Location-Privacy and Mobility Meter (LPM^2):

Quick Start Guide

Reza Shokri[1], Vincent Bindschaedler, George Theodorakopoulos, George Danezis, Jean-Pierre Hubaux, and Jean-Yves Le Boudec

Location-Privacy and Mobility Meter (LPM^2) is an object-oriented tool developed in C++ that enables the designers of location-privacy preserving mechanisms to evaluate the effectiveness of their mechanisms with respect to various location-inference attacks. This document provides the reader with a quick start guide on how to use the tool to construct the adversary's background knowledge and to run different attacks such as localization and tracking on some location traces. The tool is available online through `http://icapeople.epfl.ch/rshokri/lpm`

Version: 0.14a   Date: July 9, 2012

[1]Contact: *reza.shokri@epfl.ch*

Most people are now equipped with smart phones with many sophisticated sensors and actuators closely related to their activities. Each of these devices is usually equipped with high-precision localization capabilities, based for example on a GPS receiver or on triangulation with nearby base stations or access points. In addition, the environment is more and more populated by sensors and smart devices, with which smart phones interact.

The usage of these personal communication devices, although providing convenience to their owners, leaves an almost indelible digital trace of their whereabouts either in a sporadic or continuous manner, depending on the location-based application which is running on their smart phones.

It is crucial to note that a location trace is not only a set of positions on a map. The contextual information attached to a trace tells much about the individuals' habits, interests, activities, and relationships. It can also reveal their personal or corporate secrets. It can expose the users to unwanted advertisement and location-based spams/scams, cause social reputation or economic damage, and make them victims of blackmail or even physical violence. Additionally, information disclosure breaks the balance of power between the informed entity and the entity about which this information is disclosed.

As a response to these issues, a number of Location-Privacy Preserving Mechanisms (LPPMs) have been proposed during the last decade. However, their assessment and comparison remains problematic because of the absence of a systematic method to quantify them.

In this work, we address these issues by providing a probabilistic framework for modeling and analyzing location privacy of mobile users who make use of different LPPMs to protect themselves against the inevitable threats of using location-based services.

The location privacy of mobile users of location-based services and the success of an adversary in his location-inference attacks on the users' queries are two sides of the same coin. We rely on well-established statistical methods (such as Bayesian inference, hidden Markov model, and Monte-Carlo methods) to formalize and implement the attacks in a tool: the Location-Privacy and Mobility Meter that quantifies the location privacy of mobile users, given various location-based applications and location-privacy preserving mechanisms (LPPMs). The tool is written in C++ and is designed to be used as a static library. New LPPMs can be imported into the tool and their effectiveness on some location traces can be evaluated using the Location-Privacy and Mobility Meter [1, 2].

1 Getting Started

To get started with the tool, extract the archive containing the headers and the pre-built static library file. The main files are the static library file (*libLPM.a*) and the public C++ header file (*Public.h*). *libLPM.a* can be found in the *lib* sub-folder, whereas *Public.h* is in the *include* directory.

2 Framework

Before we get into examples of usage of the tool, we briefly explain some of the main concepts of the simulator and how they relate to the location privacy framework [1, 2].

2.1 Schedules

Schedules are the main abstraction of the tool and, as such, they represent also the basic way through which the simulation to run can be specified. More concretely, schedules are used to capture in a structured way the different stages of the model. Indeed, stages such as the application, the location-privacy preserving mechanism (LPPM), the reconstruction, or the metric are conceptualized as operations which occur in a specific order. For instance, the application stage is an operation whose input is a set of actual traces and whose output is a set of exposed traces. The application stage is typically followed by an LPPM operation whose input is a set of exposed traces and whose output is a set of observed traces. The chaining of the application and LPPM operations is an example of schedule. A schedule can be thought of as a flowgraph whose vertices are operations and which enforces a strict order in its operations. For example, an application operation cannot be directly followed by an attack operation, there must be an LPPM operation in between.

In addition, a schedule need not start or end with specific operations, it can, for instance, start directly with an LPPM operation, provided the input is a set of exposed traces. The full linear schedule includes 4 chained operations: application, LPPM, attack, and, metric. There are two possible ways to construct schedules. The first one is by tweaking the parameters of a so-called schedule template. LPM provides sample schedules which represent typical cases. This method allows for the construction of schedules with only a few lines of code. Alternatively, schedules can be constructed through a second, more powerful mechanism, called schedule builders. A schedule builder is an abstraction which allows to customize (almost) entirely schedules. For instance, this enables users provide and use their own implementation of applications, LPPMs or even attack operations. Moreover, schedule builders support advanced features such as branching (e.g. forking the schedule to allow different operations of the same type to run on a common input).

Once a schedule is created, it can be run (any number of times) through the use of the core class of the tool: LPM. The LPM class is given an input and produces one or more outputs according to the schedule it runs.

2.2 Knowledge Construction

A particular case of a schedule (although not recognized as such by the implementation) is the knowledge construction. This operation can be thought of a separate stage whose purpose is to construct the knowledge that the adversary has about the simulated users. This construction typically requires three inputs: a transitions feasibility matrix, a transitions count matrix, and a set of learning traces. The transitions feasibility matrix dictates locations from which and to which simulated users are allowed to move, during one time instance. The transitions count matrix captures transitions of simulated users which are not encoded in learning traces.

The knowledge construction process is done with respect to a specific time partitioning. This time partitioning is a way of taking into account time-dependencies in simulated users' mobility. A time partitioning is a collection of time periods which forms a partition of the observable time. For example, if the observable time starts at timestamp 1 and goes to timestamp 24, assuming one event per real-world hour, a time partitioning may be composed of three time periods: a morning time period from timestamp 7 to timestamp 11 (representing the time between 7am and 11am); an afternoon time period from timestamp 12 to timestamp 18 (representing the time between 12pm and 6pm); and a night time period from timestamp 1 to timestamp 6, and timestamp 19 to timestamp 24.

3 A Simple Example

We now give a simple example of use of the tool. Before running a schedule on some actual trace (a typical use of the LPM), the adversary knowledge must be constructed. For this, a separate procedure is used. Usually, knowledge construction is run only once but the resulting output knowledge can be used in many different schedules.

3.1 Knowledge Construction

Listing 1 shows a simple example of a C++ program to construct the knowledge out of one (possibly noisy) set of learning trace. The first thing to do is to include the library's public header file (*Public.h*). This is the only header that needs to be included when using the tool. In addition, it may be convenient to use the `lpm` namespace (of the library). This allows to drop the prefix "`lpm::`" that would otherwise be needed when referring to any class or structure defined within the LPM library.

```
1   #include "include/Public.h" // the public header (Public.h) is the only header that should be included when linking with the library
2
3   using namespace lpm; // using the namespace allows to drop the lpm:: prefix in front of classes, structs etc...
4
5   int main(int argc, char **argv)
6   {
7     LPM* lpm = LPM::GetInstance();  // get a pointer to the LPM engine (core class)
8
9     Parameters::GetInstance()->AddUsersRange(2, 5);
10    Parameters::GetInstance()->RemoveUsersRange(3, 4); // consider only users 2 and 5 (i.e. {2, 3, 4, 5} \ {3, 4})
11    ull timestamps = 24 * 7; Parameters::GetInstance()->SetTimestampsRange(1, timestamps); // consider only timestamps 1, 2, 3, ..., 168
12    Parameters::GetInstance()->SetLocationstampsRange(1, 8); // consider only locationstamps 1, 2, 3, 4, 5, 6, 7, 8
13
14    Log::GetInstance()->SetEnabled(true); // [optional] enable the logging facilities
15    Log::GetInstance()->SetOutputFileName("output"); // [optional] set the log file name (here: output.log)
16
17    File learningTraceFile("learning.trace", true); // the learning trace file (in the current directory), read-only mode
18    File outputKC("knowledge", false); // the name of the output (the knowledge), write mode
19
20    // simple time partitioning:
21    // week days partitioned into morning (7am - 12pm), afternoon (12pm - 7pm), night (0am - 7am, 7pm - 12am)
22    // weekend days partitioned into a single time period
23    ull dayLength = 24; // time instants in a day
24    ull days = timestamps / dayLength; // number of days
25    const ull weeks = 1; // number of weeks to partition
26
27    TPNode* timePart = Parameters::GetInstance()->CreateTimePartitioning(1, weeks * days * dayLength); // create a time partitioning from timestamp 1 to 168
28    TPNode* week = NULL; TPNode* weekdays = NULL; TPNode* weekend = NULL;
29
30    VERIFY(timePart->SliceOut(0, days * dayLength, weeks, &week) == true); // slice out a week
31
32    VERIFY(week->SliceOut(0, dayLength, 5, &weekdays) == true); // slice out the week days (first 5 days, assuming the first timestamp is on a Monday)
33
34    VERIFY(week->SliceOut(5*dayLength, dayLength, 2, &weekend) == true); // slice out the weekend days (the last 2 remaining days)
35
36    // create time periods for the week days: morning (7am - 12pm), afternoon (12pm - 7pm), night (0am - 7am, 7pm - 12am)
37    TimePeriod morningwd; morningwd.start = 7 * dayLength/24; morningwd.length=5 * dayLength/24; morningwd.id = 1; morningwd.dummy = false;
38    TimePeriod afternoonwd; afternoonwd.start = 12 * dayLength/24; afternoonwd.length=7 * dayLength/24; afternoonwd.id = 2; afternoonwd.dummy = false;
39    TimePeriod nightpart1; nightpart1.start = 0 * dayLength/24; nightpart1.length=7 * dayLength/24; nightpart1.id = 3; nightpart1.dummy = false;
40    TimePeriod nightpart2; nightpart2.start = 19 * dayLength/24; nightpart2.length=5 * dayLength/24; nightpart2.id = 3; nightpart2.dummy = false;
41
42    vector<TimePeriod> periods = vector<TimePeriod>();
43    periods.push_back(morningwd); periods.push_back(afternoonwd);
44    periods.push_back(nightpart1); periods.push_back(nightpart2);
45    VERIFY(weekdays->Partition(periods) == true); // partition each of the 5 week days
46
47    // create time periods for the weekend: a single time period for each day
48    TimePeriod we; we.start = 0 * dayLength/24; we.length=24 * dayLength/24; we.id = 4; we.dummy = false;
49    periods.clear(); periods.push_back(we);
50    VERIFY(weekend->Partition(periods) == true); // partition each of the 2 weekend days
51
52    Parameters::GetInstance()->SetTimePartitioning(timePart); // set the time partitioning
53
54    // print out the time partitioning
55    string str = ""; VERIFY(timePart->GetStringRepresentation(str) == true);
56    std::cout << "Time Partitioning:" << endl << str << endl;
57
58    KnowledgeInput knowledge; // construct and fill in the knowledge input
59    knowledge.transitionsFeasibilityFile = NULL;  // a NULL pointer as transitions feasibility file means users can go from any location to any location
60    knowledge.transitionsCountFile = NULL;  // a NULL pointer as transitions count file means no transitions knowledge not encoded as learning trace
61    knowledge.learningTraceFilesVector = vector<File*>();
62    knowledge.learningTraceFilesVector.push_back(&learningTraceFile); // add a pointer to the learning trace file
63
64    ull maxGSIterations = 100; // do at most 100 iterations of Gibbs sampling per user
65    ull maxSeconds = 30; // and spend at most 30 sec per user (whichever occurs first).
66    if(lpm->RunKnowledgeConstruction(&knowledge, &outputKC, maxGSIterations, maxSeconds) == false)
67    {
68      std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl;  // display the error that occur and exit gracefully.
69      return -1;
70    }
71
72    return 0; // all went well
73  }
```

