1 Introduction

This document describes an implementation of a generic Key Policy Attribute Based Encryption (KPABE) scheme, as detailed in the paper by Crampton and Pinto in CSF'14. The driving idea is that such a scheme can be implemented using an arbitrary linear secret sharing scheme (LSSS) as a building block. This implementation project had the following objectives:

- provide an implementation of a generic scheme based on any LSSS;
- compare the performance of two basic implementation schemes: Benaloh-Leichter with canonical policies (canonical BL); a tree of Shamir schemes (Shamir Tree);

The language chosen to implement was C++. The main reason was the availability of the library MIRACL to perform cryptographic computations and operations on big numbers, by advice of a member of the project (Kenny Patterson), since it is freely available, open source, fast and is perhaps the best available implementation of pairing-based cryptography. This suggested two possible languages: plain C and C++. C++ was chosen because of its object-oriented nature, that better suits the main idea of the scheme: modularity of the secret sharing scheme within the ABE scheme.

As much as possible, the code was developed with a Test-Driven Development methodology (TDD), meaning that tests should be developed before the implementation of the actual code to be tested. This was not always achieved, but at the worst the tests were developed in parallel with the code. These tests run on the console and use colour coded output to quickly signal the success or failure of tests.

The implementation is divided in different types of files:

- header files with the declaration of functions and classes;
- implementation files of said classes. These are compiled into object files only, not into executables;
- testfiles, that link with the relevant object files and produce executables to run the tests;
- a benchmark file, that produces executables to obtain performance data;
- a makefile to produce all the object and executable code;
- some configuration files that hold declarations similar to those in header files, but are separated from these to provide variant; compilations. This is necessary to produce distinct benchmark files from the same source code.

2 Architecture

The central notions of the data model are the concepts of KPABE and of LSSS scheme, and are implemented in their respective classes: `KPABE` and `SecretSharing`. The class `SecretSharing` requires other supporting classes. A secret sharing scheme implements one access policy of a specific type. The notion of access policy is captured by the class `AccessPolicy`. The function of a secret sharing scheme is to produce shares of a given secret, and to reconstruct a secret from a set of shares. Shares are held by participants of the scheme, in a many-to-one relationship. The reconstruction algorithm can be performed by anyone with access to the shares. If the access policy is not trivial, this operation usually requires some information about the share, besides its share value proper, that is known by all participants: the share's public information. Thus, the concept of share is not representable by a single primitive type. Therefore, it is captured by a class called `ShareTuple`.

Class `kpabe.cpp` is declared in file `kpabe.h` and defined in file `kpabe.cpp`. The file `testkpabe.cpp` holds tests for this construction. The class `KPABE` itself is not abstract, but it can only be instantiated with an instance of a concrete secret sharing scheme. The test file runs a battery of tests for two instantiations of `KPABE`, one for each of the two secret sharing schemes currently implemented in this project.

The classes `ShareTuple`, `AccessPolicy` and `SecretSharing` are all declared in file `secretsharing.h` and defined in file `secretsharing.cpp`. Due to the generality of the concepts they represent, the classes `SecretSharing` and `AccessPolicy` are abstract, since some of their properties can only be implemented in specific concrete instances.

This implementation provides, so far, two concrete implementations: a Benaloh-Leichter scheme for canonical policies; and a tree of simple Shamir threshold schemes. From this base it is easy to provide a simple Shamir threshold scheme (there is an old implementation of that in the `DeadCode` directory, that no longer compiles, in files `shamir2.h`, `shamir2.cpp` and `testshamir2.cpp`) and a generalized Benaloh-Leichter, that accepts a tree of `AND`, `OR` and possibly `THRESHOLD` gates. There is also preliminary work in the DeadCode section, in files `BL.h`, `BL.cpp` and `testBL.cpp`.

The canonical BL scheme is implemented by the classes `BLSS` and `BLAccessPolicy`, both declared in file `BLcanonical.h` and defined in file `BLcanonical.cpp`. Accompanying tests are implemented in file `testBLcanonical.cpp`. Analogously, the tree of Shamir schemes is captured by classes `ShTreeSS` and `ShTreeAccessPolicy`, implemented and tested in the corresponding files `ShTree.h`, `ShTree.cpp`, `testShTree.cpp`.

Because of its tree-like nature, a Shamir Tree policy is best represented by a tree data structure. There is a natural difference between leaf and inner nodes of a tree, but in this case the information that needs to be associated to the leaf nodes is specifically different of that in the inner nodes. For that reason, I developed a tree data structure specific for this project. It is captured by two classes: `NodeContent` describes a particular node, and its associated data, without regards to its position in a tree; `TreeNode` describes a tree as a collection

of nodes related in a specific way. Both these classes are declared in `tree.h` and defined in `tree.cpp`. The respective tests are in file `testtree.cpp`.

