Security Aspects in Software Development
KU 2013/2014
"Too big to fail (C1)"

Daniel Hein Johannes Winter
Thomas Kastner
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8020 Graz, Austria
sicherheitsaspekte@iaik.tugraz.at

November 27, 2013

Contents

1 Introduction

Assignment title:	Too big to fail (C1)
Group size:	2 students
Start date:	November 27, 2013
Maximum score:	15 points (15% percent of final* mark)
HARD SUBMISSION DEADLINE:	**December 15, 2013 (23:59:59 CET)**

See Section 4 for details on the submission process.
See Section 5 for the subtasks and questions of this assignment.

* Bar the oral exam.

1.1 Motivation

Integer overflows are a serious security threat present in many runtime environments and programming languages. The impact of an integer overflow depends greatly on the context and location where the overflow occurred in the program, and the underlying programming language. In case of "safe" programming languages like Java and C# integer overflows often manifest themselves as spurious array index exceptions. In programs written in low-level programming languages like C and C++ which do not abstract raw memory and array access behind a safe memory model, integer overflows are often an initial stage for successful exploitation of buffer overflows.

According to the *CWE/Sans Top 25 Most Dangerous Software Errors* list[1], buffer overflows and variants of integer overflows are still amongst the top 25 software security problems. Classic buffer overflows even rank at place 3 of this list (immediately after SQL and OS command injections).

In this task we focus on secure implementation of a simple big integer library, called *tinybn*, in the C programming language. Implementing this library allows us to address integer overflows and buffer overflows from different points of view:

Firstly, the library needs to perform low-level operations on arrays, including memory allocation, and copying of data between internal and external data buffers. Functionality associated with such operations must be protected sufficiently against integer overflows in length calculations, off-by-one errors, and out of memory conditions.

Secondly, the big integer library necessarily has to work with well-defined overflow behavior of unsigned integers, to allow construction of multi-precision integer arithmetic operations from available single-precision operations.

Finally, the finished big integer library itself can be used as a building block for applications, which require potentially unlimited size integers. An illustrative example on how many X.509 certificate toolkits get this point wrong, is discussed in the paper by Kamniski et al.[3].

1.2 Objectives

The goal of this exercise is to build up basic skills in secure C programming and in testing of security critical C code. In particular this exercise is intended to train the ability:

- to understand the overflow behavior of unsigned integer arithmetic. In particular, this exercise is designed to train your ability to detect, and to repair problems related to integer overflows in C programs. In addition, you should understand, how integer overflows relate to other types of security problems, such as buffer overflows.
- to write robust code that can tolerate invalid and out of bounds inputs. As part of this exercise, you

should improve your skills in defensive programming. You should learn, how to apply dynamic and static testing tools to indentify, locate and repair bugs in security critical code. Additionally, this exercise aims to train your ability, to write test-cases, which properly cover normal, as well as corner-case behavior of your code.

1.3 Plagiarism

⚠ *Plagiarism will not be tolerated and any contributions containing copied code will automatically receive a negative rating. It is explicitly allowed to use material and code snippets shown during the lecture or the exercises.*

2 Background

Multi-precision arithmetic, that is artihmetic on large integers which are represented using more than one native machine word, is a key foundation for implementing public-key crypto-systems.

Let us consider the RSA crypto-system as an example: Both, the encryption and the decryption operation, require exponentiation, multiplication, and modular reduction of integers in the size range of several thousands (say 2048 or 4096) bits. General purpose computers simply do not provide instructions to compute with such large integers. They are tpyically limited to 32-bit and 64-bit operations.[1]

To compute with such large *multi-precision* integers, it is thus necessary to split computations in small *single-precision* operations, which can be handled natively by the processor.

☞ *There are many textbooks dealing with algorithms for multi-precision arithmetic. For this exercise, we rely on the* Handbook of Applied Cryptography (HAC) *[4] as standard reference for the algorithms.*

See the references at the end of this document for a link to the entire online version of the HAC. For this exercise you only need chapter 14, which is about efficient implementations. This chapter is available online at:
http://cacr.uwaterloo.ca/hac/about/chap14.pdf

2.1 Definitions

In the remainder of this document, we use the terms *single-precision* words, *double-precision* words and, *multi-precision* integers. Let us briefly define these terms for clarification:

Single-precision words are *unsigned* 32-bit quantities which can be handled natively by the C compiler and the processor. We assume that the compiler is at least able to generate code for addition, subtraction and comparison of single-precision integers.

The `mp_word_t` data type discussed in appendix A.1 defines a suitable C data-type for representing single-precision words in the `tinybn` library.

Double-precision words are *unsigned* 64-bit quantites, which can be handled natively by the C compiler and (maybe) by the processor. We assume that the compiler is at least able to generate code for addition of double-precision values, and for multiplication of two single-precision values with a double-precision result.

The `mp_long_t` data type discussed in appendix A.1 defines a suitable C data-type for representing double-precision words in the `tinybn` library.

Multi-precision integers are large integers, which are internally represented as arrays of single-precision integer “digits”. The (theoretically) supported maximum size of these arrays is only bounded by

[1] Special purpose instruction set extensions like SSE (x86) or NEON (ARM) are out of scope for this exercise.

the limits of the C `size_t` type (to count the number of array elements) and the available system memory.

The low-level functions in appendix A.1 allow basic arithmetic (addition, subtraction, multiplcation, comparison) of *unsigned* multi-precision integer, without considering memory allocation[2]).

The high-level functions in appendix A.2 provide a simple to use interface to big integers. The API interface supports positive and negative integers, as well as automatic support for memory allocation.

2.2 Word-level representation

The `tinybn` library assumes that the magnitude of multi-precision integers is internally represented by an array of unsigned single-precision words. For the high-level API discussed in A.2 a sign-and-magnitude representation is used.

All positive integers $0 \leq z < B^{n+1}$ can be uniquely expressed as a sum of $n+1$ base-B digits $a_0, ..., a_n$, where each of the a_i digits is in range $0 \leq a_i < B$ (cf. [4, Chapter 14.2]):

$$z = \sum_{i=0}^{n} a_i B^i = a_n B^n + a_{n-1} B^{n-1} + \ldots + a_1 B^1 + a_0 B^0 \tag{1}$$

The standard decimal number system can for example be represented by setting $B = 10$. For our big-integer library we chose B to be two to the power of the bit-width of a single-precision word ($B = 2^{32}$), and thus obtain:

$$z = \sum_{i=0}^{n} a_i (2^{32})^i = a_n (2^{32})^n + a_{n-1} (2^{32})^{n-1} + \ldots + a_1 2^{32} + a_0 \tag{2}$$

The a_i are the single-precision words in the word arrays used by the `tinybn` library.

