• Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

• The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

• The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS30048/vm

• The purpose of the worksheet is to provide a) a tutorial-style introduction to selected technologies or concepts, and/or b) a means to explore them via hands-on tasks and challenges. Note that the worksheet is not assessed *at all*: if you are confident that you already understand the content, there is no problem with nor penalty for totally ignoring it.

• Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS30048 lab. worksheet #3

Before you start work, download (and, if need be, unarchive[a]) the file

https://assets.phoo.org/COMS30048_2025_TB-2/csdsp/sheet/lab-03.tar.gz

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-03.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive→Open` menu item to open `lab-03.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-03.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

1. Introduction

Efficient implementation of modular multi-precision integer arithmetic, i.e., arithmetic in $\mathbb{Z}_N$ for some N, is a fundamental requirement for many asymmetric cryptosystems. Concrete examples include RSA, where $N = p \cdot q$ is a product of two primes, and ElGamal, and ECC, where N is *itself* a prime (so formally we instead write $\mathbb{F}_p$, where $p = N$). As a result, cryptographic software libraries will usually include an entire suite of functionality, i.e., data structures and algorithms, which delivers an implementation of the required arithmetic. GNU Multiple Precision Arithmetic Library (GMP) is arguably the de facto standard example of such a library, which offers an highly optimised and so efficient option across a range of platforms.

The goal of this worksheet is to use GMP as a means of exploring 1) the theory underlying multi-precision integer arithmetic, and 2) the practical implementation of that theory, and application within the context of RSA. The content offers a step-by-step introduction to the library, and, although it cannot and will not cover *every* feature, it *should* equip you with a starting point that can be extended to suit.

2. GMP in theory: improve your understanding

Keeping in mind that it covers the latest version (and so may not exactly match the version you are using), the GMP documentation [3] available online at

http://gmplib.org/manual

acts as a definitive reference. Read [3, Section 3] before continuing: this summarises a range of information about 1) *how* GMP is used and implemented internally, and, importantly, 2) *why* GMP makes specific decisions within said implementation. Beyond this, the following presents selected, pertinent points by relating them to their introduction within the lecture slot(s):

- `mp_limb_t` is a type used to represent base-b digits (or limbs); the value of b depends on the GMP installation (since it can support the best choice for the platform), but the constant `GMP_LIMB_BITS` tells you the w for which $b = 2^w$.
- The data structures and algorithms used by GMP are similar to those presented in the lecture slot(s):
  - Lower-level functions starting with the prefix `mpn` are used to operate on representations of unsigned integers (i.e., elements of $\mathbb{N}$). Said representations could be described as raw arrays of limbs; the array length must be managed manually by the programmer [3, Section 8] offers a complete overview.
  - Higher-level functions starting with the prefix `mpz` are used to operate on representations of signed integers (i.e., elements of $\mathbb{Z}$). Said representations could be described as structured arrays of limbs, encapsulated in the `mpz_t`; the array length is managed automatically by GMP. [3, Section 5] offers a complete overview.
- *Unlike* the lecture slot(s), however, where the `mpz_t` data structure presented included an array of fixed size, instances of the GMP `mpz_t` data structure allow said array to grow dynamically. Although this feature is managed automatically by GMP, it demands the programmer carefully use `mpz_init` to initialise each instance of `mpz_t` before (the first) use, the use `mpz_clear` to finalise each instance of `mpz_t` after (the last) use.
- The `mpz_size` can be used to compute the number of limbs used by an instance of `mpz_t`; the i-th such limb can be accessed using `mpz_getlimbn`. You *can* also access the limbs directly, although this needs care: if you need to do so, a cleaner though less efficient approach might be to use `mpz_import` to convert an array of limbs into an instance of `mpz_t`, or `mpz_export` to convert an instance of `mpz_t` into an array of limbs.

```
8  #ifndef __HELLOWORLD_GMP_H
9  #define __HELLOWORLD_GMP_H
10
11 #include  <stdio.h>
12 #include <stdlib.h>
13
14 #include    <gmp.h>
15
16 #endif
```