Listing 1: KC simple example - Main.cpp

In the definition of the main function (entry point of the program), one needs to retrieve a pointer to the `LPM` class (line 7). The latter and many core classes of the library are *singletons*, i.e. there can exist at most one instance of those classes at any given time. Moreover, the user does not need to explicitly create instances of those classes. Getting a pointer to the instance of a singleton class is done by calling the `GetInstance()` static method of the class (the C++ syntax for this is `<classname>::GetInstance()`).

Other singleton classes of the library include `Log` (logging facilities), `Errors` (error reporting) and `Parameters`. The latter is used to define the range of the simulated users IDs, timestamps and locationstamps. For knowledge construction, in particular, the range of the locations and timestamps according to input files needs to be specified(lines 11–12). The set of simulated users also needs to be specified because it is define exactly which users exist (lines 9–10). That is, for each user in the set of simulated user, even if no mobility information is provided, a corresponding mobility profile will be created. Specifying the users set is done using one or more of the `AddUsersRange`, `RemoveUsersRange`, and `ClearUsersSet` methods of the `Parameters` class. These act on the set of users as sets-operations: union, difference, and, empty set, respectively. Simulated users ID, timestamps and locationstamps are all positive integers (`0` is a reserved value, forbidden in input files, used internally to indicate the lack of a user ID, timestamp or locationstamp). Their values are represented using the `ull` type. The latter is an unsigned long long integer (at least 64 bits) used throughout the library to represent unsigned integers.

Before any method of the `LPM` class is run, users may want to enable the logging facilities (lines 14–15). Logging is especially useful for debugging purposes but it is disabled by default. Nonetheless, errors and crashes are written to log files (*errors.log* and *crash.log*) regardless of whether logging is enabled. Errors can at any time be retrieved through the singleton `Errors` class (e.g. the `GetLastErrorMessage()` method returns a human-readable error message). It is recommended to enable logging, unless speed is a critical requirement, in which case disabling logging can significantly improve performance.

Operations of the `LPM` class such as running a schedule or constructing the knowledge of the adversary requires to specify input and/or output files. This is done through a very simple interface mainly composed of the `File` class. A LPM input / output file is created (lines 17–18) by simply providing the name of the file (if it is in the current directory) or a full filepath. For input files, files need to exist but only the filename/filepath is required (the file will be opened in read-only mode). For output files, the files need not exist (although, if they do they will be overwritten). In addition, `false` must be given as second argument to the constructor. This effectively allows the file to be written to (the file is *not* opened in read-only mode).

Knowledge construction is a process which is done with respect to a specific time partitioning, i.e., a way of conceptually splitting the observable timestamps range into time periods. Essentially, time partitioning allows to model time-dependent behavior in the mobility of users. A simple time partitioning is constructed (lines 27–52) which splits the observable timestamps range (1 to 168, conceptually a week) into week days (Monday through Friday) and weekend days (Saturday and Sunday). Each of the week days is partitioned in a morning time period (7am to 12pm), an afternoon time period (12pm to 7pm), and a night time period (0am to 7am, 7pm to 12am). Each of the weekend days is partitioned into a single time period.
More precisely, a time partitioning is created through the `CreateTimePartitioning` method of the `Parameters` class (line 27). This time partitioning starts at timestamp 1 (first argument) and ends at timestamp 168 (second argument), which corresponds to $1 \times 7 \times 24$, i.e., each day is composed of 24 timestamps (i.e. one event per hour), the whole range represents one week (of 7 days). The first 5 days of the week (assumed to start on a Monday) and the last 2 days of the week will be treated differently (lines 32 and 34). This is done through the `SliceOut` method whose last argument returns an opaque object representing the sliced out time range. The first argument represents the starting offset (note the relative indexing that makes offset 0 the first offset) of the slice, while the second argument represents the length (in number of timestamps) of the slice. The third argument is the number of repetitions, i.e., the number of contiguous slices. In the case of the first 5 days of the week, `SliceOut` extracts a group consisting of 5 slices (i.e., days) starting at offset 0, of length 24 each. In other words, the `weekdays` object represents the group of slices whose relative offsets are 0 to 23 (Monday), 24 to 47 (Tuesday), 48 to 71 (Wednesday), 72 to 95 (Thursday), and 96 to 119 (Friday). Note that because those days are grouped, any additional operation on the `weekdays` object will act identically on each day. In particular, this means that each day will be partitioned the same way and with the same time periods.
In lines 37 through 40, time periods for the week days are created. Each `TimePeriod` is a structure composed of four fields: `start` indicates the relative offset of first timestamp of the time period; `length` indicates the length (in number of time instants) that compose the time period; `id`, a unique identifier for the time period; `dummy`, a boolean which is `true` if the time period is a dummy time period, `false` if the time period is user defined. (Dummy time periods are used internally by the library to handle partial time partitions; user-defined time periods are not dummy, so this field should always be set to `false` by user code.) A collection of time periods has to form a partition of the grouped unit it is mean to partition, i.e. in our case a day. This means that every timestamp of the day (i.e. from 0 to 23) should belong to exactly one time period. This is the case here, since timestamps 7 to 11 belong to the `morningwd` time period, timestamps 12 to 18 belong to the `afternoonwd` time period, and timestamps from 0 to 6 and from 19 to 23 belong to the night time period. Note that the night time period is composed of two `TimePeriod` objects which are linked through their `id` fields. Partitioning each of week day is done by the call to the `Partition` method (line 45) whose only argument is a vector of `TimePeriod` objects.
To verify that the time partitioning is defined as intended, a concise description can be printed out using the `GetStringRepresentation` method (lines 55–56). Finally, the time partitioning defined is set as the active time partitioning using `SetTimePartitioning` (line 52).

To construct the knowledge, the `RunKnowledgeConstruction()` method of the `LPM` class is used. This method takes two mandatory parameters and two optional parameters. The two mandatory parameters are pointers to a `KnowledgeInput` structure and to an output `File` respectively. The `KnowledgeInput` should be filled with pointers to the input files of the knowledge construction, namely, files containing the transitions feasibility matrix, the transitions count matrix and one or more files containing the learning traces (lines 21-25). If the transitions feasibility file pointer is `NULL`, the tool assumes that it is possible for simulated users to move from any location to any location (in one time unit). If the transitions count file point is `NULL`, the tool assumes that there is no prior knowledge on transitions, which has not been encoded as learning traces. As for learning traces files, the tool requires at least one, but it is possible to provide any number of (possibly noisy/incomplete) learning traces files. However, every learning trace file will be considered as a separate learning trace. Naturally, a single learning trace file can contain traces for multiple simulated users.

Depending on the parameters and on the learning traces provided, knowledge construction can take a while. Therefore, two optional parameters can be given to the `RunKnowledgeConstruction()` method. Those parameters allow to specify the amount of work and/or time to be spent. The first optional parameter allows to set a maximum number of iterations for the Gibbs sampling procedure for each simulated user. The second optional parameter allows to set a maximum time (in seconds) to spend in the Gibbs sampling procedure for each simulated user. If both optional parameters are provided, the sampling procedure terminates when either the maximum number of iterations or the maximum time to spend has been reached (whichever occurs first). To remove the limit on one of the parameter the constant value `KC_NO_LIMITS` can be used. Naturally settings both parameters to `KC_NO_LIMITS` will result in an error since the knowledge construction would never terminate. It is recommended to allow for 10 or more iterations of the Gibbs sampling procedure for each simulated user.

Running the knowledge construction results in a knowledge output file to be created (if there are no errors). This file can then be subsequently used as the adversary's knowledge to run arbitrary schedules. If all goes well, the `RunKnowledgeConstruction()` method returns `true` and the output file contains the knowledge adversary for all simulated users. Otherwise, `false` is returned and a meaningful error message can be retrieved by calling the `GetLastErrorMessage()` method on the instance of the singleton `Errors` class. It is important to note that if the knowledge is successfully constructed, the active time partitioning will be saved in the knowledge file so arbitrary schedule using this knowledge will not need to define any time partitioning.