2.3 Addition and subtraction

Addition (and subtraction) of multi-precision integers is accomplished, by adding (or subtracting) the individual words of the word-level array representation and by propagating the resulting carries (respectively borrows).

At the core of multi-precision addition and subtraction we have the corresponding single precision operations. When adding or subtracting two *unsigned* t-bit integers a and b (our two single-precision operands), arithmetic is effectively performed modulo 2^t.

In case of addition, an unsigned overflow causes the t-bit result to be less than the true sum. This property allows us to easily test if the addition overflowed, and thus if a carry needs to be propagated to the next digit.

Subtraction is a bit more interesting: As long as $0 \leq a_i \geq b_i < 2^t$ holds, the result is always non-negative and less than 2^t. In case of $0 \leq a_i < b_i < 2^t$, we get a negative result, which turns into a large positive number upon when interpreted as unsigned integer.

See [4, Algorithm 14.7] and [4, Algorithm 14.9] for algorithms to do multi-precision addition and subtraction on multi-precision integers which are represented as arrays of unsigned t-bit words. Note that the addition and subtraction algorithms discussed in [4] assume two word arrays of the same length, and that different length arrays require zero-padding.

[2] Here, the caller is responsible to provide arrays of the correct size

2.4 Multiplication

Multiplication of two multi-precision integers can be reduced to multiplication of single-precision words of the operands and accumulation of partial sums. [4, Algorithm 14.12] contains an algorithmic description of standard pencil-and-paper multiplication.

Note that multi-precision multiplication algorithms generally require *all* words of the multiplicands for computing *each* word of the result. As a consequence the operand arrays and the result array must not overlap in memory, if they do, results will be incorrect.

The low-level routines discussed in A.1 document this requirement, but do not require any specific error action. Calling `MpMul` with overlapping operand and result arrays will produce wrong results, without doing any more harm.

The high-level routines discussed in A.2 need to produce correct results even in situations, where operands and results of a `BigIntMul` multiplication overlap. One possible solution is, to use a temporary big integer to hold the immediate computation results inside `BigIntMul`, until the computation is complete. Upon completion `BigIntMul` can copy the temporary big integer to the real result operand.

2.5 Signed big integers

There are different ways to represent signed big-integer quantities. One possible approach would be to extend two's complement representation to multi-precision integers. A second approach is to use a sign-and-magnitude representation, which stores an explicit indication of the sign (positive, zero, negative) and the magnitude (absolute value) of the big integer value.

In this exercise we use the sign-and-magnitude approach, since it is slightly easier to implement and does not require special handling (sign-extension, complement operation) for low-level arithmetic routines. In particular, the sign-and-magnitude representation allows us to perform all arithmetic operations on the magnitude part using the low-level operations discussed earlier.

Here are some useful facts for implementing the `tinybn` library:

- The `BigIntSgn` function can be used to reason about the sign of a big integer a. It corresponds to the signum function (sgn) and is defined as:

$$\operatorname{sgn}(a) := \begin{cases} -1, & \text{if } a < 0 \\ 0, & \text{if } a = 0 \\ +1, & \text{if } a > 0 \end{cases} \tag{3}$$

- Big-integers can be written using the signum function and the magnitude:

$$a = \operatorname{sgn}(a) \cdot |a| \tag{4}$$

- Two big-integers can be multiplied as:

$$a \cdot b = \operatorname{sgn}(a) \cdot |a| \cdot \operatorname{sgn}(b) \cdot |b| = sgn(a \cdot b) \cdot |a| \cdot |b| \tag{5}$$

  The sign of the result is determined by the sign of the operands. The magnitude of the result is determined by the product of the magnitudes of the operands.

- Addition and subtraction of two big-integers a and b can always be rewritten as either addition or subtraction of their magnitudes $|a|$ and $|b|$, followed by proper adjustment of the sign.

Some examples with $a < 0$ and $b > 0$:

$$a = \operatorname{sgn}(a) \cdot |a| = -|a| \quad (6)$$
$$b = \operatorname{sgn}(b) \cdot |b| = |b| \quad (7)$$
$$a+b = -|a| + |b| = |b| - |a| \quad (8)$$
$$a-b = -|a| - |b| = -|a| - |b| = -(|a| + |b|) \quad (9)$$

☞ *Considering all cases for* $\operatorname{sgn}(a)$ *and* $\operatorname{sgn}(b)$ *(with pencil and paper!) can significantly reduce the amount of code you have to write. All big integer addition and subtraction cases can be reduced to a few simple base cases.*

3 Software requirements and setup

The reference platform for this exercise is the VMware[2] image, which can be found in the download section [3] of the course website. The reference image is based on Xubuntu(Precise Pangolin 12.04 LTS) Linux with Apache Tomcat 7.0.12, Oracle JAVA SDK 1.7.0_40, MySQL 5.5.32, lemon 3.7.9, gperf 3.0.3, clang 3.0.6 and CUnit 2.1.2(unstable) and is shipped with a pre-installed Eclipse(Kepler).

⚠ *If you prefer to use your own setup, bear in mind that we can and will only support the reference platform virtual appliance. We can give no guarantees about the compatibility of the software used in this assignment with non-reference platform setups.*

In the remainder of this document we assume that you are using our VMware reference image. Unless explicitly noted otherwise, all command line examples shown here are to be run inside the virtual environment of the reference image and *not on your host computer*.

3.1 Updating your repository

The source code for this assignment is distributed as a Git patch to applied to your repositories. To integrate the updates for this task into your repository you simply have to download the patch-file `https://bigfiles.iaik.tugraz.at/get/3b23b766d2f08f8ffbf2ab09eaee9cf7` and merge it into your local working copy.

To merge the changes for this assignment run the following commands in your working directory:

```
# Download the patch-file for task C1 into your home-directory
sase@vm:~$ wget -O ~/assignment-c1.patch \
  https://bigfiles.iaik.tugraz.at/get/3b23b766d2f08f8ffbf2ab09eaee9cf7

# Switch to your local working copy.
sase@vm:~$ cd ~/<your_sase_repo>

# Ensure that you are on the master branch of your repository
# (We don't want to accidentially clobber a submission branch of
#  a previous task)
sase@vm:~/<your_sase_repo>$ git checkout master

# Merge the patch into your working directory using "git am".
sase@vm:~/<your_sase_repo>$ git am ~/assignment-c1.patch
```

3.2 External libraries

For this assignment your submission *MUST NOT* use any external C libraries, except for the standard C library (libc) and the preinstalled CUnit library (for the unit testing code). The build system provided with the assignment patch is already preconfigured for these libraries.