(a) *${ARCHIVE}/helloworld_gmp.h*

```
8  #ifndef __HELLOWORLD_LIBC_H
9  #define __HELLOWORLD_LIBC_H
10
11 #include  <stdio.h>
12 #include <stdlib.h>
13
14
15
16 #endif
```

(b) *${ARCHIVE}/helloworld_libc.h*

```
8  #include "helloworld_gmp.h"
9
10 int main( int argc, char* argv[] ) {
11   mpz_t r, x, y;
12
13   mpz_init( r );
14   mpz_init( x );
15   mpz_init( y );
16
17   if( 1 != gmp_scanf( "%Zd",  x ) ) {
18     abort();
19   }
20   if( 1 != gmp_scanf( "%Zd",  y ) ) {
21     abort();
22   }
23
24   mpz_add( r, x, y );
25
26   gmp_printf( "%Zd\n", r );
27
28   mpz_clear( r );
29   mpz_clear( x );
30   mpz_clear( y );
31
32   return 0;
33 }
```

(c) *${ARCHIVE}/helloworld_gmp.c*

```
8  #include "helloworld_libc.h"
9
10 int main( int argc, char* argv[] ) {
11   int   r, x, y;
12
13
14
15
16
17   if( 1 !=     scanf(  "%d", &x ) ) {
18     abort();
19   }
20   if( 1 !=     scanf(  "%d", &y ) ) {
21     abort();
22   }
23
24   r = x + y;
25
26      printf(  "%d\n", r );
27
28
29
30
31
32   return 0;
33 }
```

(d) *${ARCHIVE}/helloworld_libc.c*

Figure 1: *Source code for a GMP-based "hello world" program (left), and the libc-based equivalent (right).*

- GMP includes a variety of functions you can use to read and write input and output. Perhaps the most useful are `gmp_scanf` and `gmp_printf`, which act like the C `scanf` and `printf` functions except they support a range of extended data types and hence format specifiers. [3, Sections 10 and 11] offer a complete overview.

3. GMP in practice: use your understanding

The goal of this Section is to translate theory into practice: it attempts to offer a (limited) introduction to how 1) how GMP is used (from an external perspective), plus 2) how GMP is implemented (from an internal perspective), via two (practical) examples. Both aspects support Section 4, which presents a set of associated tasks.

3.1. Example #1: a GMP-based "hello world"

Figure 1 includes minimal GMP-based program: the goal is to 1) demonstrate central differences versus a non-GMP alternative written in vanilla C, and 2) act as a starting point (or template) for later, more complicated tasks. Rather than write "hello world"[1] to the terminal as is traditional, it instead computes a basic operation by initially reading two integers `x` and `y` from `stdin`, then writing their sum `r = x + y` to `stdout`. The Figure in fact captures *two* versions of the program side-by-side:

1. the left-hand version uses GMP to support multi-precision `r`, `x` and `y`, whereas
2. the right-hand version represents a non-GMP analogue that only supports single-precision versions of the same variables (using vanilla C).

To use the left-hand version, clearly Figure 1a must include `gmp.h`. Beyond this, we focus on differences in the two source files captured in Figure 1c and Figure 1d (the latter of which has been artificially spaced and indented so as to highlight said differences):