3.2 Creating and Running a simple Schedule

Listing 2 shows how to construct and run a simple schedule. The simple schedule is constructed by using a template schedule and tweaking the parameters. The schedule is then executed by the `RunSchedule()` method of the singleton `LPM` class.

Akin to knowledge construction, the library's public header file (*Public.h*) is included and the `lpm` namespace is used. In addition, the range of the parameters (users IDs, timestamps, locationstamps) must also specified. Unlike the other two, a set of users ID (and not a range) is specified. However, the simulated users ID set can be used to filter out some of the simulated users (i.e. the tool will only consider simulated users ID within the allowed range), but, unlike for knowledge construction, not all users define in this set need exist. It is important to note that the locationstamps range should be the same as for knowledge construction (that is, it should agree with the knowledge file). If this is not the case, a warning will be issued and the schedule will be run according to the locationstamps range in the knowledge file. Note that, in particular, this means that knowledge construction must be run for a given a set of users and a given region.

The next step is to construct a schedule. This can be done using either the schedule template method or the schedule builder method. Here we use the former method with the `SimpleScheduleTemplate` class. The simple schedule template models a simple chain of operations starting with the default application operation and ending with the default metric operation.

The `SimpleScheduleTemplate` singleton class is a stateful template which allows some options to be specified. For instance, in Listing 2, the metric type used is specified using the `SetMetricParameters` method. In this case the metric used is `Anonymity` (line 20). We now give a brief overview of the parameters which

```
1   #include "include/Public.h" // the public header (Public.h) is the only header that should be included when linking with the library
2
3   using namespace lpm; // using the namespace allows to drop the lpm:: prefix in front of classes, structs etc...
4
5   int main(int argc, char **argv)
6   {
7     LPM* lpm = LPM::GetInstance();  // get a pointer to the LPM engine (core class)
8
9     Parameters::GetInstance()->AddUsersRange(1, 10); // consider only simulated users ID 1, 2, ..., 10
10    Parameters::GetInstance()->SetTimestampsRange(2, 6);  // consider only timestamps 2, 3, ..., 6
11    Parameters::GetInstance()->SetLocationstampsRange(1, 4); // consider only locationstamps 1, 2, 3, 4
12
13    Log::GetInstance()->SetEnabled(true); // [optional] enable the logging facilities
14    Log::GetInstance()->SetOutputFileName("output"); // [optional] set the log file name (here: output.log)
15
16    // Tweak the template's parameters
17    SimpleScheduleTemplate::GetInstance()->SetApplicationParameters(Basic, 0.3);
18    SimpleScheduleTemplate::GetInstance()->SetLPPMParameters(1, GeneralStatisticsSelection, 0.1, 0.05);
19    SimpleScheduleTemplate::GetInstance()->SetAttackParameter(Strong);
20    SimpleScheduleTemplate::GetInstance()->SetMetricParameters(Anonymity);
21
22    File knowledge("knowledge");
23    Schedule* schedule = SimpleScheduleTemplate::GetInstance()->BuildSchedule(&knowledge, "simple"); // build the schedule
24
25    std::cout << schedule->GetDetailString() << endl; // print a description of the schedule
26
27    File input("actual.trace");
28    if(lpm->RunSchedule(schedule, &input, "output") == false) // run the schedule
29    {
30      std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl; // print the error message
31      return -1;
32    }
33
34    schedule->Release(); // release the schedule object (since it is no longer needed)
35
36    return 0;
37  }
```

Listing 2: Simple schedule example - Main.cpp

can be specified using the `SimpleScheduleTemplate` class. All arguments are optional and if they are not specified, the default values of the parameters will be used instead.

- **Application** parameters can be set using the `SetApplicationParameters` method. The first argument is the application type (`Basic` or `LocalSearch`). The second argument is the probability to expose a given event (for the default application, this probability is independent for each event). The default value is 0.2.
- **LPPM** parameters can be set using the `SetLPPMParameters` method. This method takes 4 optional arguments, the first of which is the obsfuscation level (2 by default, meaning that $2^2 = 4$ locations will be in the obfuscation set). The second argument is the fake injection algorithm (`UniformSelection` or `GeneralStatisticsSelection`). The third argument is the fake injection probability (0.1 by default). The last argument is the hiding probability (0.1 by default).
- **Attack** has only one parameter which can be set using the `SetAttackParameter` method. This argument specifies the *power* of the adversary (`Weak` or `Strong`, the latter is the default).
- **Metric** parameters can be set using the `SetMetricParameters` method. The first argument specifies the type of the metric to use (e.g. `Distortion`, `Entropy`, `Anonymity`, etc.). The default is `Distortion`. If the latter is the metric specified, then a distance function can optionally be given as a second argument. The value `NULL` specifies the use of the default distance function.

After all the desired schedule template parameters have been specified, the schedule can be built by calling the `BuildSchedule()` method. This method takes as first argument a pointer to a `File` containing the knowledge of the adversary (e.g. the output of the knowledge construction of the previous example, see Section 3.1). Additionally, an optional name for the schedule can be specified (second argument, line 23). The function returns a pointer to a `Schedule` object which represent the desired schedule.

This schedule can then be run by the `RunSchedule()` method of the `LPM` (line 28). The method takes three arguments: the first is a pointer to a `Schedule` object, the second is a pointer to an input `File` containing the set of actual traces on which will be the input of the simulation [2]. The last argument is a filename prefix for the output files the simulator will generate. The simple schedule only outputs the result of the metric operation to a file. Other schedule templates or constructing schedule using schedule builders allow to specify additional (intermediary) outputs.

[2]Note that if the schedule does not start with the application stage, but rather with the LPPM or with the attack, then the input file should be a set of exposed or observed traces, respectively.

After the method `RunSchedule()` returns, `Release()` should be called on the schedule object, provided the latter will no longer be used (this frees the corresponding memory). Naturally, exiting the program (returning from the main function) will also free the resources allocated, but if the program does not terminate right away (for instance if other schedules are to be run subsequently) then it is good practice to use the `Release()` method. Once the schedule object has been released, it should not be used any further. Doing so will most likely lead to some sort of crash (e.g. segmentation fault).

4 Extending LPM

The object-oriented library has been designed so that it can be extended easily in various ways. More precisely, it is possible to implement additional versions of every operation that can be used by a schedule. This allows users to control almost entirely the behavior of the tool by overriding specific classes, providing their own implementation of some operations.

4.1 Overriding Operations

Providing a specific implementation for an operation is conceptually simple. In a first step, a new class for the implementation is derived from the base operation class. In this new class, the `Execute()` method is implemented, effectively overriding the base class implementation[3]. In a second step, the default behavior is replaced by providing, usually to a schedule builder or template (or to the LPM singleton), an instance of the newly implemented operation.

A typical case is the implementation of an application or LPPM operation. We explain this procedure below and illustrate it with examples.

4.2 Filter Operations

Application and LPPM operations are different from other operations in that they are filters operations (i.e. they are children of the `FilterOperation` class). A filter operation is an operation whose `Execute()` method handle events from its input trace independently. For this purpose, the `Filter()` method is overriden instead of the `Execute()` method which is implemented only in the base class. Naturally, it is possible to override the `Execute()` method, but implementing only the `Filter()` method is usually sufficient for typical purposes and simpler. In addition to the `Filter()` method, filter operations also contain a `PDF()` method which must be overridden. This method represents the probability density function of the operation. It is critical that the two methods be consistent with each other.

4.2.1 Implementing an Application

Listing 3 is an example of a basic application operation which exposes events independently, with probability `exposeProb`. The implementation is complete because both the `Filter()` and `PDF()` methods are provided with a definition.

`Filter()` is given two input parameters: a pointer to a `Context` object containing the background knowledge, and a pointer to an `ActualEvent`. As output, it must produce an `ActualEvent` object which may or may not be exposed (the `ExposedEvent` class is derived from the `ActualEvent` class). The method returns `false` if it fails (e.g. one of the parameters was a `NULL` pointer) and `true` if it succeeds. In the latter case, the `outEvent` double pointer must point to a `ActualEvent` (or an `ExposedEvent`, is if it exposed) pointer: in particular, it cannot be `NULL`. Moreover, since an application can only decide whether or not to expose an event, the output event must be the same event as the input (i.e. it must contain the same user ID, timestamp, locationstamp).