[3] `http://www.iaik.tugraz.at/content/teaching/master_courses/sicherheitsaspekte_in_der_softwareentwicklung/practicals/downloads/`

It must be possible to successfully compile and run your programs on the VMware reference image! For details on how you can find out if your submission compiles on our submission system, see Section 4.3.

3.3 CUnit

To help you with testing your implementation, we added a set of CUnit tests, which cover basic functionality checks on the functions you have to implement as part of the tasks in Section 5.

⚠ *Please be aware that passing all of those checks does not automatically mean that your implementation is fully correct or secure. The tests only provide minimal functional checks for this task.*

4 Submission

The submission for this tasks consists of all source files, which are required to compile your solutions to the tasks discussed in Section 5. Your solutions *MUST* be compilable by our test system without requiring any modifications to the test system. See Section 4.3 on how you can get intermediate feedback about the build status of your submission.

Your submission should at least contain:

- All source files required to build your submission. They *MUST* be compilable with the build infrastructure from the assignment source patch. See the checklist in Section 4.5 for details.

4.1 Creating the submission branch

The submission itself is done via a separate `submission-c1` branch in your Git repository. To create this branch on the server, you need to execute the following steps *once*:

```
sase@vm:~/<your sase repo>$ git checkout -b submission-c1
sase@vm:~/<your sase repo>$ git push -u origin submission-c1
```

The "checkout" command (with `-b` option) first creates a new local branch in your local repository. The "push" command then publishes your local branch on the server. The `-u` option to the "push" command tells Git to setup tracking for your new branch.

Your group partner needs to setup a local copy of the submission branch as well. To do so, run the following steps in your group partner's local Git repository *once* after you have pushed the submission branch to the server:

```
sase@vm:~/<partner's sase repo>$ git pull
sase@vm:~/<partner's sase repo>$ git checkout -t -b submission-c1 remotes/origin/submission-c1
```

The "pull" command in your colleague's repository fetches changes from the server. The "checkout" command creates a local "submission-c1" branch (`-b` option), which is configured to track (`-t` option) the "submission-c1" branch on the server.

You can use the normal `git push` and `git pull` commands, after you and your colleague have finished the one-time setup of the submission branch.

☞ *We strongly recommend to read `http://git-scm.com/book/en/Git-Branching-Remote-Branches` if you have not worked with Git (remote) branches before.*

4.2 Working with the submission branch

Once you and your colleague have configured the submission branch, you can work with it like any other Git branch. You can directly commit to the submission branch via "git commit" or merge changes from (local) working branches via "git merge".

4.3 Automated compilation

For this task, we will use an automated build system, to regularly compile your submissions in our reference environment. Usually the builds are scheduled at 4:00 AM every day.

Our build bot tries to checkout your submission branch using the steps discussed in section 4.4. On success, the bot then tries to build your submission using our reference build environment. The commit considered by the build bot will be tagged with a Git tag named `iaik/buildbot/c1/XX` where `XX` increases with each scheduled build.

You can fetch the tags generated by the build-bot using either `git pull` or `git fetch --tags`. The build history of your submission branch is accessible via the `git tag -l` command:

```
sase@vm:~/<your sase repo>$ git fetch --tags
sase@vm:~/<your sase repo>$ git tag -n3 -l iaik/buildbot/j/*
 iaik/buildbot/c/1 This commit was built successfully by IAIK's Build-Bot
  Build-Schedule: Wednesday, November 26, 2013 2:50:41 PM CET
  Build-Status: SUCCESS
...
```

4.4 Verifying your submission

You can easily verify your submission by cloning a fresh copy of your repository into a new directory, and trying to check out the `submission-c1` branch. The Git commands used by our build system to clone your submissions are semantically equivalent to:

```
$ git clone git@teaching.student.iaik.tugraz.at:sase2013gXX.git
$ cd sase2013gXX
$ git fetch origin
$ git checkout -b submission-c1 origin/submission-c1
```

Our build-system automatically tags the commit considered as final submission, with an `iaik/submission/c1` tag, once the submission deadline has passed. We will not consider any commits after the `iaik/submission/c1` tag as submission for this task.

4.5 Checklist

Here is a checklist with all steps required to successfully complete this task:

- ☐ Download the source-patch for this task from the IAIK website and apply it to your Git repository as discussed in section 3.1.
- ☐ Create and setup your submission branch as discussed in section 4.1.
- ☐ Solve the tasks discussed in section 5 and commit the code of your solutions to your Git repository.
  - For tasks 5.1 and 5.2 ensure that complete code of your solutions is committed. In particular, check that your changes in the `tasks/c1/src` and `tasks/c1/tests` directories are committed.
  - For task 5.3, question (3.a) ensure that your testcases are comitted in the `tasks/c1/tests/bigint/test_bigint_internals.c` source file.
  - For task 5.3, question (3.b) ensure that you commited your modifications of `tasks/c1/klee/offbyone.c` and `tasks/c1/klee/urldecode.c` source files.
- ☐ Checkout your local `submission-c1` branch and ensure, that everything you want to submit is merged. Ensure that your code can be compiled with the build infrastructure from the assignment source patch.
- ☐ Push your `submission-c1` branch to our servers. You can push the submission branch as often as you like. Any changes after the submission deadline will be ignored.
- ☐ Verify that your submission builds, using the steps discussed in Section 4.3. Follow the steps in Section 4.4 to verify, that you pushed the correct code to the server.

4.6 Deadline

HARD SUBMISSION DEADLINE: December 15, 2013 (23:59:59 (CET))

⚠ *We highly recommend to submit your solution at least one or two days before the ultimate submission deadline. Last minute submissions are always risky with respect to unforeseen technical difficulties.*

5 Tasks

In the practical part of the Too big to fail assignment, you are asked to implement parts of a simple big-integer arithmetic library.

The public *Application Programming Interface (API)* of the library is defined in the `tasks/c1/src/tinybn.h` header file supplied with the assignment patch. A detailed description[4] of these functions and structures, can be found in appendix A.1 and A.2.