[1] http://en.wikipedia.org/wiki/"Hello,_World!"_program

- In the left-hand version (Line #11), the type of `r`, `x` and `y` is `mpz_t` (meaning multi-precision integer); in the right-hand version their type is `int` (meaning single-precision, typically 32- or 64-bit integer).
- In the left-hand version (Lines #13 to #15), before we start using `r`, `x` and `y` they *must* be initialised by using `mpz_init`; in the right-hand version this clearly is unnecessary.
- In the left-hand version (Lines #17 to #22), we read `x` and `y` from `stdin` by using `gmp_scanf` with the format string `%Zd` (to specify multi-precision, decimal format); in the right-hand version we use `scanf` as normal.
- In the left-hand version (Line #24), we add `x` and `y` to form `r` by using the `mpz_add` function; in the right-hand version we use the + operator as normal.
- In the left-hand version (Line #26), we write `r` to `stdout` by using `gmp_printf` with the format string `%Zd` (to specify multi-precision, decimal format); in the right-hand version we use `printf` as normal.
- In the left-hand version (Lines #28 to #30), after we finish using `r`, `x` and `y` they *must* be finalised by using `mpz_clear`; in the right-hand version this clearly is unnecessary.

Assuming there is an installation[2] of GMP available, compilation should be trivial: you simply add `-lgmp` to the end of your normal GCC-based build command.

3.2. Example #2: dissecting a GMP `mpz_t` instance

Recall from the lecture slot(s) that to represent a multi-precision integer x, we use a base-b expansion

$$\begin{aligned} \hat{x} &= \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{n-1} \rangle \\ &\mapsto x \\ &= \pm \sum_{i=0}^{n-1} \hat{x}_i \cdot b^i \end{aligned}$$

where each $\hat{x}_i$ is one of n digits taken from the digit set $X = \{0, \ldots, b-1\}$. The choice of b is dictated by the platform: given a processor whose word size is $w = 64$ bits for example, we opt for $b = 2^w = 2^{64}$ to ensure each $\hat{x}_i$ is word- and hence register-sized, and thus can be operated on by using native, hardware supported arithmetic.

How does this theory translate into practice? Even given the lecture slot(s), you might not believe GMP *actually* does this! We can demonstrate it does, however, by dissecting an instance of the `mpz_t` structure: by looking inside the structure, we can inspect the actual representation used by GMP and show it matches the above. Figure 2 shows two programs that do this in different ways. Both read a multi-precision integer `x` from `stdin`, then print each limb within the resulting representation. However,

1. the left-hand version uses the function `mpz_export` to export the limbs into an array called `t` (the various constants instruct it to use little-endian ordering), while
2. the right-hand version *directly* inspects fields within `x`, voiding any sort of abstraction offered by GMP.

For this task, one *could* even consider a version half-way between the two, by using the functions `mpz_size` and `mpz_getlimbn` in place of direct access to `_mp_size` and `_mp_d`. Either way, these versions both produce the same result: given the (decimal) input

$$x = 12345678901234567890123456789012345678901234567890$$

say, both form an `x` represented as a base-2^{64} sequence of limbs

$$\hat{x} = \langle 12446928571455179474, 11585827206506214328, 3 \rangle.$$

[2] GMP is a standard package on *most* distributions of Linux, but may require installation on Windows for example. It can often be useful to verify that you linked against the version of GMP you intended to: note that `gmp_version` captures a human-readable version string, so `printf( "%sn", gmp_version );` will print the GMP version to `stdout`.

```
8  #ifndef __DISSECT_GMP_EXPORT_H
9  #define __DISSECT_GMP_EXPORT_H
10
11 #include  <stdio.h>
12 #include <stdlib.h>
13
14 #include    <gmp.h>
15
16 #endif
```

(a) *${ARCHIVE}/dissect_gmp_export.h*

```
8  #ifndef __DISSECT_GMP_STRUCT_H
9  #define __DISSECT_GMP_STRUCT_H
10
11 #include  <stdio.h>
12 #include <stdlib.h>
13
14 #include    <gmp.h>
15
16 #endif
```