Some operations are repeatedly needed throughout the code that were not readily available in the language. These have been implemented aside as "utility" functions, in files `utils.h` and `utils.cpp`, with some tests in file `testutils.cpp`. These include the declaration of all header files needed by the project, definition of codes to write in colour for a terminal, definitions of macros for writing messages to the terminal in a quick fashion, mostly used for debugging, a flag that tells whether to ignore debug output messages, some error codes for exception messages, constants defining how to represent policy operators, functions to report the result of tests, and some more specialized functions. These will be described below in more detail.

3 Implementation Details

3.1 Using MIRACL

This project implements a cryptographic scheme for use in the real-world. Since its security is based on number-theoretic complexity assumptions, it requires very large numbers that can not be handled by the primitive types and functions of C++. Therefore, implementation relies on an external library, MIRACL, to provide these services. The current version is 4.0.

In particular, it uses MIRACL's ability to manipulate large integers, and operations on asymmetric pairings of groups $G_1 \times G_2 \to G_T$.

MIRACL provides different instantiations of pairing curves, with varying levels of AES-comparable security from 80 bits to 256 bits. For efficiency reasons, we chose Barreto-Naehrig curves for a 128-bit security level, which is currently considered secure in practice.

To specify this choice, the code must define two macros:

```
#define MR_PAIRING_BN    // Barreto-Naehrig curves
#define AES_SECURITY 128 // AES-128 security
```

This is defined at the start of `utils.h`, which is then ultimately included by all source files in the project. In order to use the pairing related functions, this file also includes the header `pairing_3.h`, which gives the interface for Type 3 asymmetric group pairings.

Code with MIRACL in C++ requires special care. Any variable declared in C++ is immediately initialized at the point of declaration with some default value. In the case of classes, like `Big`, this means invoking a default constructor. Therefore, MIRACL must be initialized at the start of any source code that will generate an executable file. In this project, that means some of the test files and the benchmarks.

MIRACL is essentially a C library, that also provides a C++ interface. The way to initialize in C is to invoke `mirsys` to obtain a Miracl Instance Pointer (mip). This is a static global pointer that gives access to all the internal workings of MIRACL:

```
miracl *mip = mirsys(5000,0);
```

In C++, this can be replaced by the creation of a MIRACL instance and then taking its address, for example:

```
Miracl precision(5,0);
miracl* mip = &precision;
```

However, even this is not necessary in this particular project. The basis for the pairing computations we use is the class `PFC` (for Pairing Friendly Curve), declared in `pairing_3.h`. For the choice of curve of this project, the respective implementation is in `bn_pair.cpp`. There, it can be seen that the constructor of `PFC` already calls `mirsys`, and so another call is not needed.

Therefore, in this project initialization is done simply by creating a `PFC` class and then getting a handle to the Miracl pointer. This can be done with:

```
PFC pfc(AES_SECURITY);
miracl *mip=get_mip();
mip->IOBASE=16;
```

The third line is not strictly needed. It is already included in the constructor of `PFC`. I include it only for clarity, so that the reader will know its value without having to read the source code of an external library.

Class `PFC` does not redefine a copy constructor, and uses the default. This does not seem to work well in practice, attempts to pass an instance of `PFC` by value have always resulted in a long list of errors. There should be no reason to do that either, as this is mostly used for access to functions over external data. Therefore, variables of `PFC` type should always be passed as references, which is also the fastest way to pass values in and out of functions.

3.2 `utils.h`/ `utils.cpp`/ `utils_impl.tcc`

These files hold declarations and definitions of basic constants and functions to be used throughout the whole code. In most cases, they are nothing more than conveniences.

The first such convenience is the full set of included headers for the whole project. The macros beginning with `sh` denote ANSI codes for colours, so that the program can write coloured to the terminal. However, when redirecting the output of the program to a file, these colour codes will remain untranslated and pollute the output. They are mainly used to help in quickly reporting to the user, mostly for console debugging and for communicating tests' results.

The macro `NODEBUG` controls whether the program should output debugging information. This is done by calling special functions `DEBUG` and `ENHDEBUG` in the code as convenient. If the `NODEBUG` is *not defined*, then the debugging mode is active, and some text is output to the terminal. Otherwise, these instructions just do nothing. The difference between `DEBUG` and `ENHDEBUG` is strictly cosmetic, offering two different colour schemes to help the programmer and debugger distinguish information with different levels of importance.

Other macros `OUT`, `ENHOUT` and `REPORT` also serve to send coloured data to the screen, offering even more ways to distinguish information. These, however, are not turned off by the `NODEBUG` macro and so always produce a visible result. All the debugging and reporting macros allow for a very convenient way to output data, by streaming directly to standard output without the need for conversion functions.

There are four error codes defined, their macros all starting by `ERR_`. These are used in particular places where the code must throw an exception, because data input from the outside lead to some particular kind of error. The value of these macros is a `std::string`, that is meant to be output as the prefix of the error message of the thrown exception.

Following, are three constants, of type `std::string`, that tell the program how operators for policy expressions are represented. Currently, `OR` represents logical OR, `AND` represents logical AND and `THR` represents a threshold function. Policies for use with this project should be written in prefix, or functional, notation, with each policy operator followed by a parenthesized argument list. Arguments, that is, attributes, should be represented by positive integer numbers only.