[3]The latter method is the only method which needs to be overridden.

```cpp
class BasicApplicationOperation : public ApplicationOperation
{
private:
  double exposeProb;

public:
  BasicApplicationOperation(double prob) : ApplicationOperation("BasicApplicationOperation"), exposeProb(prob) {}

  bool Filter(const Context* context, const ActualEvent* inEvent, ActualEvent** outEvent);
  double PDF(const Context* context, const ActualEvent* inEvent, const ActualEvent* outEvent) const;
};

bool BasicApplicationOperation::Filter(const Context* context, const ActualEvent* inEvent, ActualEvent** outEvent)
{
  if(context == NULL || inEvent == NULL || outEvent == NULL)
  {
    SET_ERROR_CODE(ERROR_CODE_INVALID_ARGUMENTS);
    return false;
  }

  double randomSample = RNG::GetInstance()->GetUniformRandomDouble();
  if(randomSample <= exposeProb)
  {
    *outEvent = new ExposedEvent(*inEvent); // expose
  }
  else
  {
    *outEvent = new ActualEvent(*inEvent); // do *not* expose
  }

  return true;

}

double BasicApplicationOperation::PDF(const Context* context, const ActualEvent* inEvent, const ActualEvent* outEvent) const
{
  VERIFY(outEvent != NULL);

  if(inEvent->GetUser() != outEvent->GetUser() ||
    inEvent->GetTimestamp() != outEvent->GetTimestamp() ||
    inEvent->GetLocationstamp() != outEvent->GetLocationstamp())
  {
    return 0.0; // The probability of modifying the event (i.e. the user id, timestamp or locationstamp) is 0.0 ! */
  }

  return (outEvent->GetType() == Exposed) ? exposeProb : 1.0 - exposeProb;
}
```

Listing 3: Implementing an Application example - BasicApplicationOperation.cpp/h

In listing 3, the definition of the `Filter()` method starts by checking the parameters. To determine whether or not to expose the event, sampling from a Bernoulli random variable with $p = exposeProb$ is used. A random sample can be generated using the `GetUniformRandomDouble()` method from the singleton class RNG, which returns a random double in the range $]0, 1[$. If the sample is such that the event should be exposed, an exposed event (copy of the input event) is created. Otherwise, a copy of the actual event is created and provided as output event (the input event could be provided as output event, but it is recommended to create a copy as done in the listing). Note that, in this example, the implementation does not make use of the background knowledge. A more sophisticated application operation may take into account the likelihood of observing the input event in the decision to expose the event.

The `PDF()` function takes three input parameters, the first two being the same as for `Filter()`. The last parameter is a pointer to an `ActualEvent` (or an `ExposedEvent`) object. The method returns a double which represent the probability density function of `outEvent` conditional upon `inEvent` and `context`. That is, the probability to observe `outEvent` given that the associated actual event is `inEvent` and the knowledge is contained in the `context`. Since an application can only expose an event or not, this probability is 0 if `inEvent` and `outEvent` are different (i.e. they have different user ID, timestamp, or locationstamp). If on the other hand, the two events are identical, the probability of exposing `inEvent` is, in our case, `exposeProb`. Otherwise, the probability (of not exposing `inEvent`) is $1 -$ `exposeProb`. To know whether an object of type `ActualEvent` is exposed or not, one can call the `GetType()` method on it. This method returns either `Exposed` or `Actual` (i.e. not exposed). Note that the `VERIFY` macro can be used[4] to ensure a condition is true at runtime: it can be thought of as an assertion mechanism provided by the library.

In the example of listing 3, it is clear that `Filter()` and `PDF()` are consistent, since `Filter()` exposes events with a probability that is consistent with the pdf. Note that for more sophisticated applications, it may be useful to call the `PDF()` method from inside `Filter()` in order to help with the sampling. In such a case, special care needs to be taken to manage the creation (and possible destruction) of `outEvent`, since the latter must exist when calling `PDF()`.

Once the new application has been properly defined, it remains to make the schedule use it (instead

[4]throughout the program, not only when implementing an application.

```cpp
// ...

BasicApplicationOperation* basicApp = new BasicApplicationOperation(0.3); // create an instance with exposeProb = 0.3
SimpleScheduleTemplate::GetInstance()->SetApplicationOperation(basicApp);
basicApp->Release(); // release the App instance => give exclusive ownership of the object to the schedule

// ...
```

Listing 4: Using an newly implemented Application operation - Main.cpp

of the default one). Listing 4 illustrates this procedure when using a schedule template such as the `SimpleScheduleTemplate`. First, an instance of the new application is created and given to the schedule using the `SetApplicationOperation()` method. Finally the `Release()` method is called on the application object. This, unlike for schedules, does not free resources (because the `SetApplicationOperation()` method calls `AddRef()`), but rather gives ownership of the object to the schedule. The three lines of code in listing 4 are meant to replace line 17 of listing 2.

4.2.2 Implementing a LPPM

Despite the differences in purpose between the two types of filters operations, implementing a LPPM is similar to implementing an application. Naturally, for a LPPM, `inEvent` and `outEvent` are pointers to an `ActualEvent` (exposed or not) and `ObservedEvent` object, respectively. Listing 5 provides an example of a hiding LPPM (i.e. there is some probability `hidingProb` of removing the location of an exposed event). In addition to hiding, the LPPM also does basic anonymization by mapping user IDs to pseudonyms using a time-invariant permutation. This anonymization mechanism is provided by the base class of LPPMs: `LPPMOperation`. As a result, very little coding is required to provide this mechanism.

In the implementation of `Filter()`, the output `ObservedEvent` is constructed from the input event. Since this LPPM does not do time obfuscation, the timestamp is directly copied from `inEvent`. Note that `ObservedEvents` contain sets of timestamps and locationstamps. The pseudonym used is the one from the random permutation computed by the base class of LPPMs. The `GetPseudonym()` method allows to retrieve this pseudonym using the user ID as input. There are now two cases depending on whether `inEvent` is exposed. If it is, we determine whether to hide the location by sampling from a Bernoulli random variable with $p = hidingProb$. Here hiding means *not* adding the locationstamp of the input event to the locationstamps set of `outEvent*`. This is because when the output `ObservedEvent` is constructed, its locationstamps set is empty. If, on the other hand, `inEvent` is *not* exposed (i.e. it is `Actual`), we do nothing (we do not add the location of input event).

The definition of the `PDF()` method echoes that of `Filter()` (as it must, since the two must be consistent). Recall that it returns a double which represent the probability density function of `outEvent` conditional upon `inEvent` and `context`. Basically, the implementation starts by looking at whether the timestamp and locationstamp of `outEvent` are different from that of `inEvent`. If they are, the probability is 0. Moreover, since the LPPM can only hide (or not) a location, the probability of `outEvent` having a locationstamp set containing more than a single location is also 0. In other words, according to what the `Filter()` function does, the support of the pdf only contains locationstamps set which are either empty or contain a single location: the same as the locationstamp of `inEvent`. Furthermore, conditional upon `inEvent` *not* being exposed, the locationstamps set must be empty. On the hand other, if `inEvent` is exposed, then the locationstamps set is empty with probability *hidingProb*, otherwise it contains the *true* location with probability $1 - hidingProb$.

Similarly as for applications, listing 6 illustrates the procedure to make use of a newly implemented LPPM when using a schedule template such as the `SimpleScheduleTemplate`. An instance of the new LPPM is first created and given to the schedule using the `SetLPPMOperation()` method. After that, the `Release()` method is called on the LPPM object giving ownership of the object to the schedule. The lines of code in listing 6 are meant to replace line 18 of listing 2

```cpp
class HidingLPPMOperation : public LPPMOperation
{
private:
  double hidingProb;
public:
  HidingLPPMOperation(double prob) : LPPMOperation("HidingLPPMOperation"), hidingProb(prob) {};

  bool Filter(const Context* context, const ActualEvent* inEvent, ObservedEvent** outEvent);
  double PDF(const Context* context, const ActualEvent* inEvent, const ObservedEvent* outEvent) const;
};

bool HidingLPPMOperation::Filter(const Context* context, const ActualEvent* inEvent, ObservedEvent** outEvent)
{
  if(context == NULL || inEvent == NULL || outEvent == NULL)
  {
    SET_ERROR_CODE(ERROR_CODE_INVALID_ARGUMENTS);
    return false;
  }

  VERIFY(inEvent->GetType() == Actual || inEvent->GetType() == Exposed);

  ull pseudonym = GetPseudonym(inEvent->GetUser()); // get the pseudonym for that user from the random permutation

  ObservedEvent* event = *outEvent = new ObservedEvent(pseudonym);
  ull timestamp = inEvent->GetTimestamp();
  event->AddTimestamp(timestamp);

  ull minLoc = 0; ull maxLoc = 0; // get minLoc and maxLoc
  VERIFY(Parameters::GetInstance()->GetLocationstampsRange(&minLoc, &maxLoc) == true);

  ull location = inEvent->GetLocationstamp();

  if(inEvent->GetType() == Actual) // event is *not* exposed (do nothing)
  {
    ; // do nothing
  }
  else
  {
    // event is exposed (either we hide the location, or we do not)

    double randomSample = RNG::GetInstance()->GetUniformRandomDouble();
    if(randomSample <= hidingProb)
    {
      ; // do nothing (i.e. hide the location)
    }
    else
    {
      event->AddLocationstamp(location); // add the original location (i.e. do not hide the location)
    }
  }

  return true;
}

double HidingLPPMOperation::PDF(const Context* context, const ActualEvent* inEvent, const ObservedEvent* outEvent) const
{

  VERIFY(context != NULL && inEvent != NULL && outEvent != NULL);

  ull minLoc = 0; ull maxLoc = 0;
  VERIFY(Parameters::GetInstance()->GetLocationstampsRange(&minLoc, &maxLoc) == true);
  ull trueTimestamp = inEvent->GetTimestamp();

  set<ull> timestamps = set<ull>();
  outEvent->GetTimestamps(timestamps);

  if(timestamps.size() != 1 || timestamps.find(trueTimestamp) == timestamps.end()) { return 0.0; /* This LPPM does not modify the timestamps */}

  ull trueLoc = inEvent->GetLocationstamp();

  set<ull> locs = set<ull>();
  outEvent->GetLocationstamps(locs);

  ull locsInSet = locs.size();

  if(locsInSet >= 2) { return 0.0; } /* This LPPM *never* outputs events with more than one location is the locationstamp set. */

  if(locsInSet == 1)
  {
    ull loc = *locs.begin();

    if(loc != trueLoc) { return 0.0; } // this LPPM can only hide the "true" location, it *never* distorts it
  }

  if(inEvent->GetType() == Actual) // event is *not* exposed (i.e. we have to return pdf conditional upon the event *not* being exposed)
  {
    return (locsInSet == 0) ? 1.0 : 0.0; /* When the event is *not* exposed, the locationstamps set is always empty! */
  }
  else  // event is exposed (i.e. we have to return pdf conditional upon the event being exposed)
  {
    return (locsInSet == 1) ? 1.0 - hidingProb : hidingProb; // hide with probability 'hidingProb'
  }

  return 0.0;
}
```