☞ *The unit tests provided as part of the assignment source patch define the minimal functional requirements for the big integer arithmetic functions.*

5.1 *(2.5 points)* Low-level (unsigned) multi-precision arithmetic

In this task you have to securely implement some basic low-level functions, which allow you to add, subtract, compare and multiply unsigned multi-precision integers, which are represented by arrays of

[4] Extracted with Doxygen from `tasks/c1/src/tinybn.h`

32-bit unsigned integers. For simplicity reasons, we only support unsigned arithmetic in the low-level primitives of the tinybn library.

These functions are essential for solving the following sub-tasks. They form the back-bone of the higher-level big integer primitives in the next sub-task.

☞ *The detailed description of the low-level multi-precision arithmetic functions of to this task can be found in appendix A.1.*

Questions:

(1.a) ☐ *(1.0 points)* **Addition and subtraction of multi-precision integers** Implement the `MpAdd` and `MpSub` functions discussed in appendix A.1. See [4, Algorithm 14.7] and [4, Algorithm 14.9] for suitable algorithms to add and subtract multi-precision integers.

Test that handling of carries (for addition) and borrows (for subtraction) works properly. Ensure that your functions are robust against invalid parameters, such as `NULL` pointers!

(1.b) ☐ *(1.0 points)* **Multiplication of multi-precision integers** Implement the `MpMul` function to multiply two multi-precision integers. See [4, Algorithm 14.12] for a suitable multiplication algorithm.

Ensure that your multiplication function is robust against invalid parameters, such as `NULL` pointers. Consider the behavior of your function for very large operand array sizes.

(1.c) ☐ *(0.5 points)* **Comparing multi-precision integers** Implement the `MpCompare` function to compare the magnitude of two multi-precision integers.

Ensure that your multiplication function is robust against invalid parameters, such as `NULL` pointers.

5.2 *(7.5 points)* High-level big integer arithmetic

The low-level primitives from the preceding task require the programmer to exercise great care, when manually allocating the word-arrays for operands and results. Especially with multiplication it is easy to get sizes wrong, and to cause a buffer overflow.

In this task we concentrate on implementing a higher-level API, which takes care of operand sizes, and which adds support for signed arithemtic. The big integers supported by the high-level API, will be represented with sign (positive or negative) and magnitude.

☞ *The detailed description of the high-level big integer functions related to this task can be found in appendix A.2.*

⚠ *The 5.2 and 5.2 are critical for the testability of all other parts of the task 5.2. Submission which fail to provide working implementation of 5.2 and 5.2 will receive a very low score for task 5.2.*

Questions:

(2.a) ☐ *(0.5 points)* **Allocating and deallocating big-integers**

Define suitable fields for the `BigInteger` data-structure of your implementation and implement the `BigIntAlloc` and `BigIntFree` functions to allocate and deallocate big integers. Additionally implement the `BigIntCopy` function.

Ensure that your functions are robust against invalid parameters, such as NULL pointers, and out-of-memory errors.

(2.b) ☐ *(0.5 points)* **Accessing the word-level representation** Implement the `BigIntGetWordCount`, `BigIntGetAt` and `BigIntSetAt` functions to user's to access the word-level representation of your big integers

Ensure that your functions are robust against invalid parameters. Consider the out of memory behavior of your `BigIntSetAt` function.

(2.c) ☐ *(2.0 points)* **Import and export of binary data** Implement the `BigIntLoad` and `BigIntSave` functions, to allow conversion between big-integers and plain byte arrays.

Ensure that your functions are roubust against invalid parameters. Consider the out of memory behavior of your `BitIntLoad` function. Ensure that your `BigIntLoad` and `BigIntSave` functions do not perform any out-of-bounds memory accesses.

(2.d) ☐ *(0.5 points)* **Comparison and sign handling** Implement the `BigIntCompare` function to allow comparison of sign and magnitude of big integers. Moreover implement the `BigIntAbs`, `BigIntSgn`, `BigIntNeg` functions to allow manipulation of the sign of a big integer.

Ensure that your functions are robust against invalid parameters. Consider the out of memory behavior of your `BigIntAbs` and `BigIntNeg` functions.

(2.e) ☐ *(2.0 points)* **Multiplication** Implement the `BigIntMul` function to multiply big integers. Use `MpMul` from the preceding task to multiple the magnitudes, and properly propagate the sign of the operands. (e.g. multiplying two negative numbers should yield a positive number).

Ensure that your functions are robust against invalid parameters, and consider the out of memory behavior. Ensure that size computations and memory allocations are free of integer overflows.

(2.f) ☐ *(2.0 points)* **Addition and subtraction** Implement the `BigIntAdd` and `BigIntSub` functions to add and subtract big integers. Use our `MpAdd` and `MpSub` functions from the preceding task to manipulate the magnitudes, and properly handle the signs of the operands.

Ensure that your functions are robust against invalid parameters, and consider their out of memory behavior. Check that carries (for addition) and borrows (for subtraction) are handled properly.

☞ *See [4, Note 14.10] for some information on dealing with borrows resulting from subtraction.*

☞ *Recall that adding a positive number a and a negative number b can be written as subtraction: $a \geq 0, b < 0 : a + b = |a| - |b|$. Similarly subtraction of a negative number b from a positive number a can be written as addition: $a \geq 0, b < 0 : a - b = a - (-|b|) = a + b$.*

5.3 *(5.0 points)* Testing and static code analysis

In tasks 5.1 and 5.2 you were required to securely implement the `tinybn` big integer library. In this task we concentrate on testing your implementation.

Questions:

(3.a) ☐ *(3.5 points)* **Analyze and improve test coverage with lcov.** Measuring code coverage, that is determing which parts of the code are actually reached during unit test or program execution, is one important method to improve code qualitiy.

Measure the code coverage reached by the unit tests in `tasks/c1/tests/bigint` with the `lcov` tool. Then extend the unit tests in `tasks/c1/tests/bigint/test_bigint_internals.c` to increase the line coverage to the highest percentage possible.

To get full points for this questions, ensure that the line coverage[5], as reported by `make lcov-runtests`, for all functions which you implemented in the big-integer library is 100%.

