCCESS);
```

Chapter 4

Sorter

This program sorts split dictionnary files, which are sequences of contiguous `struct dict_t` records. It reads a dictionnary file, sorts it by hash, and rewrites it.

1a ⟨ * 1a⟩≡
⟨*Header files to include* 1b⟩
⟨*Auxiliary functions* 3a⟩
⟨*The main program* 2a⟩

We need the usual standard headers.

1b ⟨*Header files to include* 1b⟩≡ (1a)

```
#define _XOPEN_SOURCE 500   /* strdup */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "preprocessing.h"
```

2a ⟨*The main program* 2a⟩≡ (1a)

```
int main(int argc, char **argv)
{
        ⟨Process the command line 2b⟩
        ⟨Load the dictionnary in dictionnary 2c⟩
        ⟨Sort dictionnary by increasing hash 3b⟩
        ⟨Write dictionnary to the .sorted file 3c⟩
        exit(EXIT_SUCCESS);
}
```

The only accepted command-line argument is the name of file to sort.

2b ⟨*Process the command line* 2b⟩≡ (2a)

```
if (argc < 2)
        errx(1, "missing argument: FILENAME");
char *in_filename = argv[1];
```

To read the dictionnaty file in memory, we first query its size, then allocate memory accordingly, then finally read it in the allocated space.

2c ⟨*Load the dictionnary in* dictionnary 2c⟩≡ (2a)

```
struct stat infos;
if (stat(in_filename, &infos))
        err(1, "fstat");
struct dict_t *dictionnary = malloc(infos.st_size);
if (dictionnary == NULL)
        err(1, "failed to allocate memory");
FILE *f_in = fopen(in_filename, "r");
if (f_in == NULL)
        err(1, "fopen failed");
size_t check = fread(dictionnary, 1, infos.st_size, f_in);
if ((check != (size_t) infos.st_size) || ferror(f_in))
        err(1, "fread : read %zd, expected %zd", check, infos.st_size);
if (fclose(f_in))
        err(1, "fclose %s", in_filename);
```

To actually perform the sort, we use the standard `qsort` function. It is not the fastest possible solution, but it is the simplest. We have to provide a comparison function for `qsort` and extra care has to be taken, because we cannot just substract the `u64`. This is an extra source of inefficiency.

3a ⟨*Auxiliary functions* 3a⟩≡ (1a)

```
int cmp(const void *a_, const void *b_)
{
        struct dict_t *a = (struct dict_t *) a_;
        struct dict_t *b = (struct dict_t *) b_;
        return (a->hash > b->hash) - (a->hash < b->hash);
}
```

3b ⟨*Sort* `dictionnary` *by increasing hash* 3b⟩≡ (2a)

```
int n_entries = infos.st_size / sizeof(*dictionnary);
qsort(dictionnary, n_entries, sizeof(*dictionnary), cmp);
```

Once the dictionnary has been sorted in memory, it has to be written back down to the file system. To avoid erasing the original file, we write it down to *another* file.

3c ⟨*Write* `dictionnary` *to the* `.sorted` *file* 3c⟩≡ (2a)

```
char *out_filename = strdup(in_filename);
memcpy(out_filename + strlen(out_filename) - 8, "sorted", 7);
FILE *f_out = fopen(out_filename, "w");
if (f_out == NULL)
        err(1, "cannot create output file %s", out_filename);
check = fwrite(dictionnary, sizeof(*dictionnary), n_entries, f_out);
if (check != (size_t) n_entries)
        err(1, "fwrite inconsistensy %zd vs %d", check, n_entries);
if (fclose(f_out))
        err(1, "fclose %s", out_filename);
```

Chapter 5

Merger

This program takes sub-dictionnary files and writes down a **hash file** (i.e. a sequence of `u64`) in ascending order, without duplicates. Optionnaly, the output can be randomly permuted.

1a ⟨ * 1a⟩≡
⟨*Header files to include* 1b⟩
⟨*Auxiliary functions* 2a⟩
⟨*The main program* 3a⟩

We need the usual standard headers.

1b ⟨*Header files to include* 1b⟩≡ (1a)

```
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <string.h>
#include <err.h>
#include <getopt.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/time.h>
#include <assert.h>