Finally, these files declare and define a few functions.

`guard` this is basically an assertion: if the condition fails, the program stops. The difference to a normal `assert` is that this function outputs a message before stopping.

`test_diagnosis` this function offers a standard way to run a test: it writes whether the test was successful, and if not increases an error (test failure) count.

`print_test_diagnosis` this function prints a final message reporting whether a test batch was successful or how many tests failed.

`convertStrToInt` this is a convenient function to parse a `std::string` into an int. If the `std::string` does not represent an integer number, the function throws an exception with the appropriate error message.

`convertIntToStr` this function calls a function from the standard library to do this conversion. It provides an alternative, possibly easier to remember, name.

`exprTokenize` this function is important for use in parsing policies. It divides a `std::string` into individual tokens, which are separated by a single delimiter that can be passed as argument. The result is returned in an outbound `std::vector`. Unlike a normal tokenizer, this function is able to recognize parts of the sub-expression that are protected by parentheses, and which are not tokenized. For example, tokenizing the expression `3, 7, g(1,5,2)` with a delimiter ',' will yield 3 tokens: `3`, `7`, `g(1,5,2)`.

`trim` this is a basic function that removes all white space from the beginning and the end of a `std::string`.

`isSuffix` this tests whether the second argument begins with the first argument. Both arguments are of type `std::string`.The name is misleading, as indeed no argument is a suffix of the other, but instead the first is a prefix of the second, or the second is the first with an added suffix.

Finally, a group of template classes are declared, for use in several places of the project. Template classes must be defined in the header file, but in order to keep the distinction of declaration and definition in different files, I have adopted a standard practice: define template functions in a third file and include at the right point in the header file. This third file is `utils_impl.tcc`.

`contains` this function receives a vector with elements of type `T` and an element of said type, and returns the index of this element in the vector, if it exists there. Otherwise, it returns -1.

`addVector` this function receives two vectors of the same type and appends all the elements of the second vector to the first vector, in the same order.

`debugVector` this is a function that receives a vector of some primitive type and a `std::string` message. It lists the contents of the vector, preceding each element by the message and its position. It requires that the vector type can be output to a stream. This relies on the `DEBUG` macro, and so it will not produce results if `NODEBUG` is defined.

`debugVectorObj` this is just like `debugVector`, except it works with types that can not be streamed but implement a method `to_string()`.

`outVector` this works just like `debugVector`, but instead uses the `OUT` macro. Therefore, this always prints something (unless the vector is empty).

3.3 `secretsharing.h/ secretsharing.cpp`

These files hold declarations and definitions for the classes `ShareTuple`, `AccessPolicy` and `SecretSharing`. The first is a basic class to represent the concept of a share in a secret sharing scheme, and has little functionality. The others are abstract classes, that can not be directly instantiated, and represent the concepts of Secret Sharing scheme and a corresponding Access Policy.

`ShareTuple` This class represents a share. It maintains three pieces of information:

- the share of a given secret, which is a `Big` value;
- the participant that holds this share, which is an integer. In this project, participants of a secret sharing scheme, and consequently attributes of an ABE scheme, are represented by integers.
- an identifier for the share, that makes it unique within a given policy. This is a `std::string` and should hold all the public information necessary for the reconstruction process.

The methods of this class are mostly trivial, allowing access to the internal data, string representation of a query and the basic needs: constructors, copy-constructor, assignment operator and equality testing.

`AccessPolicy` This class represents an Access Policy. Since this is the base class for any such policy, it has only basic functionality. The only internal data is a list of participants, held in a `vector<int>`. Constructors at this level only initialize this participants list in three different ways: with one single participant, with participants numbered from 1 to n, or with an arbitrary list of participants.

It has the following non-trivial functions:

`findCoefficients()` **pure virtual** this function returns the reconstruction coefficients associated with a set of share identifiers of a linear secret sharing scheme. It is a property of such schemes that such a set of coefficients exists and is independent of the share values themselves.

The share identifiers do not have to be all different, and neither have they to correspond to all shares of the policy. Ideally, they should all be compatible with the policy, although checking for that is the responsibility of the concrete class implementing this function. Since the share identifiers contain all public information, this is enough to compute the coefficients.

This function is pure virtual in this project. However, it is possible to give a default implementation using the coefficient extraction algorithm presented in [Crampton,Pinto 14], which means that in absolute terms it should be virtual but not purely so. That algorithm is relatively efficient, but much slower than a dedicated algorithm, since it requires a linear number of share distributions and reconstructions. Keeping this function pure virtual at least forces any new secret sharing scheme to implement a proper dedicated method.

`getNumShares()` **pure virtual** this function returns the number of shares generated by this policy.

`evaluateIDs()` **pure virtual** this function receives a list of share identifiers and decides whether they satisfy the policy. If so, it returns a minimal set of shares that allow reconstruction of the secret.

`obtainCoveredFrags()` **pure virtual** this method receives a list of participants (`vector<int>`) and returns two lists of indices into key and ciphertext fragments (also `vector<int>`) and a list of identifiers of the corresponding shares.