☞ *The makefile of this assignment is already prepared for measuring code coverage and for generating reports using the `lcov`[6] tool. To measure the code coverage of the test suite invoke `make` as:*

```
# Switch to the C1 directory of your local repository
sase@vm:~$ cd ~/<your_sase_repo>/tasks/c1

# Rebuild everything with "lcov-runtests"
```

[5] Ideally the branch coverage should also be close to 100% which usually is harder to achieve.
[6] `http://ltp.sourceforge.net/coverage/lcov.php`

```
sase@vm:~/<your_sase_repo>/tasks/c1$ make lcov-runtests

# Open the results in a browser
sase@vm:~/<your_sase_repo>/tasks/c1$ chromium-browser _cov/index.html
```

☞ *The test framework already provides several useful functions*[7] *to simulate out of memory conditions from within test cases.*

(3.b) ☐ *(1.5 points)* **Using symbolic execution to exhibit bugs.** Symbolic execution is a powerful program analysis method, which tries to consider all possible execution paths of a program for all possible inputs. The LLVM toolkit includes a symbolic analysis tool called *KLEE*[8], which can be used to automatically find inputs, which trigger invalid behavior (e.g. buffer overflows).

Use the KLEE tool to find the bugs in the `offbyone.c`, and `urldecode.c` programs. They are in the `tasks/c1/klee` directory of your repository. First, you need to add appropriate `klee_make_symbolic` and `klee_assert` calls to the source files. Then run your modified files for throught the KLEE tool.

☞ *You need to compile and install KLEE in your reference virtual machine. See the `tasks/c1/klee/install.txt` file for the required commands.*

⚠ *Before trying to build or use KLEE, ensure that `llvm-gcc-4.6` is NOT installed on your virtual machine. Moreover ensure that the `gold` linker is installed as your system default linker. You can use the following commands:*

```
sase@vm:~$ sudo apt-get remove llvm-gcc-4.6
sase@vm:~$ sudo apt-get install binutils-gold
```

☞ *The makefile in the `tasks/c1/klee` already provides special targets for compiling and testing your code with KLEE.*

A Module Documentation

A.1 Low-level multi-precision arithemtic functions.

Defines

- #define **MP_WORD_C**(x) UINT32_C(x)

  Declares a single-precision word constant.
- #define **MP_WORD_BITS** 32U

  Bit-size of a single-precision word.
- #define **MP_WORD_MAX** UINT32_MAX

  Maximum value of a single-precision word.
- #define **MP_LONG_C**(x) UINT64_C(x)

  Declares a double-precision word constant.
- #define **MP_LONG_MAX** UINT64_MAX

  Maximum value of a double-precision word.

[7] See `TestSetAllocSizeLimit`, `TestSetAllocFaultCountdown`, `TestNoAllocFaults`
[8] `http://klee.llvm.org/`

Typedefs

- typedef uint32_t **mp_word_t**

  Single-precision word (32-bit).
- typedef uint64_t **mp_long_t**

  Double-precision word (64-bit).

Functions

- bool **MpAdd** (**mp_word_t** ∗z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Adds two (unsigned) multi-precision integers.
- bool **MpSub** (**mp_word_t** ∗z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Subtracts two (unsigned) multi-precision integers.
- int **MpCompare** (const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Compares two multi-precision integers.
- void **MpMul** (**mp_word_t** ∗restrict z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Multiplies two (unsigned) multi-precision integers.

A.1.1 Define Documentation

A.1.1.1 #define MP_LONG_C(*x*) UINT64_C(x)

This macro allows portable definition of (small) `mp_long_t` constants.

A.1.1.2 #define MP_LONG_MAX UINT64_MAX

The **MP_LONG_MAX** (p. 14) macro denotes the largest value which can be stored in a single **mp_long_t** (p. 15) variable.

A.1.1.3 #define MP_WORD_BITS 32U

The **MP_WORD_BITS** (p. 14) macro indicates the number of bits in a single **mp_word_t** (p. 15) word. Currently this value is fixed to 32 bits.

A.1.1.4 #define MP_WORD_C(*x*) UINT32_C(x)

This macro allows portable definition of (small) `mp_word_t` constants.

A.1.1.5 #define MP_WORD_MAX UINT32_MAX

The **MP_WORD_MAX** (p. 14) macro denotes the largest value which can be stored in a single **mp_word_t** (p. 15) variable.

A.1.2 Typedef Documentation

A.1.2.1 typedef uint64_t mp_long_t

The **mp_long_t** (p. 15) type is an unsigned integer type with twice the bit-size of **mp_long_t** (p. 15). It is used by the big integer library to represent double precision values (such as the results of multiplying two single precision words). This type is currently fixed to the standard unsigned 64-bit (`uint64_t`) integer data type.

A.1.2.2 typedef uint32_t mp_word_t

The **mp_word_t** (p. 15) type is an unsigned integer type, used by the big integer library to represent single precision words. It is currently fixed to the standard unsigned 32-bit (`uint32_t`) integer data type.

A.1.3 Function Documentation

A.1.3.1 bool MpAdd (mp_word_t * *z*, const mp_word_t * *a*, size_t *len_a*, const mp_word_t * *b*, size_t *len_b*)

This functions adds two multi-precision integers given by the word arrays `a` (of length `len_a`) and `b` (of length `len_b`). The result is stored in the `z` word array, the length of the result array is the larger of `len_a` and `len_b`.

In case of different operand lengths ($len\_a \neq len\_b$), the missing words of the shorter operand are assumed to be zero.

Parameters

`out`	*z*	points to the word array where the result (sum) will be stored. The array must be large enough to hold $\max\{len\_a, len\_b\}$ elements.
`in`	*a*	points to the word array with the first addend (left operand). This pointer may refer to the same memory region as the `z` pointer.
	len_a	is the length of the word array referenced by the `a` parameter. This function only accesses elements of the `a` array with index i in range $0 \leq i < len\_a$. No access to the `a` array is performed if `len_a` is zero.
`in`	*b*	points to the word array with the second addend (right operand). This pointer may refer to the same memory region as the `z` pointer.
	len_b	is the length of the word array referenced by the `b` parameter. This function only accesses elements of the `b` array with index i in range $0 \leq i < len\_b$. No access to the `b` array is performed if `len_a` is zero.

Returns

The carry value of the multi-precision addition. The carry comes from the final single-precision addition in the top word of the result, and can be zero (result fits into $\max\{len\_a, len\_b\}$ words) or one, (carry in top word).

A.1.3.2 int MpCompare (const mp_word_t * *a*, size_t *len_a*, const mp_word_t * *b*, size_t *len_b*)

This functions compares the magnitude of two unsigned multi-precision integers. the return value is `0` if both integers are equal, `1` if `a` is greater than `b`, or `-1` if `a` is less than `b`.