#include "preprocessing.h"
```

We use a small function to measure the passage of time accurately.

2a ⟨*Auxiliary functions* 2a⟩≡ (1a) 2b▷

```
double wtime()
{
        struct timeval ts;
        gettimeofday(&ts, NULL);
        return (double)ts.tv_sec + ts.tv_usec / 1E6;
}
```

2b ⟨*Auxiliary functions* 2a⟩+≡ (1a) ◁2a 4b▷

```
void * load_dict(const char *filename, u64 *size_)
{
        struct stat infos;
        if (stat(filename, &infos))
                err(1, "fstat failed on %s", filename);
        u64 size = infos.st_size;
        assert(size % sizeof(struct dict_t) == 0);
        u64 *content = malloc(size);
        if (content == NULL)
                err(1, "failed to allocate memory");
        FILE *f = fopen(filename, "r");
        if (f == NULL)
                err(1, "fopen failed (%s)", filename);
        u64 check = fread(content, 1, size, f);
        if (check != size)
                errx(1, "incomplete read %s", filename);
        fclose(f);
        *size_ = size / sizeof(struct dict_t);
        return content;
}
```

3a ⟨*The main program* 3a⟩≡ (1a)

```
int main(int argc, char **argv)
{
        ⟨Process the command line 3b⟩
        ⟨Load all input files 4a⟩
        ⟨Sort input 4c⟩
        ⟨Deduplicate to auxiliary array 5a⟩
        if (randomize) {
                ⟨Randomly permute output 5b⟩
        }
        ⟨Dump output 6⟩
        exit(EXIT_SUCCESS);
}
```

The name of input and output files are given on the command-line. Let us deal with the technicalities first.

3b ⟨*Process the command line* 3b⟩≡ (3a)

```
struct option longopts[3] = {
        {"output", required_argument, NULL, 'o'},
        {"randomize", no_argument, NULL, 'r'},
        {NULL, 0, NULL, 0}
};
char *out_filename = NULL;
bool randomize = false;
signed char ch;
while ((ch = getopt_long(argc, argv, "", longopts, NULL)) != -1) {
        switch (ch) {
        case 'o':
                out_filename = optarg;
                break;
        case 'r':
                randomize = true;
                break;
        default:
                errx(1, "Unknown option\n");
        }
}
if (out_filename == NULL)
        errx(1, "missing --output FILE");
if (optind >= argc)
        errx(1, "missing input filenames");
u32 P = argc - optind;
char **in_filenames = argv + optind;
```

4a ⟨*Load all input files* 4a⟩≡ (3a)

```
struct dict_t *dict[P];
u64 size[P];
u64 total_size = 0;
for (u32 i = 0; i < P; i++) {
        printf("* Loading %s\n", in_filenames[i]);
        dict[i] = load_dict(in_filenames[i], &size[i]);
        total_size += size[i];
}
struct dict_t *IN = malloc(total_size * sizeof(*IN));
total_size = 0;
for (u32 i = 0; i < P; i++) {
        memcpy(IN + total_size, dict[i], size[i] * sizeof(*IN));
        total_size += size[i];
        free(dict[i]);
}
```

We use a cheap strategy: sort, then deduplicate.

4b ⟨*Auxiliary functions* 2a⟩+≡ (1a) ◁2b

```
int cmp(const void *a_, const void *b_)
{
        struct dict_t *a = (struct dict_t *) a_;
        struct dict_t *b = (struct dict_t *) b_;
        return (a->hash > b->hash) - (a->hash < b->hash);
}
```

4c ⟨*Sort input* 4c⟩≡ (3a)

```
printf("--- Sorting\n");
qsort(IN, total_size, sizeof(*IN), cmp);
```

Duplicates are detected on output. We simply keep the previous value actually written and discard an eventual output if it is equal to this value.

5a ⟨*Deduplicate to auxiliary array* 5a⟩≡ (3a)