Firstly, this function clearly intrudes in the concept domain of ABE, as it is aware that it must return indices into lists of fragments. The argument names reflect this position, as they suggest "attributes" instead of "participants". The reason it ended up in `AccessPolicy` and not `KPABE` is that

its function requires to verify all shares in the policy which presupposes knowledge of it.

A better design, which I did not have time to implement, would be to have instead a function in `AccessPolicy` that returned a list of share identifiers and participants in a given order, and then extract `obtainCoveredFrags()` to `KPABE`, passing it that list of identifiers. As it is, though, each policy implements its own traversal directly and makes these checks on the fly which, at least, has the benefit of being faster.

Now, the purpose of this function is to use the first argument, with a list of participants, representing the attributes present in a ciphertext, and identify which shares match those participants (which key fragments correspond to attributes in this list). For each match, return in the outbound variables: the index of the share (it is assumed that key fragments submitted by a decryptor are in exactly the same order as the shares are traversed here), the index of the participant that matched (covered) the share, and the identifier of the share. All these vectors should therefore have the same size, and the data in the same indices should relate to the same share.

This matching is needed to perform the pairing operation in the decryption process.

`evaluate()` this function receives a list of `ShareTuple` instances and decides whether they satisfy the policy. The default implementation, present in `secretsharing.cpp`, creates a new vector with the share identifiers of the received shares and calls `evaluateIDs()`. After receiving the result, it uses the witness share identifiers to select the witness shares to return to its environment.

Pure virtual functions are not defined in this class, and must forcibly be implemented by any *concrete* class descendant of `AccessPolicy`. They are so specific to the concrete implementation that it is impossible to give code for them at this level.

`SecretSharing` This class represents a generic secret sharing scheme.

3.4 kpabe.h/ kpabe.cpp

3.5 BLcanonical.h/ BLcanonical.cpp

3.6 tree.h/ tree.cpp

3.7 ShTree.h/ ShTree.cpp

4 Comparison of Canonical BL and Shamir Tree schemes

One of the fundamental thesis of [Crampton, Pinto 14] is that canonical BL can be implemented with much less effort than Shamir Tree, reducing the possibility of bugs and therefore possible security problems of the resulting scheme. I could witness this first hand during this project. Indeed, the canonical BL implementation was much simplified by the fact that its policy can be represented by a simple list of minimal sets. This makes the evaluation of the policy and the secret reconstruction particularly easy: one can linearly traverse the minimal sets and until one is found that is satisfied by the attributes present in the reconstructing set of shares. This can be done, at its most basic, by testing for each element of the target minimal set whether it is contained in the reconstructing set. This, again, can be done by a linear search. Other techniques can be used when the size or the number of sets is too large, for example using hash tables indexed by individual set elements, and containing as values other hash tables classified in the same form. This would have large space requirements but the structure would be easy to populate with sets (with a polynomial time overhead) and would provide a very fast search of a satisfied minimal set, if any.

In contrast, the policy of a Shamir Tree requires a tree like representation, or some complex way to linearize such a structure. The evaluation is recursive in nature, although it can be made iterative. However, this requires extra code for adequate structures, much more careful implementation and is a larger drain of mental resources for the coder. In this project, that was reflected in the need to create a dedicated data structure, the corresponding tests. Different ideas had to be tested for the tree representation, until an option was made for the use of smart pointers to adequately manipulate the tree. The effort for the programmer was much higher than needed for the implementation of the canonical BL version: I estimate to have spent at least twice or thrice as much time to implement the `ShTreeSS` and `ShTreeAccessPolicy` classes as for the corresponding canonical BL versions, plus the time needed for the tree imlementation. The main effort was in the procedures to reconstruct the tree with a bottom-up approach and in the procedure to compute the reconstruction coefficients. The reason for this approach is an optimization suggested in GPSW06 for decryption in the ABE scheme. The naïve way to decryt is to first pair all key fragments with corresponding ciphertext fragments, and then work up the tree as these are combined. However, typically not all leaves are necessary in a successful reconstruction. By identifying a minimal set of leaves first, we avoid

doing unnecessary pairing operations.

A top-down approach would also be possible. The simplest case would traverse the tree in a depth-first manner, but unsatisfied leaves could only be identified at the lowest level. An alternative would be to use the information about the position of the necessary shares to curtail the traversal and only follow down paths that will be needed later. This however requires extra logic to keep track of the current position in the tree traversal, to search for the relevant information in the share public information and given the recursive nature of this descent would probably require a stack to reconstruct the value of a node from values held by descendants and that have not been computed yet. Altogether, it would be probably more complex than the current solution.

Besides the bottom-up reconstruction, I also implemented another variant, that could actually be used for the generic `SecretSharing`: compute the reconstruction coefficients with `findCoefficients` and then linearly combine these with the shares. Although the code for computing the coefficients could be similar to the initial reconstruction method, the implemented version uses two hash tables and a possibly longer running time to obtain a simpler, easier to understand (and maintain) code. After the difficulty of implementing the first reconstruction, I found this method preferable. However, I have not done tests to measure the relative efficiency of both methods.