Listing 5: Implementing a LPPM example - HidingLPPMOperation.cpp/h

4.2.3 Using the background knowledge

In the above examples, neither the application, nor the LPPM used the background knowledge on the mobility of users (the `context` object, parameter to `Filter()` and `PDF()`, was not used). The latter is an object containing a set of user profiles, one per user, encompassing the mobility for that user. Retrieving the

```cpp
// ...

HidingLPPMOperation* hidingLPPM = new HidingLPPMOperation(0.2); // create an instance with hidingProb = 0.2
SimpleScheduleTemplate::GetInstance()->SetLPPMOperation(hidingLPPM);
hidingLPPM->Release(); // release the LPPM instance => give exclusive ownership of the object to the schedule

// ...
```

Listing 6: Using an newly implemented Application operation - Main.cpp

```cpp
// ... this piece of code could be placed inside Filter() or PDF() ...

ull minLoc = 0; ull maxLoc = 0; // retrieve the location parameters
VERIFY(Parameters::GetInstance()->GetLocationstampsRange(&minLoc, &maxLoc) == true);
ull numLoc = maxLoc - minLoc + 1;

ull numPeriods = 0;  TPInfo tpInfo; // get time period parameters
VERIFY(params->GetTimePeriodInfo(&numPeriods, &tpInfo) == true);
ull minPeriod = tpInfo.minPeriod;
ull numStates = numPeriods * numLoc;

ull user = inEvent->GetUser(); // get the user ID
ull loc = inEvent->GetLocationstamp(); // get the locationstamp
ull tm = inEvent->GetTimestamp(); // get the timestamp

ull tp = Parameters::GetInstance()->LookupTimePeriod(tm); // lookup the time period associated with timestamp tm
VERIFY(tp != INVALID_TIME_PERIOD); // make sure that timestamp tm belongs to a valid time period

UserProfile* profile = NULL; // retrieve the profile for user
VERIFY(context->GetUserProfile(user, &profile) == true);

double* transitionMatrix = NULL; // retrieve the transition matrix
VERIFY(profile->GetTransitionMatrix(&transitionMatrix) == true && transitionMatrix != NULL);

double* steadystateVector = NULL; // retrieve the steady-state vector
VERIFY(profile->GetSteadyStateVector(&steadystateVector) == true && steadystateVector  != NULL);

ull tpIdx = tp - minPeriod; // compute time period part of rowIndex and columnIndex
ull locIndex = loc - minLoc; // compute location part of rowIndex and columnIndex
ull entryIndex = GET_INDEX(tpIdx * numLoc + locIndex, tpIdx * numLoc + locIndex, numStates); // the GET_INDEX macro takes 3 parameters: rowIndex, columnIndex,

double transitionLikelihood = transitionMatrix[entryIndex]; // transition probability of going from location loc (in time period tp) to loc (in time period tp

double presenceProbability = steadystateVector[GET_INDEX(tpIdx, locIndex, numStates)]; // steady-state probability of being at location loc (in time period tp

// Tips: sub steady state and transitions vectors can be extracted using Algorithms::GetSteadyStateVectorOfSubChain(), and Algorithms::GetTransitionVectorOfSu
// ...
```

Listing 7: Using the background knowledge

profile of a given user can be done by calling the `GetUserProfile()` on the `context` object. This method returns a `UserProfile` object which encapsulates the transition matrix (a two-dimensional array of doubles of size `numPeriods` · `numLoc` × `numPeriods` · `numLoc`) and the steady-state vector (a one-dimension array of doubles of size `numPeriods` · `numLoc`) for that user.

Listing 7 provides a simple of example of usage of the knowledge. This example computes the probability of being at location `loc` using the steady-state vector, and, the probability of going from location `loc` to location `loc` (i.e. staying at `loc`) in one time instance. First, the `UserProfile` object associated with the user is retrieved from the `context`. The transition matrix and steady-state vector are then retrieved from the `UserProfile`. Finally, the proper indices are computed and the probabilities are retrieved. A chunk of code, similar to the one of this example could for instance be used in the implementation of the `Filter()` or `PDF()` method of an application or LPPM.

4.2.4 Implementation Logic

Implementing new applications or LPPMs is not always straightforward and it helps to follow a well-defined pattern. If the pdf is simple (i.e. it does not involve complex probabilities computations), it is a good idea to implement the `PDF()` method first. This allows, its use as a subroutine in the implementation of `Filter()`. An easy way to implement the `PDF()` method is to think of the conditional part of the pdf (i.e. `inEvent` and `context`) as distinct branches in a tree and implementing each branch separately. This is the approach taken in listing 5. Indeed, after dealing with cases not in the support of the pdf, we have an if-else construct conditional on whether `inEvent` is exposed. Figure 1 illustrate this by a tree representation of the effect of both the application and LPPM described above, on an actual event (u, t, r).

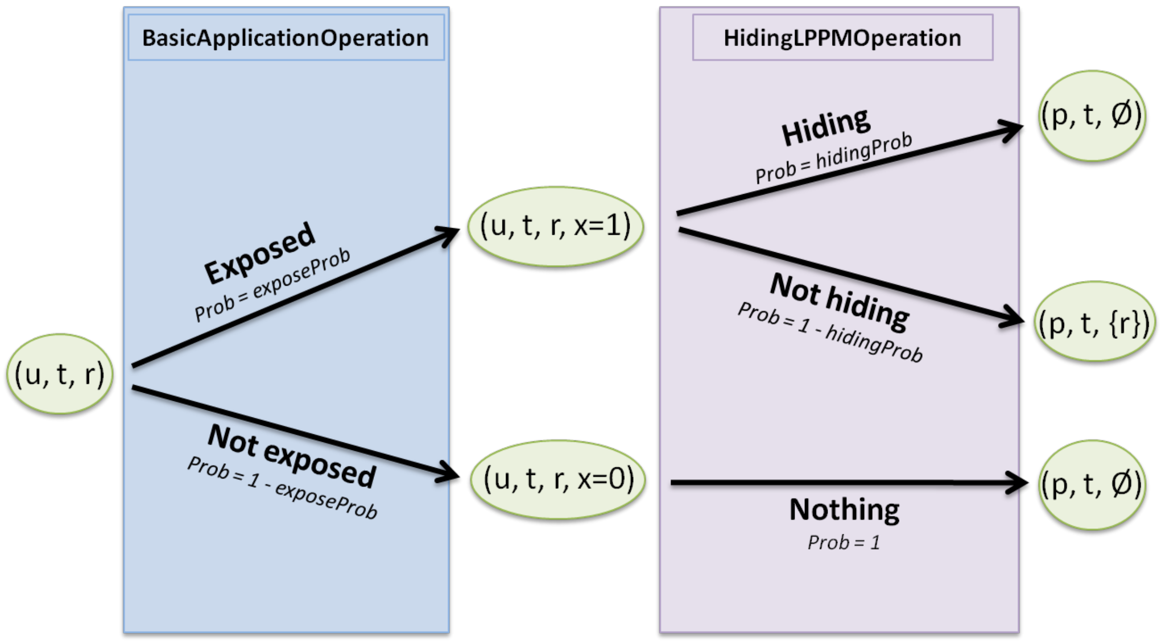

Figure 1: Tree representation of the BasicApplicationOperation and HidingLPPMOperation. (u, t, r) is an actual event with user ID u, timestamp t, and, locationstamp r. $x = 1$ indicates the event is exposed, whereas $x = 0$ indicate it is not. p is a pseudonym for user u, i.e. $p = \sigma(u)$, where σ denotes the anonymization permutation. $\emptyset$ is the empty set.

5 Advanced Features

In what follows, we described the main advanced features offered by the library. Readers with no prior experience using the tool may want to skip this section.

5.1 Custom Schedules

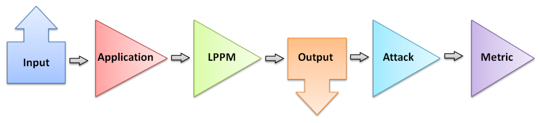

Figure 2: A simple schedule (with intermediary output after LPPM)

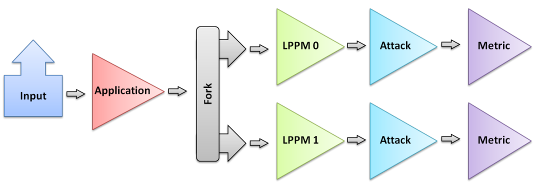

Figure 3: A simple branching schedule

Perhaps, one of the most convenient feature of the library is the ability to customize entirely a schedule. This can be done using the interface provided by the `ScheduleBuilder` class. The latter is a helper class which constructs a schedule step by step according to the information provided by the user. Moreover, schedules builders allow to go beyond what can typically be achieved by constructing schedules through a template (see Section 3.2). Indeed, schedule builders allow partial schedules (e.g. schedules which either do not start with the application stage or do not finish with the metric operation) to be constructed. Furthermore, they provide a convenient way to use the branching feature, one of the most useful construct.

Branching is the process of splitting a schedule (at a given position) into two or more independent subschedules. A schedule which uses branching is in effect a tree, whereas a simple schedule not using branching is a linear sequence of operation. Figures 2 and 3 illustrate the branching feature. More precisely, Figure 2 presents a simple schedule much similar to the one of Section 3.2 (in addition, the one in the figure uses an intermediary output), whereas Figure 3 presents a simple branching schedule which can for instance be used to compare two LPPMs[5].