```
u64 *OUT = malloc(total_size * sizeof(*OUT));
u64 n = 0;
int duplicates = 0;
int collisions = 0;
struct dict_t prev;
prev.hash = 0;
prev.preimage.nonce = 0;
prev.preimage.counter = 0;
for (u64 i = 0; i < total_size; i++) {
        if (IN[i].hash != prev.hash) {
                OUT[n++] = IN[i].hash;
        } else {
                /* diagnose duplicate */
                if ((IN[i].preimage.counter == prev.preimage.counter)
                                        && (IN[i].preimage.nonce == prev.preimage.nonce))
                        duplicates++;
                else
                        collisions++;
        }
        prev = IN[i];
}
printf("-- item processed = %" PRId64 ", hash collisions=%d, duplicate=%d\n", n, collisions, duplicat
```

5b ⟨*Randomly permute output* 5b⟩≡ (3a)

```
for (u32 i = 0; i < n - 1; i++) {
        u32 j = i + (OUT[i] % (n - i));
        u64 x = OUT[i];
        OUT[i] = OUT[j];
        OUT[j] = x;
}
```

6 ⟨*Dump output* 6⟩≡ (3a)

```
printf("-- Saving\n");
FILE *f_out = fopen(out_filename, "w");
if (f_out == NULL)
        err(1, "cannot open %s for writing", out_filename);
size_t check = fwrite(OUT, sizeof(*OUT), n, f_out);
if (check != n)
        err(1, "incomplete fwrite");
if (fclose(f_out))
        err(1, "fclose on %s", out_filename);
```

Chapter 6

Hash File Checker

This program checks that a **hash file** is actually sorted and without duplicates.

1 ⟨ * 1⟩≡
⟨*Header files to include* 2⟩
⟨*The main program* 3⟩

We need the usual standard headers.

2 ⟨*Header files to include* 2⟩≡ (1)
```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

#include "preprocessing.h"
```

3 ⟨*The main program* 3⟩≡ (1)
```
int main(int argc, char **argv)
{
        ⟨Open input file and allocate buffer 4⟩
        u64 prev = 0;
        u32 processed = 0, size = 0;
        while (!feof(f)) {
                ⟨Fill buffer from input file 5⟩
                ⟨Check that buffer is in-order and duplicate-free 6⟩
        }
        fclose(f);
        exit(EXIT_SUCCESS);
}
```

4 ⟨*Open input file and allocate* `buffer` 4⟩≡ (3)

```
if (argc < 2)
        errx(1, "missing input filename");
FILE *f = fopen(argv[1], "r");
if (f == NULL)
        err(1, "cannot open %s for reading", argv[1]);
static const u32 BUFFER_SIZE = 131072;
u64 *buffer = malloc(BUFFER_SIZE * sizeof(*buffer));
if (buffer == NULL)
        err(1, "cannot allocate buffer");
```

`processed` counts the number of hashes in previously processed buffers. `size` is the size of the current buffer.

5 ⟨*Fill* `buffer` *from input file* 5⟩≡ (3)

```
processed += size;
size = fread(buffer, sizeof(*buffer), BUFFER_SIZE, f);
if (ferror(f))
        err(1, "fread failed");
```

6 ⟨*Check that* `buffer` *is in-order and duplicate-free* 6⟩≡ (3)

```
for (u32 i = 0; i < size; i++) {
        if (prev >= buffer[i])
                errx(1, "F[%d] (%016" PRIx64 ") >= F[%d] (%016" PRIx64 ")",
                        processed + (i-1), prev, processed + i, buffer[i]);
        prev = buffer[i];
}
```

Chapter 7

Forger

This program forges three hash file with known solutions. This is helpful to ensure that the solving code is correct (it must find the solutions...). The files are written to the current directory.

1a ⟨ * 1a⟩≡
⟨*Header files to include* 1b⟩
⟨*Auxiliary functions* 3b⟩
⟨*The main program* 2a⟩

We need the usual standard headers.

1b ⟨*Header files to include* 1b⟩≡ (1a)

```
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <err.h>
#include "hasher.h"
```

2a ⟨*The main program* 2a⟩≡ (1a)

```
int main(int argc, char **argv)
{
        ⟨Process the command line 2b⟩
        ⟨Generate random lists 3a⟩
        ⟨Sort B and C 3c⟩
        ⟨"Fix" solutions and display them 4a⟩
        ⟨Sort A 3d⟩
        ⟨Display collision indexes 4b⟩
        ⟨Write hash files 4c⟩
        exit(EXIT_SUCCESS);
}
```

The average size of lists to generate is given on the command-line.

2b ⟨*Process the command line* 2b⟩≡ (2a)