4.1 Implementation via Monotone Span Programs

It can be argued that the spirit of the schemes in GPSW06 was to use Monotone Span Programs (MSP) to directly reconstruct the secret, instead of evaluating the tree. This is debatable, as the authors proposed the aforementioned optimization in their conference version where there is no mention of MSPs. Furthermore, their decryption scheme does not separate the reconstruction of the secret from the computation of the plaintext. Nevertheless, I try to decouple the two below by giving a secret reconstruction from their decryption algorithm. This requires a representation of the LSSS by a Monotone Span Program, that is, a matrix M and a naming function ρ. M contains a line for each share given by the scheme.

Given these, the reconstruction implicitly outlined in GPSW06 would take the following steps:

1. identifying a minimal set of necessary shares α: γ;
2. identifying a sub-matrix of M, M_γ with only the lines corresponding to that set of shares;
3. computing the coefficients for each share in γ;
4. for each line of the matrix M_γ:
   (a) compute the product between every line and a fixed vector of random data;

(b) multiply the result by the corresponding coefficient;

5. add the results for all the lines

6. add the components of the resulting line

This is in fact very similar to the present implementation: steps 1, 3, 4b, 5 are included part of the generic reconstruction process given above. Steps 2, 4a are the implicit computation of the share, done in the key generation phase; 6 finally corresponds to adding all the different shares together.

It is known that any generic distribution algorithm uses a fixed set of random data and can create each share by computing independent linear combinations of these data. The matrix M can then be seen as the collection of random coefficients of these linear combinations. The implementation in this project computes these shares in the key generation phase by invoking the secret sharing distribution algorithm, possibly using a more efficient algorithm than directly computing the linear combinations. But the reconstruction ends up doing exactly the same operations of this GPSW06 proposal, unless the secret sharing scheme's reconstruction allows for a simpler logic, *as is precisely the case of the canonical BL*, where no coefficients have to be computed, or a possibly cheaper method, which might be the case of the first reconstruction algorithm for Shamir Tree. In any case, this implememtation allows the possibility of relying on a black box implementation of a secret sharing scheme, and possibly a more efficient implementation of the distribution function.

It should be noted that for decryption in the KPABE scheme, we actually need to obtain the reconstruction coefficients α themselves. But this is precisely the point the authors of GPSW06 do not address: such calculation is dependent on the concrete secret sharing scheme. And this gives a clear comparison between canonical BL and Shamir Tree: for the first, the coefficients of all shares in a minimal necessary set are all 1; for the latter, these require some tree-like evaluation. The difference in complexity and logic effort is evident.

5 Testing Environment

All the classes and main functions used in this project are tested, albeit in an indirect way in the case of abstract classes. The testing code is consistent all across the project. Each testing file first begins with an `#includes` call to the relevant header file, where the tested code is declared. Then, a `main()` function does some preparation, if necessary, and calls the function `runTests`. For `testkpabe.cpp`, `testBLcanonical.cpp` and `testShTree.cpp`, this preparation includes setting up the MIRACL library and some randomization. Furthermore, `kpabe.cpp` runs the function `runTests()` twice, each taking an instance of `KPABE` with a different secret sharing scheme but with the exact same policy. Before each call to `runTests`, it prepares the `KPABE` instance by running its `setup()` function.

The function `runTests` might take more or less arguments, depending on what is needed by the individual tests, but it has some invariants: it always

initializes an variable `errors` to 0 that counts how many tests have failed so far, and it always returns this value. Inside, `runTests` usually calls other functions that run specific tests and return their own count of errors. These functions, do all the heavy lifting, preparing and executing each test. One test corresponds to one call to the utility function `test_diagnosis`, which takes three parameters: the message to output to the terminal signalling that a test was called; the condition that the test must satisfy, and the current error count. This function increments the counter if the condition fails, and outputs a message to the terminal saying that the test has either passed (in green) or failed (in red).

Finally, `runTests` always calls the utility function `print_test_result` that shows how many tests have failed in the total package. If all tests pass, with colour coding for quick perception of the result.

6 Performance Measurement

One of the objectives of this project is to test relative performance of canonical BL and Shamir Tree for policies represented in canonical form. The gathering of data for this analysis is implemented in file `benchmark.cpp`. It is used to obtain time measurements for each of the basic operations of a KPABE scheme. This file is used to create four different variants, for two different types of secret sharing scheme and two different ways of generating ciphertext and key fragments in the KPABE scheme.

The specific configuration chosen is dictated by the constants that are defined, and these are declared in pairs of auxiliary header files. The choice of how to form attributes is defined in file `atts.h`, which is indirectly included via file `kpabe.h`. This is only relevant when the corresponding object file `kpabe1.o` or `kpabe2.o` is linked to the compilation of `benchmark.cpp`. Which one is used is dictated by the appropriate `make` command. The alternative files `kpabe1.o` and `kpabe2.o` are also created by appropriate `make` commands, that ensure that `atts.h` is created in the proper way: for `kpabe1.o`, `atts.h` is a copy of `atts.h_1` and analogously for `kpabe2.o`. The file `atts.h` can be deleted at will as long as the files `atts.h_1` and `atts.h_2` are not, since the `make` command will always reconstruct the former.