(b) *${ARCHIVE}/dissect_gmp_struct.h*

```
8  #include "dissect_gmp_export.h"
9
10 int main( int argc, char* argv[] ) {
11   mpz_t x;
12
13   mpz_init( x );
14
15   if( 1 != gmp_scanf( "%Zd", x ) ) {
16     abort();
17   }
18
19   size_t n = mpz_size( x );
20
21   mp_limb_t t[ n ];
22
23   mpz_export( t, NULL, -1, sizeof( mp_limb_t ), -1, 0, x );
24
25   for( int i = 0; i < n; i++ ) {
26     if( i != 0 ) {
27       gmp_printf( "+" );
28     }
29
30     gmp_printf( "%llu*(2^(64))^(%d)", t[ i ], i );
31   }
32
33   gmp_printf( "\n" );
34
35   mpz_clear( x );
36
37   return 0;
38 }
```

(c) *${ARCHIVE}/dissect_gmp_export.c*

```
8  #include "dissect_gmp_struct.h"
9
10 int main( int argc, char* argv[] ) {
11   mpz_t x;
12
13   mpz_init( x );
14
15   if( 1 != gmp_scanf( "%Zd", x ) ) {
16     abort();
17   }
18
19   size_t n = abs( x->_mp_size );
20
21   mp_limb_t* t = x->_mp_d;
22
23
24
25   for( int i = 0; i < n; i++ ) {
26     if( i != 0 ) {
27       gmp_printf( "+" );
28     }
29
30     gmp_printf( "%llu*(2^(64))^(%d)", t[ i ], i );
31   }
32
33   gmp_printf( "\n" );
34
35   mpz_clear( x );
36
37   return 0;
38 }
```

(d) *${ARCHIVE}/dissect_gmp_struct.c*

Figure 2: *Source code for two GMP-based programs that illustrate how multi-precision integers are represented (as instances of `mpz_t`).*

Why does this make sense? Clearly, said representation means

$$\begin{aligned}
& & & & \langle 12446928571455179474, 11585827206506214328, 3\rangle_{(2^{64})} \\
\mapsto & & 12446928571455179474 & \cdot & (2^{64})^0 \\
+ & & 11585827206506214328 & \cdot & (2^{64})^1 \\
+ & & 3 & \cdot & (2^{64})^2 \\
\\
= & & 12446928571455179474 & \cdot & 1 \\
+ & & 11585827206506214328 & \cdot & 18446744073709551616 \\
+ & & 3 & \cdot & 340282366920938463463374607431768211456 \\
\\
= & & & & 12446928571455179474 \\
+ & & & & 213720789360641398609774928034474754048 \\
+ & & & & 102084710076281539039012382229530463436 8 \\
\\
= & & & & 123456789012345678901234567890123456789 0
\end{aligned}$$

so we *are* representing the correct value.

4. Some hands-on tasks and challenges

4.1. Build, execute, and experiment with example #1

Focusing on Section 3.1, and so Figure 1:

1. Fix the working directory:
   ```
   cd ${ARCHIVE}
   ```
2. Build the source code:
   ```
   make all
   ```
3. Execute the left- or right-hand version of the example via
   ```
   ./helloworld_gmp
   ```
   or
   ```
   ./helloworld_libc
   ```
   typing input into the terminal, and verifying that the associated output is as expected.
4. Perform some experiments with, and/or make alterations to, the example:
   (a) Currently the example computes the sum of `x` and `y`: investigate, and make use of some other arithmetic functions available in GMP, e.g., `mpz_sub` or `mpz_mul`.
   (b) Identify the limit(s) where the values of `x` and `y` will produce the correct result in the left-hand, GMP-based version but the incorrect result in the right-hand, libc-based version. Can you explain these limits in a precise way, framed against the representations for `x` and `y` used by the two versions?

4.2. Build, execute, and experiment with example #2

Focusing on Section 3.2, and so Figure 2:

1. Fix the working directory:
   ```
   cd ${ARCHIVE}
   ```
2. Build the source code:
   ```
   make all
   ```
3. Execute the left- or right-hand version of the example via
   ```
   ./dissect_gmp_export
   ```
   or
   ```
   ./dissect_gmp_struct
   ```
   typing input into the terminal, and verifying the associated output is as expected.