In case of different operand lengths ($len\_a \neq len\_b$), th missing words of the shorter operand are assumed to be zero.

Parameters

`in`	*a*	points to the word array with the first (left) operand.
	len_a	is the length of the word array referenced by the `a` parameter. This function only accesses elements of the `a` array with index i in range $0 \le i < len\_a$. No access to the `a` array is performed if `len_a` is zero.
`in`	*b*	points to the word array with the second (right).
	len_b	is the length of the word array referenced by the `b` parameter. This function only accesses elements of the `b` array with index i in range $0 \le i < len\_b$. No access to the `b` array is performed if `len_a` is zero.

Returns

0 if both multi-precision integers are equal.
1 if big-integer `a` is strictly greater than big-integer `b`.
-1 if big-integer `a` is strictly less than big-integer `b`.

A.1.3.3 void MpMul (mp_word_t *restrict *z*, const mp_word_t * *a*, size_t *len_a*, const mp_word_t * *b*, size_t *len_b*)

This function multiplies two multi-precision integers given by the word arrays `a` (of length `len_a`) and `b` (of length `len_b`). The result is stored in the `z` word array, the length of the result array is the sum of `len_a` and `len_b`.

Note

Multi-precision multiplication needs to access both operand arrays while computing the product. This function requires that the result array `z` does not overlap with the operand array `a` or `b`. - Violation of this restriction will produce incorrect results in the `z` array.

Parameters

`out`	*z*	points to the word array where the result (product) will be stored. The array must be large enough to hold len_a+len_b elements. The memory region referenced by this pointer MUST NOT overlap the memory regions referenced by the `a` or `b` parameters.
`in`	*a*	points to the word array with the first factor (left operand). This pointer MUST NOT not refer to the same memory region as the `z` pointer.
	len_a	is the length of the word array referenced by the `a` parameter. This function only accesses elements of the `a` array with index i in range $0 \le i < len\_a$. No access to the `a` array is performed if `len_a` is zero.
`in`	*b*	points to the word array with the second factor (right operand). This pointer MUST NOT refer to the same memory region as the `z` pointer.
	len_b	is the length of the word array referenced by the `b` parameter. This function only accesses elements of the `b` array with index i in range $0 \le i < len\_b$. No access to the `b` array is performed if `len_a` is zero.

A.1.3.4 bool MpSub (mp_word_t * *z*, const mp_word_t * *a*, size_t *len_a*, const mp_word_t * *b*, size_t *len_b*)

This function subtracts two multi-precision integers given by the word arrays `a` (minuend of length `len_a`) and `b` (subtrahend of length `len_b`). The result is stored in the `z` word array, the length of the result array is the larger of `len_a` and `len_b`.

In case of different operand lengths ($len\_a \neq len\_b$), the missing words of the shorter operand are assumed to be zero.

Note

Individual words of the operands are subtracted as *unsigned* integers, and have C unsigned subtraction overflow semantics. (e.g. subtracting 1 from 0 will produce $2^{32} - 1$.

Parameters

`out`	z	points to the word array where the result (sum) will be stored. The array must be large enough to hold $\max\{len\_a, len\_b\}$ elements.
`in`	a	points to the word array with the minuend (left operand). This pointer may refer to the same memory region as the `z` pointer.
	len_a	is the length of the word array referenced by the `a` parameter. This function only accesses elements of the `a` array with index i in range $0 \le i < len\_a$. No access to the `a` array is performed if `len_a` is zero.
`in`	b	points to the word array with the subtrahend (right operand). This pointer may refer to the same memory region as the `z` pointer.
	len_b	is the length of the word array referenced by the `b` parameter. This function only accesses elements of the `b` array with index i in range $0 \le i < len\_b$. No access to the `b` array is performed if `len_a` is zero.

Returns

The borrow value of the multi-precision subtraction. The borrow comes from the final single-precision subtraction in the top word of the result, and can be zero ($a \ge b$ or one ($a < b$).

A.2 High-level big-integer arithmetic functions.

Functions

- **BigInteger** ∗ **BigIntAlloc** (void)

  Allocates a new big integer.
- void **BigIntFree** (**BigInteger** ∗z)

  Deallocates a big integer.
- bool **BigIntLoad** (**BigInteger** ∗z, const unsigned char ∗data, size_t len)

  Loads the value of a big-integer from a byte array.
- bool **BigIntSave** (unsigned char ∗data, size_t len, const **BigInteger** ∗z)

  Stores the value of a big-integer to a byte array.
- size_t **BigIntGetWordCount** (const **BigInteger** ∗z)

  Gets the minimum number of words required to represent the magnitude of big-integer `z`.
- **mp_word_t BigIntGetAt** (const **BigInteger** ∗z, size_t index)

  Gets a word from the word array of a big-integer.
- bool **BigIntSetAt** (**BigInteger** ∗z, size_t index, **mp_word_t** value)

  Sets a word in word array of a big-integer.
- bool **BigIntAdd** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Adds two big integers. ($z = a + b$)
- bool **BigIntSub** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Subtracts two big integers. ($z = a - b$)
- bool **BigIntMul** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Multiplies two big integers. ($z = a \cdot b$)
- bool **BigIntCopy** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Copies the sign and magnitude of a big integer. ($z = a$)
- bool **BigIntNeg** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Negates a big integer. ($z = -a$)
- bool **BigIntAbs** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Gets the absolute value of a big integer. ($z = |a|$)
- int **BigIntSgn** (const **BigInteger** ∗a)

  Gets the sign of a big integer. ($z = sgn a$)
- int **BigIntCompare** (const **BigInteger** ∗a, const **BigInteger** ∗b)

  Compares the sign and magnitude of two big integers.

A.2.1 Function Documentation

A.2.1.1 bool BigIntAbs (BigInteger ∗ z, const BigInteger ∗ a)

This function returns the absolute value of `a`, by copying the magnitude to `z` and making the sign of `z` positive (respectively zero, if the magnitude is zero).

Parameters

in,out	*z*	points to the destination integer. z may refer to the same object as a.
in	*a*	points to the source integer.

Returns

true if the operation succeeded.
false if the operation failed. (e.g. bad parameters, out of memory).