It is clear that one could also compare two LPPMs using the schedule in Figure 2 twice, once for each LPPM. However, since the framework is probabilistic, two different runs can't be farily compared unless the simulation is repeated a lot of times. This is a strong motivation to use branching schedules such as the one in Figure 3, since when running the latter, the `LPM` instance will run the two branches' LPPM operations on the same input (which is the output of the application operation). In other words, each branch forms a subschedule which is executed independently but uses the input of its parent schedule.

5.1.1 A Simple Custom Schedule

```
#include "include/Public.h"

using namespace lpm;

int main(int argc, char **argv)
{
  LPM* lpm = LPM::GetInstance();

  Parameters::GetInstance()->AddUsersRange(1, 10);
  Parameters::GetInstance()->SetTimestampsRange(1, 5);
  Parameters::GetInstance()->SetLocationstampsRange(1, 8);

  Log::GetInstance()->SetEnabled(true);
  Log::GetInstance()->SetOutputFileName("output");

  // create the builder object (we name the schedule which will be created: "Simple LPPM comparison schedule")
  ScheduleBuilder* builder = new ScheduleBuilder("Simple LPPM comparison schedule");

  // set the default inputs: we only give the knowledge file, for the rest, the default values will be used
  File contextFile("knowledge");
  VERIFY(builder->SetInputs(&contextFile) == true);

  // create and set the application
  ApplicationOperation* application = new DefaultApplicationOperation(0.3);
  VERIFY(builder->SetApplicationOperation(application) == true);

  // release ownership so that the schedule is the only owner of the operation instance
  application->Release();

  // create the LPPMs
  LPPMOperation* lppms[] = { new DefaultLPPMOperation(1, 0.1, UniformSelection, 0.0),
                new DefaultLPPMOperation(1, 0.0, UniformSelection, 0.1) };

  // branch the schedule (one branch per LPPM)
  ushort branches = 2;
  VERIFY(builder->ForkSchedule(branches) == true);

  for(ushort branchIdx = 0; branchIdx < branches; branchIdx++)
  {
    // get the branch's schedule builder
     ScheduleBuilder* branchBuilder = builder->GetBranchScheduleBuilder(branchIdx);

     // get and set the LPPM
     LPPMOperation* lppm = lppms[branchIdx];
     VERIFY(branchBuilder->SetLPPMOperation(lppm) == true);

     // create and set the attack
     AttackOperation* attack = new StrongAttackOperation(); break;
     VERIFY(branchBuilder->SetAttackOperation(attack) == true);

     // set the metric
     VERIFY(branchBuilder->SetMetricType(Distortion) == true);

     // release ownership so that the schedule is the only owner of the operations' instances
     lppm->Release();
     attack->Release();
   }

   Schedule* schedule = builder->GetSchedule();
   VERIFY(schedule != NULL);

  // Free the builder (this essentially severs the tie between the builder and the schedule)
  delete builder;

  File input("actual.trace");
  if(lpm->RunSchedule(schedule , &input , "output") == false) // run the schedule
  {
    std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl ; // print the error message
    return -1;
  }

  schedule->Release(); // release the schedule object (since it is no longer needed)

  return 0;
}
```

Listing 8: Simple custom schedule example - Main.cpp

Listing 8 presents a simple custom schedule example using branching. The schedule implemented is an instance of the one depicted in Figure 3, which can be used to compare two different LPPMs with respect

[5] The `SimpleLPPMComparisonScheduleTemplate` class implements this schedule but additionally allows to compare any number of LPPMs

to a single attack (`StrongAttackOperation`) and metric (`Distortion`).

After the usual include directive, namespace usage declaration, parameters and log configuration, we create a `ScheduleBuilder` object (line 17). The latter is then used to set the properties of the operations of the schedule (input, application, lppms etc.). This *must* be done in order (the same order as in the schedule, see Figure 3). Indeed, first we call `SetInputs` and give a pointer to the knowledge file (named `contextFile`). Using this version of `SetInputs` tells the builder that we want to start the schedule from the beginning (i.e. we start with the input of the application operation), and that we want to use the default values for the input components we did not specify (e.g. `InputOperation`, `LoadContextOperation`). Other versions of `SetInputs` (i.e. overloads) allows to start the schedule at a later stage, and, to specify the instance of the various input components to use.

The `SetXXX` functions of the `ScheduleBuilder` class return a boolean (`true` or `false`) to report success or failure. Here, for conciseness, we use the `VERIFY` macro to check the return value. That said, when building complex schedule, users should not use the `VERIFY` macro, but rather a combination of `if`-constructs and calls to `GetLastErrorMessage()`.

After `SetInputs` successfully returns, calling `SetApplicationOperation()` is allowed and so, we create and set the application operation. Immediately following the application operation, the schedule forks offs into two branches (see Figure 3). Since, both branches must be constructed independently, we create an array with the two LPPMs and use a for loop to iterate over the two branches to construct them. The actual branching occurs when we call `ForkSchedule()` (line 35), the latter takes the number of branches (a `ushort`) as parameter. Note that, `ForkSchedule()` does not take a position parameter to know where to branch off the schedule. Indeed, the position is implicit and equal to the current position maintained by the builder object. In our case, since we just called `SetApplicationOperation()` successfully, the current position is *after* the application stage, that is, *before* the LPPM stage. As a consequence, each branch needs to be constructed starting with the LPPM operation. This is done in the body of the for loop for each branch using the associated schedule builder object (after forking, each branch gets its own independent builder). The latter can be retrieved using the `GetBranchScheduleBuilder()` method. The branch can then be constructed using this builder object (and *not* the main builder object, which we just forked). Note that this builder allows to build schedule with the same features as the main builder, in particular, it is possible to fork off the branch' schedule. As the reader may have noticed, the attack operation is created inside the loop, i.e. it is instantiated once for each branch. While it is possible to use the same object for both branches, it is recommended (unless the performance loss of object creation is a serious issue) to create new objects for separate branches. This will ensure that the reference counting mechanism will work properly.

Once the whole schedule is built, the branch schedules should not be used any more and the schedule builder should only be used to extract the `Schedule` object (using `GetSchedule()`) before being destroyed or re-initialized to build other schedule (this is done using `NewSchedule()`). The `Schedule` object can then be run using the `RunSchedule()` method of the `LPM` class, as usual.

In addition to branching, schedule builder support the creation of partial schedules (i.e. schedules which either stop before the metric operation, or, start after the application operation). The logic in constructing such schedules is straightforward, as is illustrated by the next examples.

5.1.2 A Simple Partial Schedule

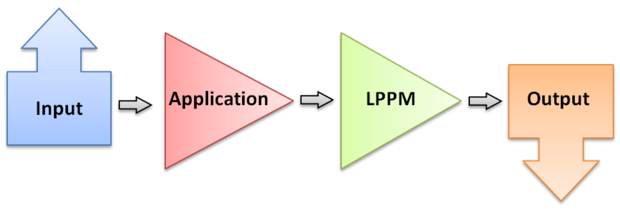

Figure 4: A partial schedule (“Stop-After-LPPM”)

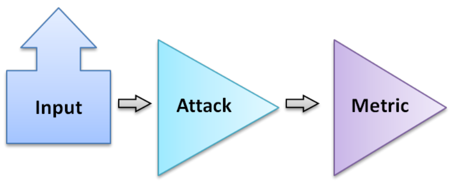

Figure 5: Another partial schedule (“Start-After-LPPM”)

```
1   #include "include/Public.h"
2
3   using namespace lpm;
4
5   int main(int argc, char **argv)
6   {
7     LPM* lpm = LPM::GetInstance();
8
9     Parameters::GetInstance()->AddUsersRange(1, 10);
10    Parameters::GetInstance()->SetTimestampsRange(1, 5);
11    Parameters::GetInstance()->SetLocationstampsRange(1, 8);
12
13    Log::GetInstance()->SetEnabled(true);
14    Log::GetInstance()->SetOutputFileName("output");
15
16    // create the builder object
17    ScheduleBuilder* builder = new ScheduleBuilder("Stop-After-LPPM schedule");
18
19    // set the default inputs: we only give the knowledge file, for the rest, the default values will be used
20    File contextFile("knowledge");
21    VERIFY(builder->SetInputs(&contextFile) == true);
22
23    // create and set the application
24    ApplicationOperation* app = new DefaultApplicationOperation(0.5, Basic);
25    VERIFY(builder->SetApplicationOperation(app) == true);
26    app->Release(); // release ownership so that the schedule is the only owner of the operation instance
27
28    // create and set the LPPM
29    LPPMOperation* lppm = new DefaultLPPMOperation(1, 0.1, GeneralStatisticsSelection, 0.1);
30    VERIFY(builder->SetLPPMOperation(lppm) == true);
31    lppm->Release(); // release ownership so that the schedule is the only owner of the operation instance
32
33    VERIFY(builder->InsertOutputOperation() == true); // insert intermediary output
34
35    Schedule* schedule = builder->GetSchedule(); // retrieve the schedule right now (=> schedule stops after LPPM).
36    VERIFY(schedule != NULL);
37
38    // Free the builder (this essentially severs the tie between the builder and the schedule)
39    delete builder;
40
41    File input("actual.trace");
42    if(lpm->RunSchedule(schedule , &input, "output") == false) // run the schedule
43    {
44      std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl ; // print the error message
45      return -1;
46    }
47
48    schedule->Release(); // release the schedule object (since it is no longer needed)
49
50    return 0;
51  }
```

Listing 9: Simple partial schedule example - Main.cpp

Listing 9 provides an example of a partial schedule which start at the "beginning" (i.e. with the application) but stops after the LPPM operation is completed. That is, the attack and the metric are not used. There are two main reasons to use such a schedule: either we are not interested in the result of the attack and metric operations, or, we are but we want to run the attack and metric operations at a later time. Naturally, stopping after the LPPM is merely an example, one may wish to stop right after the application operation, instead.