4. Perform some experiments with and/or make alterations to the example:
   (a) Investigate low(er)-level functions provided by GMP, which are prefixed with `mpn`: these operate directly on sequences of limbs. For example, can you reproduce the behaviour of the other, "hello world" example using `mpn_add`?
   (b) Consider `x`, the instance of `mpz_t` that uses the internal sequence of limbs `x->_mp_d` to represent an integer value; one could imagine implementing a high(er)-level function by directly manipulating said sequence of limbs, and thus the associated representation and value. For example, can you write a function (say `mpz_inc`) that increments any such `x` *without* using `mpn_add`, `mpz_add`, or similar?

4.3. Develop your own GMP-supported RSA implementation

1. Recall that, for a security parameter λ, RSA key generation can be summarised as follows:

   1 select random $\frac{\lambda}{2}$-bit primes p and q
   2 compute $N = p \cdot q$
   3 compute $\Phi(N) = (p-1) \cdot (q-1)$
   4 select random $e \in \mathbb{Z}^*_N$ such that $\gcd(e, \Phi(N)) = 1$
   5 compute $d = e^{-1} \pmod{\Phi(N)}$
   6 return public key (N, e) and private key (N, d)

   Given that

   - [3, Section 5.13] describes `mpz_urandomm` for generation of random integers (within a given range), and
   - [3, Section 5.9] includes `mpz_probab_prime_p` for primality testing, `mpz_gcd` for computation of GCD, and `mpz_invert` for computation of modular inversion,

   implement a function

   ```
   void rsa_keygen( mpz_t N, mpz_t e, mpz_t d, int lambda )
   ```

   that realises said key generation process.
2. Recall that modular exponentiation forms the basis for both RSA encryption, i.e.,

   $$c = m^e \pmod{N},$$

   and decryption, i.e.,

   $$m = c^d \pmod{N}.$$

   Given that

   - [3, Section 5.15] describes `mpz_sizeinbase` which can be used to compute the number of bits used within the representation of a given integer,
   - [3, Section 5.11] describes `mpz_tstbit` which can be used to inspect the i-th such bit,
   - [3, Section 5.7] describes `mpz_mod` for computation of modular reduction, such that a modular multiplication can be computed by composition of `mpz_mul` and `mpz_mod`,
   - [3, Section 5.7] describes `mpz_powm` for computation of modular exponentiation, which can be used as a known-good reference implementation to test against,

   implement a function

   ```
   void powm( mpz_t r, mpz_t x, mpz_t y, mpz_t N )
   ```

   that realises the 1Exp-L2R-Binary (or binary, left-to-right single-exponentiation) algorithm presented in the lecture slot(s), and thereby

   ```
   void rsa_enc( mpz_t c, mpz_t m, mpz_t e, mpz_t N )
   ```

   and

   ```
   void rsa_dec( mpz_t m, mpz_t c, mpz_t d, mpz_t N )
   ```

   for RSA encryption and decryption.

4.4. Extend your own GMP-supported RSA implementation

Explore options for improving the efficiency of your implementation, e.g.,

1. replace your implementations of `rsa_keygen` and `rsa_enc` with alternatives based on use of a "small" encryption exponent,

2. replace your implementations of `rsa_keygen` and `rsa_dec` with alternatives based on use of the CRT (see, e.g., [2]),
3. replace your implementation of `powm` with an alternative based on use of Montgomery multiplication (see, e.g., [1]),
4. replace your implementation of `powm` with an alternative based on use of pre-computation, e.g., the 1Exp-L2R-FixedWindow (or windowed, left-to-right single-exponentiation) algorithm presented in the lecture slot(s).

References

[1] P.L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (see p. 8).

[2] J-J. Quisquater and C. Couvreur. "Fast decipherment algorithm for RSA public-key cryptosystem". In: *IEE Electronics Letters* 18.21 (1982), pp. 905–907 (see p. 8).

[3] *The GNU Multiple Precision (GMP) Arithmetic Library*. URL: http://gmplib.org/manual (see pp. 2, 3, 7).