A.2.1.2 bool BigIntAdd (BigInteger * *z*, const BigInteger * *a*, const BigInteger * *b*)

This function adds the big integers a and b and stores the result in z. This operation honors the signs of its operands; e.g. adding a positive and a negative big integer effectively behaves as subtraction.

Parameters

in,out	*z*	points to the big-integer receiving the result. z may refer to the same object as a and/or b.
in	*a*	points to the first (left) big integer operand.
in	*b*	points to the second (right) big integer operand.

Returns

true if the operation succeeded.
false if the operation failed. (e.g. bad parameters, out of memory).

A.2.1.3 BigInteger* BigIntAlloc (void)

This function allocates a new big-integer object and initializes it to represent the value zero.

Returns

the newly allocated big-integer object. The returned object should deleted with **BigIntFree** (p. 20) when it is no longer needed, in order to prevent memory leaks.
NULL if the allocation failed (e.g. due to an out of memory condition)

A.2.1.4 int BigIntCompare (const BigInteger * *a*, const BigInteger * *b*)

This function compares the sign and magnitude of big integers a and b. The return value indicates if a is less than b (-1), greater than b (+1), or equal to b (0).

Parameters

in	*a*	points to the first (left) big integer operand.
in	*b*	points to the second (right) big integer operand.

Returns

-1, if and only if $a < b$.
0, if $a = b$ or if at least one of the operands is NULL.
+1, if and only if $a > b$.

A.2.1.5 bool BigIntCopy (BigInteger * *z*, const BigInteger * *a*)

This function copies the sign and magnitude of a to z. It behaves as a no-operation if a and z refer to the same object.

Parameters

in,out	*z*	points to the destination integer. z may refer to the same object as a.
in	*a*	points to the source integer.

Returns

`true` if the operation succeeded.
`false` if the operation failed. (e.g. bad parameters, out of memory).

A.2.1.6 void BigIntFree (BigInteger * *z*)

This function deletes a big-integer, which had been previously allocated with **BigIntAlloc** (p. 19), and releases all associated (memory) resources.

Parameters

`in,out`	*z*	is the big-integer to be deleted.

A.2.1.7 mp_word_t BigIntGetAt (const BigInteger * *z*, size_t *index*)

This function returns the word at position `index` of the word-array of big integer `z`. The least significant word is at index `0`.

Parameters

`in`	*z*	the big integer to access.
`in`	*index*	the zero-based index of the word to be accessed.

Returns

the word at the given index of the word array of big integer `z`, if `z` and `index` are valid. (the returned value may be zero)
0, if `z` is `NULL` or `index` is outside the word array bounds.

A.2.1.8 size_t BigIntGetWordCount (const BigInteger * *z*)

This function returns the minimum number of **mp_word_t** (p. 15) words, that is required to represent the magnitude (absolute value) of `z`. The return value of this function is greater or equal than 1 for any non-null `z` object.

Parameters

`in`	*z*	the big integer to be queried.

Returns

the minimum number of words required to represent the magnitude of the given big integer, if `z` is not `NULL`. The returned value is greater or equal than 1.
0, if and only if `z` is `NULL`

A.2.1.9 bool BigIntLoad (BigInteger * *z*, const unsigned char * *data*, size_t *len*)

This function sets the magnitude of the big-integer `z` to the unsigned value represented by the `data` byte array. The `len` parameter gives the number of valid data bytes found at `data`.

The `data` array encodes the unsigned value in big-endian byte-order. The most significant byte being at index zero. The `data` array may contain one or more leading zeros.

The following code snippet demonstrates how the big integer value `0x123456789ABCDEF` can be loaded into a big-integer `p`.

```
...

unsigned char raw[9] = {
   0x01, 0x23, 0x45, 0x67,
   0x89, 0xAB, 0xCD, 0xEF
};

BigInteger *p = BigIntAlloc();
BigIntLoad(p, raw, 9);
```

...

Note

This function only loads the magnitude (absolute value) of z. The sign of z is assumed to be positive (or zero if the magnitude is zero). It is the API user's responsibility to restore the sign, if needed.

Parameters

in,out	*z*	is the valid big-integer to be set to a new value.
in	*data*	is a pointer to the byte-array representing the new value of z. This value may be NULL if len is zero.
in	*len*	is the length of the byte-array referenced by the data parameter. - This function only accesses elements of the byte-array with indices i in range $0 \leq i < len$. No array access is done if len is zero.

Returns

true if the new value of the big-integer was successfully assigned from the byte array.
false in case of an error. (e.g. out of memory, bad parameters)

A.2.1.10 bool BigIntMul (BigInteger * *z*, const BigInteger * *a*, const BigInteger * *b*)

This function multiplies the big integers a and b and stores the result in z. This operation honors the signs of its operands; e.g. multiplying two negative integer yields a positive integer.

Note

The high-level allows a and/or b to be the same object as the result z (a and/or b may alias with z). In order to produce correct results in this case, this function may need to create a temporary big integer for holding the result, while the multiplication is in progress.

Parameters

in,out	*z*	points to the big-integer receiving the result. The z may refer to the same object as a and/or b.
in	*a*	points to the first (left) big integer operand.
in	*b*	points to the second (right) big integer operand.

Returns

true if the operation succeeded.
false if the operation failed. (e.g. bad parameters, out of memory).

A.2.1.11 bool BigIntNeg (BigInteger * *z*, const BigInteger * *a*)

This function negates the sign of a and stores the negated big integer in z. The magnitude of a is copied to z without any changes.

Parameters

in,out	*z*	points to the destination integer. z may refer to the same object as a.
in	*a*	points to the source integer.

Returns

true if the operation succeeded.
false if the operation failed. (e.g. bad parameters, out of memory).

A.2.1.12 bool BigIntSave (unsigned char * *data*, size_t *len*, const BigInteger * *z*)

This function writes the len least-significant bytes of the magnitude of the big-integer z to the byte array described by data and len. The result will be padded with leading zeros if len is greater than the

minimum required value. The result will be silently truncated, if the length is less than the minimum required value. The maximum number of bytes required to store a big integer can be estimated by multiplying the result of **BigIntGetWordCount** (p. 20) with the size of **mp_word_t** (p. 15).

Note

This function only stores the magnitude (absolute value) of `z`. It is the API user's responsibility to separately save the sign, if needed.

Parameters

`in`	*data*	is a pointer to the byte-array receiving the `len` least-signifcant bytes of the magnitude of `z`. This pointer may be `NULL` if `len` is zero.
`in`	*len*	is the length of the byte-array referenced by the `data` parameter. This function will accesses all elements of the byte-array with indices i in range $0 \leq i < len$. No array access is done if `len` is zero.
`in,out`	*z*	is the valid big-integer to be set to a new value.