The other regarding the type of secret sharing scheme to use is done by a similar scheme, via the header file `benchmark_defs.h`. This is a copy of either `benchmark_defs_bl.h` or `benchmark_defs_sh.h` and the appropriate `make` command will choose which of these files supplies the contents of `benchmark_defs.h`. This file will then include the appropriate header file for the chosen secret sharing scheme and define the constants `SS_TYPE` and `SS_ACC_POL_TYPE` that simply replace the name of the classes for a `SecretSharing` and an `AccessPolicy` respectively.

The first task of `main()` is to setup the MIRACL library and seed the randomization process. Then, it invokes `parseInput` to check which switches the command was invoked with. These define what operations should be measured. Although there are four basic operations, there are 13 switches, corresponding

to 12 specific measures plus the switch `all`, which runs all of them.

The reason of this discrepancy is that there are 5 variants each of key generation and decryption. Setup and Encryption are straightforward operations, but the others require a policy and *a priori* different kinds of policies might have an influence in the time necessary for the operation. Since it is impossible to test all kinds of policies, I have chosen 5 kinds that are defined by two parameters: the maximum number of shares in the policy (referred in the code as leaves, since these are the leaves of the policy tree. Note that a canonical BL can also be represented as a tree, although this is not the most efficient representation) and another parameter k of varying purpose. All the policies generated are naturally represented in canonical form.

Types of policies:

Uniform in this policy, the leaves are grouped in minimal sets of equal size. This size is determined by the parameter k.

Linear in this policy, the size of minimal sets follows a linear progression, where the ratio is set by the parameter k. The first set has only one element.

Exponential in this policy, the size of minimal sets follows a geometric progression, where the ratio is formally set by paramenter k, but which is fixed at $k = 2$.

Inverted Linear this policy is just like the linear case, except that the size the sets decreases to 1 instead of increasing from 1.

Inverted Exponential this policy is just like the exponential case, except that the size the sets decreases to 1 instead of increasing from 1.

Each measurement tests 5 main variants of one parameter. In some cases, each variant will have other sub-variants. These are explained below.

The general idea to measure the time of an operation is to obtain from the system the exact time before the operation starts and the time right after it ends. The basic function to return the current time, `time()`, returns a value measured in seconds, which is too large to estimate times of much finer resolution.

An alternative function, `clock()`, measures time in clock ticks. A macro `CLOCKS_PER_SEC` represents the number of clock ticks per second for the system in question. I did some preliminary tests with this function and I did not obtain reliable results, usually receiving half the actual time spent (measured by an external clock). Because of this, I decided to use `time()`, which proved to be consistent with external clocks. My first tests were done with basic operations, implemented in file `basic-benchmark.cpp`. Some of these operations took in the order of microseconds, others in the order of milliseconds, but in any case always less than seconds.

To achieve a proper measurement, I took the strategy or repeating a given operation a sufficiently large number of times (usually at least a few hundred, and when possible in the low thousands). The total time for this batch of executions is then divided by the number of iterations, to obtain the average

operation time. This also has the advantage of smoothing out differences in each iteration by returning an average.

6.1 `measureSetup()`

This function measures the amount of time needed for a setup operation. This operation generates some fixed values, like the key randomness and its public encoding as an element of GT, and the public and private values for each attribute. The number of attributes is the only variable quantity in this process. This function runs 5 scenarios, with the following number of attributes in the universe: 5, 10, 100, 500, 1000. The number of repetitions for each scenario is stored in variable `varRepeats`.

6.2 `measureEncrp()`

This function measures the amount of time needed for an encryption operation. This requires a fixed operation, a multiplication in the pairing target group. Plus, it requires the generation of one fragment for each attribute associated to the encryption.

Again, 5 scenarios are tested, with the number of ciphertext attributes varying through 5, 10, 100, 500, 1000. This measures two ways to do encryption: one is the original encryption method, detailed in the literature since Sahai Waters 05, where the plaintext and ciphertext are elements of the group GT.

The other method is taken from the sample implementation of Fuzzy-IBE in the MIRACL library, where a message is a string of limited size (as dictated by the size of a `Big` instance in MIRACL) which is interpreted as a `Big`. I saw no rationale in the documentation to MIRACL as to why this change should have been made, and so I decided to obtain performance data for each.

Accordingly, each iteration executes two encryptions where only the plaintext varies (and the internal randomness of the encryption, of course). The traditional plaintext is stored in a variable `M` of type `GT`, and is initialized to be a random value. Since MIRACL does not offer a method to obtain a random element of `GT`, this function first computes a fixed element of `GT` by pairing the parameters `P` and `Q`, of types `G1` and `G2` respectively and then raises this to a random `Big` integer.