Listing 9 shows that partial schedules are built just like other custom schedule using the builder interface. Indeed, we start by creating a builder object which we then use to specify the sequence of operations which represent our schedule. The builder infers that we want to start from the beginning by our use of the first version of the `SetInputs()` method. To tell the builder that we want to end the schedule after the LPPM, we simply call the `GetSchedule()` method at the right "time". Indeed, calling `GetSchedule()` means we are done with the schedule. Since we did not specify an attack operation, the builder infers that we want the schedule to end right after the LPPM operation (the last operation we specified).

We emphasize that application and LPPM operations produce their outputs internally, unlike the metric which writes its output to file. This means that by default, if a schedule ends before the metric operation, no output is written to file. Therefore, if one wants some output (as one should, in such a case), the schedule needs to include an *intermediary output*. Intermediary outputs are optional operations whose output is what they received as input, but which additionally write their input to file. Thus, they can be used to retrieve the result of a previous operation. This is what is done in Listing 9 at line 33: we insert an output operation after the LPPM operation. Note that one can insert at most one output operation per schedule position (the only valid positions for this are before the application operation, before the LPPM operation, and, before the attack operation), and that intermediary output are not limited to be used only before a schedule ends. This means that we could have inserted an output operation before the LPPM operation (that is after the application operation). In particular, we underline that partial schedule which stop after the attack operation (but before the metric operation) have limited usefulness. Indeed, the attack and metric operations are meant to be run "together", one after the other. So, it is possible to construct a schedule which ends right after the attack but since the output of the attack is not meaningful without the metric operation, it is not possible to dump this output to file.

Figure 4 depicts the partial schedule presented in Listing 9.

5.1.3 Another Partial Schedule

The previous partial schedule started from the “beginning” (i.e. the application operation) and stopped before the “end”. It is possible to construct a schedule which does not start from the “beginning”. This is done in our next example, illustrated by Listing 10 and Figure 5.

```cpp
#include "include/Public.h"

using namespace lpm;

int main(int argc, char **argv)
{
  LPM* lpm = LPM::GetInstance();

  Parameters::GetInstance()->AddUsersRange(1, 10);
  Parameters::GetInstance()->SetTimestampsRange(1, 5);
  Parameters::GetInstance()->SetLocationstampsRange(1, 8);

  Log::GetInstance()->SetEnabled(true);
  Log::GetInstance()->SetOutputFileName("output");

  // create the builder object
  ScheduleBuilder* builder = new ScheduleBuilder("Start-After-LPPM schedule");

  File contextFile("knowledge");

  // create instances of the application and the LPPM used, so that we can retrieve their PDFs.
  // note that in this case, we must *not* release ownership of these objects before we're done running the schedule
  ApplicationOperation* app = new DefaultApplicationOperation(0.5, Basic);
  LPPMOperation* lppm = new DefaultLPPMOperation(1, 0.1, GeneralStatisticsSelection, 0.1);

  // retrieve the application and LPPM PDFs
  FilterFunction* applicationPDF = dynamic_cast<FilterFunction*>(app);
  FilterFunction* lppmPDF = dynamic_cast<FilterFunction*>(lppm);

  SchedulePosition startPos = ScheduleInvalidPosition;
  File actualTraceFile("actual.trace");

  // set the inputs: we give the knowledge file, the actual trace file, the application and LPPM PDFs
  // (for the rest, the default values will be used).
  VERIFY(builder->SetInputs(&contextFile, &actualTraceFile, &startPos, applicationPDF, lppmPDF) == true);

  // check that we provided the inputs correctly (i.e. we make sure the builder correctly inferred that
  // we want our schedule to start after the LPPM operation: ScheduleBeforeAttackOperation <=> after LPPM operation)
  VERIFY(startPos == ScheduleBeforeAttackOperation);

  // create and set the attack
  AttackOperation* attack = new StrongAttackOperation();
  VERIFY(builder->SetAttackOperation(attack) == true);
  attack->Release(); // release ownership

  // set the metric type
  VERIFY(builder->SetMetricType(Distortion) == true);

  Schedule* schedule = builder->GetSchedule(); // retrieve the schedule
  VERIFY(schedule != NULL);

  // Free the builder (this essentially severs the tie between the builder and the schedule)
  delete builder;

  // because the schedule starts after the LPPM, the input to RunSchedule is an observed trace !
  File input("observed.trace");
  if(lpm->RunSchedule(schedule , &input, "output") == false) // run the schedule
  {
    std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl ; // print the error message
    return -1;
  }

  schedule->Release(); // release the schedule object (since it is no longer needed)

  app->Release(); // it is now safe to release the application and LPPM instances
  lppm->Release();

  return 0;
}
```

Listing 10: Another partial schedule example - Main.cpp

Schedule not starting from the “beginning” have the particularity that they require more inputs. The reason being that since the input file is not an actual trace file, the corresponding actual trace file still has to be provided (because the metric operation needs it). Moreover, the attack operation uses the application and LPPM pdfs, so when a schedule start after the application operation, the application PDF must be provided. Similarly, when a schedule starts after the LPPM operation, both the application and the LPPM pdfs must be provided. Naturally, the knowledge file also needs to be provided, as always. All of this is specified using the builder’s `SetInputs` methods which comes in several versions (the one used here is not the same as the one used in previous examples, which assumed the schedule to start from the beginning). The different versions are detailed in the reference manual.

In our case, `SetInputs` takes as first parameter, a pointer the knowledge file. The second parameter is a pointer to the actual trace file, while the third is an optional pointer to a `SchedulePosition` object which allows to either specify the starting position (the pdfs provided must be consistent with this), or, to retrieve the start position inferred by the schedule builder based on the provided arguments. The remaining two arguments are the application pdf and LPPM pdf, respectively. If the LPPM pdf is not needed (i.e. if the starting position is meant to be before the LPPM operation), one should pass a NULL as last argument. Note that the application pdf must be specified, since a schedule which does not start at the beginning must start either before the LPPM operation or before the attack operation (starting before the metric operation is not possible for the reasons explained in Section 5.1.2).

A simple way to recover a pdf from an instance of the appropriate filter operation can be done by downcasting into a `FilterFunction` pointer (lines 27-28). It is critical to provide pdfs which are consistent with the input trace. That is, the application pdf and LPPM pdf provided must be identical to the ones which have been used to produce the input observed trace. Otherwise, the behavior is undefined. We emphasize that different parameters values typically lead to different pdfs, this is indeed the case for the default implementations.

Note that since the application and LPPM operations instances used to provide the PDFs are *not* part of the schedule, the latter will not own them. Therefore, they cannot be released as they would typically be. Instead, one must wait until the schedule is no longer in use before calling `Release()` to free them (lines 65-66).

Once `SetInputs` successfully returns, the rest of the schedule can be built as usual. Naturally, the next operation specified must be consistent with the starting position of the schedule. For instance, in our case, we start after the LPPM operation. As a result, the next operation which has to be specified is the attack operation (line 42).

It is interesting to note that Listing 9 can be used to produce an observed trace which can later be processed using Listing 10. Indeed, the former schedule stops where the latter starts and, the application and LPPM pdfs are identical in both cases (since the same application and LPPM operations with the same parameters are used). However, there is little upside in breaking a schedule in two like that. Moreover, it is worth noting that if the goal is for instance to run different attacks or metrics on the same observed trace, using branching is more elegant, more efficient, and, less error-prone. Thus, it would be, in such a case, the preferred method.

5.1.4 Implementing a Distance Function

`Distortion`-based metrics are defined up to a distance function. The default distance function between locationstamp u and v is defined as $1_{u \neq v}$. Users can provide their own distance function by providing an implementation of the `MetricDistance` class. The `MetricDistance` class defines a single method which computes the distance between to locations given as parameters. This method returns a double.

```
// ...

class AbsoluteLocationstampDistance : public MetricDistance
{
  virtual double ComputeDistance(ull firstLocation, ull secondLocation) const;
};

double AbsoluteLocationstampDistance::ComputeDistance(ull firstLocation, ull secondLocation) const
{
  if(firstLocation == secondLocation) { return 0.0; }
  return ABS((double)firstLocation - (double)secondLocation);
}

// ...
// ...

MetricDistance* myDistanceFunction = new AbsoluteLocationstampDistance();
VERIFY(builder->SetMetricType(Distortion, myDistanceFunction) == true);

// ...
```

Listing 11: Distance function implementation

Listing 11 which provides an example definition of a very simple distance function: If $u = v$ the distance is defined to 0. The distance is defined as the *absolute value* (the `ABS` macro) between the two locationstamp values u and v. Note that the distance function defined is meant as an example, it may not define a *good* (or useful) notion of distance between two locations.

In addition to the class definition, Listing 11 contains a code snippet (lines 17-18) which illustrates the use of the newly defined distance when specifying the metric using a schedule builder. Naturally, one can also use one the schedule templates.

5.2 Artificial Trace Generator

The artificial trace generator modules can be thought off as a particular type of schedule. Like any schedule, its behavior can be customized. This is done by overriding the default `TraceGeneratorOperation` and passing an instance of the newly implemented object to the engine.