Returns

`true` if `len` least-significant bytes of the magnitude of the big-integer was successfully written to the byte array.
`false` in case of an error. (e.g. out of memory, bad parameters)

See also

BigIntLoad (p. 20) for additional details on the format of the byte array.

A.2.1.13 bool BigIntSetAt (BigInteger * *z*, size_t *index*, mp_word_t *value*)

This function sets the word at position `index` of the word-array of big integer `z` to `value`. The least significant word is at index `0`. The word array of the big integer automatically grows, if the given index is outside the current bounds.

Parameters

`in`	*z*	the big integer to access.
`in`	*index*	the zero-based index of the word to be set.

Returns

`true` if the word was successfully set.
`false` in case of an error. (e.g. bad parameter, out of memory)

A.2.1.14 int BigIntSgn (const BigInteger * *a*)

This function implements the signum function for big integer. It returns a value indicating if the given big integer `a` is positive (+1), negative (-1), or zero (0).

Parameters

`in`	*a*	points to the source integer.

Returns

-1, if and only if $a < 0$.
0, if $a = 0$ or if the `a` pointer is `NULL`.
+1, if and only if $a > 0$.

A.2.1.15 bool BigIntSub (BigInteger * *z*, const BigInteger * *a*, const BigInteger * *b*)

This function subtracts the big integers `a` and `b` and stores the result in `z`. This operation honors the signs of its operands; e.g. subtracting a negative from a positive integer effectively behaves as addition.

Parameters

`in,out`	z	points to the big-integer receiving the result. `z` may refer to the same object as `a` and/or `b`.
`in`	a	points to the first (left) big integer operand.
`in`	b	points to the second (right) big integer operand.

Returns

`true` if the operation succeeded.
`false` if the operation failed. (e.g. bad parameters, out of memory).

B Data Structure Documentation

B.1 BigInteger Struct Reference

An arbitrary-precision big integer.

B.1.1 Detailed Description

The **BigInteger** (p. 24) data-type represents a signed arbitrary precision big-integer.

Note

The internal details of this data-structure are not part of the public big integer API.

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

- **tinybn_imp.h**

C File Documentation

C.1 tinybn.h File Reference

Public API of the tinybn big integer library.

Defines

- #define **MP_WORD_C**(x) UINT32_C(x)

  Declares a single-precision word constant.
- #define **MP_WORD_BITS** 32U

  Bit-size of a single-precision word.
- #define **MP_WORD_MAX** UINT32_MAX

  Maximum value of a single-precision word.
- #define **MP_LONG_C**(x) UINT64_C(x)

  Declares a double-precision word constant.
- #define **MP_LONG_MAX** UINT64_MAX

  Maximum value of a double-precision word.

Typedefs

- typedef uint32_t **mp_word_t**

  Single-precision word (32-bit).
- typedef uint64_t **mp_long_t**

  Double-precision word (64-bit).

Functions

- bool **MpAdd** (**mp_word_t** ∗z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Adds two (unsigned) multi-precision integers.
- bool **MpSub** (**mp_word_t** ∗z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Subtracts two (unsigned) multi-precision integers.
- int **MpCompare** (const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Compares two multi-precision integers.
- void **MpMul** (**mp_word_t** ∗restrict z, const **mp_word_t** ∗a, size_t len_a, const **mp_word_t** ∗b, size_t len_b)

  Multiplies two (unsigned) multi-precision integers.
- **BigInteger** ∗ **BigIntAlloc** (void)

  Allocates a new big integer.
- void **BigIntFree** (**BigInteger** ∗z)

  Deallocates a big integer.
- bool **BigIntLoad** (**BigInteger** ∗z, const unsigned char ∗data, size_t len)

  Loads the value of a big-integer from a byte array.
- bool **BigIntSave** (unsigned char ∗data, size_t len, const **BigInteger** ∗z)

  Stores the value of a big-integer to a byte array.
- size_t **BigIntGetWordCount** (const **BigInteger** ∗z)

  Gets the minimum number of words required to represent the magnitude of big-integer ***z****.*
- **mp_word_t BigIntGetAt** (const **BigInteger** ∗z, size_t index)

  Gets a word from the word array of a big-integer.
- bool **BigIntSetAt** (**BigInteger** ∗z, size_t index, **mp_word_t** value)

  Sets a word in word array of a big-integer.
- bool **BigIntAdd** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Adds two big integers. ($z = a + b$*)*
- bool **BigIntSub** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Subtracts two big integers. ($z = a - b$*)*
- bool **BigIntMul** (**BigInteger** ∗z, const **BigInteger** ∗a, const **BigInteger** ∗b)

  Multiplies two big integers. ($z = a \cdot b$*)*
- bool **BigIntCopy** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Copies the sign and magnitude of a big integer. ($z = a$*)*
- bool **BigIntNeg** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Negates a big integer. ($z = -a$*)*
- bool **BigIntAbs** (**BigInteger** ∗z, const **BigInteger** ∗a)

  Gets the absolute value of a big integer. ($z = |a|$*)*
- int **BigIntSgn** (const **BigInteger** ∗a)

  Gets the sign of a big integer. ($z = sgn a$*)*
- int **BigIntCompare** (const **BigInteger** ∗a, const **BigInteger** ∗b)

Compares the sign and magnitude of two big integers.

C.1.1 Detailed Description

Warning

This header file defines the public API interface of the tinybn big integer library. It is part of the assignment specification and MUST NOT be changed. Details (declaration of internal functions of your implementation, defines, type definitions, ...) should go into the `tinybn_imp.h` (p. 26) header file.

C.2 tinybn_imp.h File Reference

Public API of the tinybn big integer library.

Data Structures

- struct **BigInteger**

  An arbitrary-precision big integer.

C.2.1 Detailed Description

Note

This header file contains any implementation details of your big integer library. You can put declarations of internal helper functions, type and structure definitions, constants, etc. in this file.

References

[1] *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. `http://cwe.mitre.org/top25/`.

[2] *VMWare*. `http://www.vmware.com/`.

[3] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. *PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure*. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2010. Available online at: `http://dx.doi.org/10.1007/978-3-642-14577-3_22`.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 1996. An online version is available at: `http://cacr.uwaterloo.ca/hac/`.