The MIRACL-like plaintext is instead always the string `"hello world to be encrypted"`. This is transformed into a `Big` and stored in variable `sM`. To do this transformation, we execute first `mip->IOBASE=256`. This parameter tells MIRACL how to convert a `Big` into a string, for example for printing its contents to the screen, or how to initialize a `Big` from a string, by defining the numerical base in which all `Big` are represented. Usually, this value is 16, so that a `Big` is written in hexadecimal form. By using base 256, a `Big` is instead represented as a series of bytes, which allows MIRACL to instantiate an integer from a page of text, simply taking the ASCII code of each letter as the value of the corresponding byte. After this conversion, the code simply sets the base back to the usual 16.

6.3 `measureKeyFunc(...)`

The code tests 5 scenarios, defined by the maximum number of shares in the policy. These are passed in as the array `leavesInPolicy`, and an indication of its size in `nLeavesVars`. The number of repetitions for each of these cases is passed in array `varRepeats`. Each type of policy allows several different variants: for example, we might have a uniform policy with sets of size 1, or 100. The general code does not know how many policies it should produce, now with what parameters. Instead, it uses data and functions passed in from the outside to do this:

`makePolicy` is a pointer to a function that generates a policy. Whatever this function is, it receives two parameters represented by variables `nLeaves` (which is one of the scenarios described in `leavesInPolicy`) and a parameter k. It also returns a policy as a `std::string`, and modifies the value of two outbound variables passed to it: `realNLeaves` and `realNSets`, that indicate, respectively, how many leaves and minimal sets are in the generated policy.

`start_k` is an argument received by `measureKeyFunc(...)` that gives the starting value of the parameter k.

`stop(...)` is a pointer to a function that receives parameters `nLeaves` and `k` and decides either to produce a new policy with these parameters or stop the loop and advance for the next scenario.

`next(...)` is a pointer to a function that receives the parameter `k` and returns the next value of `k`, to use in the next iteration of this loop.

While the function `stop(...)` allows the `for` loop to be executed, `makePolicy` is invoked to return a new policy, which is then used to create a new access policy, corresponding secret sharing scheme and finally KPABE scheme. After initializing the latter, the code runs a batch of key generations.

I now give details on how to produce each policy.

Uniform policy The scenarios for this type of policy have the following numbers of leaves: 8, 32, 128, 512, 1024. The initial value of `k` is 1, and at each step this is multiplied by 8. Therefore, the number of leaves in a set always evenly divides the maximum number of leaves, and so the real number of leaves is always the maximum.

The function `makeKeyUnifPolicy` is used to produce a uniform policy. It does so by creating minimal sets of k consecutive integers starting at multiples of k. No element appears in more than one minimal set. The creation of minimal sets is handled by the function `makeMinimalSet`, which is also called from the functions for other policies.

`makeMinimalSet` This is the function responsible for creating minimal sets for policies, based on two parameters: `first` is the value of the first element to include in the set; `codedLength` indicates how many elements should be in the set.

The parameter `codedLength` can be positive or negative. When it is positive, it simply indicates how many elements to include in the set, and that the elements should be consecutive and increasing from the first element. When it is negative, its absolute value is the number of elements to include in the set. These elements should be consecutive, but *decreasing* from the first element. The order in which the minimal sets are concatenated is also reversed, with larger sets appearing before the previous ones. This has the consequence that when not all elements are used in a policy, those missing are the smallest ones.

Linear policies These policies have the same maximum numbers of leaves as the uniform policies. However, the structure of the minimal sets is different. Each minimal set has `k` more elements than the previous one. The value of `k` starts in 2 and increases by 3 while it stays under 10 (as defined by functions `nextLinPol()` and `stopLinPol`). In practice, it takes values 2, 5 and 8.

The policy only uses as many minimal sets as it is possible to have without exceeding the maximum of the current scenario. This results in irregular values of leaves and minimal sets in the successive policies, achieving the possible maximum only for 1024 leaves when the increment parameter is 2. The first minimal set always has the single element 0. Consecutive minimal sets will have consecutive values of elements without ever repeating one of them. An example policy has the sets $(0), (1, 2, 3), (4, 5, 6, 7, 8), \ldots$.

The inverse linear policies follow the same general logic, except that now the smallest set is at the end of policy and not at the start. Its single element is the largest element in the policy, and no longer 0. Each set has k more elements than the following set, and so the first set is the largest but has the smallest elements. One example policy has the following sets (`nLeaves = 32, k = 2`): $(15, 14, 13, 12, 11, 10, 9, 8, 7), (22, 21, 20, 19, 18, 17, 16), (27, 26, 25, 24, 23), (30, 29, 28), (31)$.

Exponential policies These policies also have the same number of leaves as the other kinds. However, only one value of the parameter `k` is considered: 2. In these policies, the first set always has the single element 0. Consecutive sets have double as many elements as the previous. Since the number of leaves in each set is a geometric progression starting in 1, and the maximum number of leaves is a pure power of 2, the resulting number of leaves is exactly the maximum minus 1.