```
if (argc < 2)
        errx(1, "missing argument N");
u32 avg_size = atoi(argv[1]);
```

To generate random lists, we use SHA256 in counter mode. The sizes of the list may fluctuate.

3a ⟨*Generate random lists* 3a⟩≡ (2a)

```
u32 N[3];
u64 *H[3];
srand48(1337);
for (u32 k = 0; k < 3; k++) {
        N[k] = avg_size / 2 + (((u32) mrand48()) % avg_size);
        printf("H[%d] has size %d\n", k, N[k]);
        H[k] = malloc(N[k] * sizeof(u64));
        if (H[k] == NULL)
                err(1, "cannot allocate hashes");
        struct preimage_t pre;
        pre.nonce = 0;
        pre.counter = 0;
        u32 ptr = 4;
        u32 hash[8];
        u64 *randomness = (u64 *) hash;
        for (u32 n = 0; n < N[k]; n++) {
                if (ptr == 4) {
                        compute_full_hash(k, &pre, hash);
                        pre.counter++;
                        ptr = 0;
                }
                H[k][n] = randomness[ptr++];
        }
}
```

To sort the lists, we again use `qsort` with a comparator.

3b ⟨*Auxiliary functions* 3b⟩≡ (1a)

```
int cmp(const void *a_, const void *b_)
{
        const u64 *const a = (u64 *) a_;
        const u64 *const b = (u64 *) b_;
        return (*a > *b) - (*a < *b);
}
```

3c ⟨*Sort B and C* 3c⟩≡ (2a)

```
qsort(H[1], N[1], sizeof(u64), cmp);
qsort(H[2], N[2], sizeof(u64), cmp);
```

3d ⟨*Sort A* 3d⟩≡ (2a)

```
qsort(H[0], N[0], sizeof(u64), cmp);
```

To ensure that there are solutions we know, we force solutions at the very begining and at the very end of B and C. We also impose a random solution.

4a ⟨*"Fix" solutions and display them* 4a⟩≡ (2a)

```
u32 b = ((u32) mrand48()) % N[1];
u32 c = ((u32) mrand48()) % N[2];
H[0][0] = H[1][0] ^ H[2][0];
H[0][1] = H[1][N[1] - 1] ^ H[2][1];
H[0][2] = H[1][b] ^ H[2][c];
u64 x[3] = { H[0][0], H[0][1], H[0][2] };
printf("%016" PRIx64 " ^ %016" PRIx64 " ^ %016" PRIx64 " = 0\n",
        x[0], H[1][0], H[2][0]);
printf("%016" PRIx64 " ^ %016" PRIx64 " ^ %016" PRIx64 " = 0\n",
        x[1], H[1][N[1] - 1], H[2][N[2] - 1]);
printf("%016" PRIx64 " ^ %016" PRIx64 " ^ %016" PRIx64 " = 0\n",
        x[2], H[1][b], H[2][c]);
```

After A has been sorted, we must scan it to find the values of $x[i]$.

4b ⟨*Display collision indexes* 4b⟩≡ (2a)

```
u32 y[3];
for (u32 k = 0; k < 3; k++) {
        for (u32 d = 0; d < N[0]; d++)
                if (H[0][d] == x[k]) {
                        y[k] = d;
                        break;
                }
}
printf("H[0][%08x] ^ H[1][%08x] ^ H[2][%08x] = 0\n", y[0], 0, 0);
printf("H[0][%08x] ^ H[1][%08x] ^ H[2][%08x] = 0\n", y[1], N[1] - 1, N[2] - 1);
printf("H[0][%08x] ^ H[1][%08x] ^ H[2][%08x] = 0\n", y[2], b, c);
```

finally, we write down the hashes in fixed files.

4c ⟨*Write hash files* 4c⟩≡ (2a)

```
char *names[3] = {"foo.000", "bar.000", "foobar.000"};
for (u32 k = 0; k < 3; k++) {
        FILE *f = fopen(names[k], "w");
        if (f == NULL)
                err(1, "cannot open %s for writing", names[k]);
        size_t check = fwrite(H[k], sizeof(uint64_t), N[k], f);
        if (check != N[k])
                errx(1, "incomplete write");
        fclose(f);
}
```