Like the inverse linear policies, the inverse exponential have the same structure of their non-inverse relative, but their smallest minimal set is at the end and has the highest element. Each minimal set has double the amount and smaller elements than the next set. An example policy is $(16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1), (24, 23, 22, 21, 2$

6.4 `measureDecFunc(...)`

This function measures the decryption policy. As is the case for Key Generation, decryption tests different kinds of policies. The exact same functions used for Key Generation are also used here. Therefore, I will only describe the general behaviour.

This function starts by preparing plain texts to encrypt. As all the other operations, it has 5 basic scenarios for the number of leaves in the policy. These are passed in array `leavesInPolicy`, whose size is passed in `lvsInPol`. However, since encryption can take a variable number of attributes, each of these basic scenarios has 5 further sub-scenarios, where the number of attributes in the cipher text varies through 10, 50, 150, 600, 1024.

Before any of the scenarios begin, this function prepares two plaintexts, exactly as described in `measureEncrp`. It measures the decryption of both plaintexts, testing an eventual difference between the two encryption/decryption techniques.

Then it starts an outer loop that represents the basic scenarios. For each of them, it prepares policy variants according to the functions passed as arguments. After getting an adequate policy, it creates a set of elements that are authorized to decrypt the ciphertext, that is, that contains some minimal set. The way to create such a set depends on the particular policy used and the main work is delegated to the function pointed to by `findMiddleSetSizeAndFirstElement`, which returns the first element of the set in the outbound variable `begin` and the intended size of this set, as the return value.

Finally, given that for a given scenario the average execution varied significantly according to the size of this decrypting set, stored in variable `minSetSize`, the true number of repetitions is scaled for each policy, by dividing `varRepeats[i]` by `minSetSize`.

At this point, a new loop is started to run the timed batch of operations for each sub-scenarios.

Creating the decrypting set of attributes Each type of policy creates a different kind of decrypting set, using a different function. These functions receive three parameters: the maximum number of leaves, the real minimal sets in the policy, and the parameter `k` used to produce the policy. The functions' return value is the size of the decrypting set to be produced, while an outbound variable records the first element of that set.

`middleSetUnifPol` This function is used for uniform policies. It returns a size equal to that of the minimal sets in the policy. For the beginning element, it first finds the index of the set occupying the middle position among all sets, if their number is odd (first set's index is 0) or of the first set after the middle point, if their number is even. Then, it returns the element occupying the first position of this set. Thus, these parameters allows the exact reconstruction of some minimal set of the policy.

`middleSetLinPol` This function is used for non-inverted linear policies. As for uniform policies, it computes the index of the middle set or the one after the middle point. It computes the exact size of this minimal set, with help of the arithmetic progression ratio used to build the policy in the first place. Then, it computes the first element in that set.

By construction, the first element of the set with index i is the total of elements in all the previous sets. And since these increase linearly, the size of the i^{th} set ($i \geq 0$) is $1 + i \cdot k$, for some increment k. The first element of the i^{th} set is then: $\sum_{j=0}^{i-1}(1 + jk) = i + i(i-1)k/2$.

Thus, this function also reconstructs exactly a minimal set in the policy.

`middleSetLinPolInv` This function is used for inverted linear policies. It starts by computing the index of the middle set. The logic to compute the size and the first element, however, is more complicated. Given that the linear progression is in reverse, the smallest set, with size 1, has index `nSets - 1`. Therefore, the size of the i^{th} set is now $1 + (nSets - 1 - i) \cdot k$.

In each set, the first element is the largest element that has not been used in any smaller set. The last element is the first element of the previous set plus one. The elements on the set don't have to be ordered, so this function computes the value of the smallest and therefore last element in the set.

The largest element is the maximum number of leaves minus 1, and so is always known. Then, the smallest element in set i is this largest number minus the sum of the sizes of all sets with index equal or larger than i up to `nSets - 1` minus 1. Since the size of the last set is precisely 1, this sum reduces to

$$\sum_{j=i}^{nSets-2} (1+(nSets-1-j)\cdot k) = (nSets-1-i)\left(1 + k \cdot (nSets-1) - k \cdot (nSets-2+i)/2\right).$$

This produces exactly one of the minimal sets in the policy.

`middleSetExpPol` This function is used for exponential policies. Like its counterparts, it starts by computing the index of the middle set. The size of the set is simply computed as the power of k with that index as exponent. The first element of set i is the count of all elements that came before, which corresponds to the sum of all powers of k before i. This is $(k^i - 1)/(k - 1)$.

This produces exactly one of the minimal sets in the policy.

`middleSetExpPolInv` This function is used for inverse exponential policies. The index of the middle set is computed in the same way. The size of the corresponding set obeys the same exponential function, only the exponent is instead the difference between the index of the target set and that of the first set.

The value of the smallest element in the i^{th} set now is equal to the maximum number of leaves minus the sum of all powers of k from exponent 0 to `nSets-1 - i`. This sum is $(k^{nSets-i} - 1)/(k - 1)$.

This produces exactly one of the minimal sets in the policy.