The default implementation generates actual traces by sampling from the knowledge (i.e. the transition matrix) provided for each user. The generator is invoked using a specific method of the `LPM` class. The method to use is either `GenerateTracesFromKnowledge()` which invokes the default implementation, or `GenerateTraces()` which allows to provide one's own implementation for the generator.

```
1   #include "include/Public.h" // the public header (Public.h) is the only header that should be included when linking with the library
2
3   using namespace lpm; // using the namespace allows to drop the lpm:: prefix in front of classes, structs etc...
4
5   int main(int argc, char **argv)
6   {
7     LPM* lpm = LPM::GetInstance();  // get a pointer to the LPM engine (core class)
8
9     Parameters::GetInstance()->AddUsersRange(1, 4); // consider simulated users ID 1, 2, 3, 4
10    Parameters::GetInstance()->SetTimestampsRange(1, 40); // consider timestamps 1, 2, 3, ..., 40
11    Parameters::GetInstance()->SetLocationstampsRange(1, 20); // consider locationstamps 1, 2, 3, ..., 20
12
13    Log::GetInstance()->SetEnabled(true); // [optional] enable the logging facilities
14    Log::GetInstance()->SetOutputFileName("output"); // [optional] set the log file name (here: output.log)
15
16    File knowledge("artificial.knowledge", true); // the knowledge file to use as input
17    File output("artificial.trace", false); // the output file for the output of the generator (i.e. an actual artificial trace)
18
19    if(lpm->GenerateTracesFromKnowledge(&knowledge, &output) == false) // generate traces using the knowledge provided
20    {
21      std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl;
22      return -1;
23    }
24
25    return 0;
26  }
```

Listing 12: Using the default trace generator - Main.cpp

Listing 12 illustrates the usage of the default trace generator. First, the parameters are set. The users set corresponds to users whose mobility is in the knowledge file provided (named `artificial.knowledge`, line 16). Similarly, the locationstamps range must be consistent with the transition matrices in the knowledge. The timestamps range is used to control the length, start, and, end timestamps, of he generated trace. In our case the output trace will contain for each user 40 events, starting at timestamp 1 and ending at timestamp 40. It will be stored in the provided output file (named `artificial.trace`, line 17). The `GenerateTracesFromKnowledge()` method performs the actual work by reading and constructing the knowledge (input to the generator) and invoking the trace generator.

Using one's own implementation of a trace generator (through the use of the `GenerateTraces()` method) is done through the same procedure as the implementation of any schedule operation. First, the operation (here: `TraceGeneratorOperation`) is overridden by deriving a new class from the base class. After that, an instance of the newly implemented operation is created and provided to the appropriate method of the LPM class (`GenerateTraces()`). The `TraceGeneratorOperation` takes a `TraceGeneratorInput` object as input (users should provided their own implementation of this, if needed) and outputs a `File` containing the generated actual traces.

5.3 Profiling Mobility: Context Analysis Schedules

Independently of location privacy, the tool provides several metrics to quantify specific aspects of mobility. Those metrics can be computed on the mobility profiles of users previously saved in knowledge file produced by the knowledge constructor.

```
1   ...
2
3       ContextAnalysisSchedule* schedule = new ContextAnalysisSchedule("Randomness Analysis Schedule", new RandomnessAnalysisOperation());
4
5       File knowledge("knowledge");
6       if(lpm->RunContextAnalysisSchedule(schedule, &knowledge, "output-filename") == false ) // run the schedule
7       {
8         std::cout << Errors::GetInstance()->GetLastErrorMessage() << endl ; // print the error message
9         return -1;
10      }
11
12  ...
```

Listing 13: Context analysis schedules - Main.cpp

Listing 13 illustrates the usage of a schedule to evaluate the randomness of users' mobility patterns. `ContextAnalysisSchedule` is a special schedule which is constructed using only a schedule name and a `ContextAnalysisOperation` (line 3). `ContextAnalysisOperations` take as input the context from a knowledge file, evaluate specific metrics, and output the result to a user-specified file. To run such schedules the `RunContextAnalysisSchedule` method of the `LPM` class is used (line 6). The first argument is the schedule, the second is the knowledge file to use, and the third is the filename for the output.

Currently, the library provides three such analysis operations:

- `RandomnessAnalysisOperation` evaluates the randomness of the mobility of each simulated user according to the entropy rate metric. The output format consist of one line per simulated user in the format `<user ID>: <entropy rate order 0>, <entropy rate order 1>`. Both entropy rate values are (normalized) floating point values between 0 and 1.

- `PredictabilityAnalysisOperation` evaluates the prediction error in the mobility of each simulated user. The output format consist of one line per simulated user in the format `<user ID>: <prediction error order 0>, <prediction error order 1>`. The operation supports an arbitrary distance function, if none is specified the default distance function is used.

- `AbsoluteSimilarityAnalysisOperation` evaluates the similarity between the mobility of any two pairs of simulated users. The output format consist of one line per pair of simulated users in the format `<user1 ID>, <user2 ID>: <absolute similarity order 0>, <absolute similarity order 1>`. Both similarity values are floating point values between 0 and 1. Note that the metric (for order 1) is not symmetric.

In addition to those, additional operations can be defined through inheritance from the `ContextAnalysisOperation` class.

6 Compiling and Linking

To compile one of the examples given above (e.g. with *gcc*), one needs to specify the path to main directory of the library. For instance, the command: "`g++ -I"<path-to-lib>" -Wall -c -o"Main.o" "Main.cpp"`", will build the file "Main.cpp" assuming that the library archive was extracted to the directory <path-to-lib>.

To link the object file "Main.o" to the static library, use the command: "`g++ -L"<path-to-lib>/lib" -o"Main" ./Main.o -lLPM`". The result will be the executable file "Main".

7 File Format

All files used by the library (input and output) are text files with a specific format. We describe those formats in what follows. Failure to provide the tool with properly formatted files will in most case lead to the program exiting gracefully (and outputting an error). If this does not happen, the behavior is undefined.

7.1 Actual Traces

Actual trace files have one event per line. The format of this tuple is `<user ID>, <timestamp>, <locationstamp>` (three unsigned integers). Listing 14 shows an example: in the third line, simulated user 2, at time 4 is at location 3. Naturally, the same actual trace file can contain events of multiple simulated users (in our case of simulated users 2 and 7). However, the file must be complete and must not contain duplicates entries. This means that the file cannot contain two events for the same simulated user at the same timestamp (even if the two locationstamp are the same). Moreover, if the time parameters have been specified such that all events must be between timestamp t_{min} and t_{max}, then for every simulated users and every timestamp $t \in [t_{min}, t_{max}]$ there must be a tuple. It is also required that all locationstamp be in the appropriate range. Note that, if there is a tuple for some simulated user at some time $t' > t_{max}$ or $t' < t_{min}$, there is no problem: those tuples will be filtered out. Furthermore, there is no need to have a trace for every simulated user whose ID is in the specified range.

```
2, 2, 1
2, 3, 1
2, 4, 3
2, 5, 1
7, 2, 2
7, 3, 4
7, 4, 2
7, 5, 2
```

Listing 14: actual trace file / learning trace file

7.1.1 Exposed Traces

Exposed trace files have the same format as actual trace files, except that the tuple also contains a fourth element. The format of the tuple is `<user ID>, <timestamp>, <locationstamp>, <exposed>`, where `exposed` is a boolean element (i.e. its value must be `0` or `1`) which specifies whether the corresponding event is exposed (`1` means exposed, `0` means not exposed). Naturally, the same rules as for actual trace files about the range of users IDs, timestamps and locationstamps apply here as well.

For convenience, an exposed trace file can be used in place of an actual trace file (but *not* vice-versa), provided *none* of the events are exposed (i.e. the fourth element of every tuple must be `0`).

7.1.2 Observed Traces

Observed trace files have the same format as actual trace files, except that the timestamp and locationstamp elements of each tuple is a set (possibly empty) whose elements are separated by '|'.

7.2 Learning Traces

Learning traces have the same format as actual traces except that they can be only partial traces (some events may be missing). It is also required that locationstamps and timestamps be in the range given by the user.

7.3 Transitions Feasibility Matrix

The transitions feasibility matrix file (used for knowledge construction) contains a boolean square matrix of size $R \times R$, where R is the number of locations considered (if l_{min}, l_{max} define the range of locations, then $R = l_{max} - l_{min} + 1$). A `1` (respectively, `0`) in the matrix entry (i, j) means that it is possible (respectively not possible) to go from location $l_{min} + i - 1$ to location $l_{min} + j - 1$ in one time unit. Listing 15 shows an example of such a matrix with $R = 5$. If $l_{min} = 1$, then it is *not* possible (see lines 2 and 3) to go from location 3 to location 2 (and vice-versa) in one time unit. However, all other transitions are possible.

```
1, 1, 1, 1
1, 1, 0, 1
1, 0, 1, 1
1, 1, 1, 1
```

Listing 15: transitions feasibility file

7.4 Transitions Count

Listing 16 shows an example of transitions count file, which is a list of transitions count matrices (at most one per simulated user) for 4 locations and two time periods. The format is the simulated user ID in a line by itself, followed by an empty line and the corresponding transitions count matrix ($RP \times RP$). P is number of time periods in the active time partitioning. Entry (i, j) of the matrix (for 0 based indices) corresponds to locations r_1 and r_2 and to locations p_1 and p_2 through the equations: $i = p_1 \cdot R + r_1$ and $j = p_2 \cdot R + r_2$. Two empty lines are used as separators between consecutive simulated users.

```
2

2, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 3, 0


7

0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 2, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 3, 0, 0, 0, 1, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 2
0, 0, 0, 0, 0, 0, 2, 0
```

Listing 16: transitions count file

7.5 Metrics and Output Formats

Whenever the output files of the tool correspond to *possible* input files, they have the format described in the previous section. For instance, this means that when the tool outputs an intermediary trace file (e.g. the output of the LPPM, an observed trace), the format used is the same as for input files. This means that an observed trace file produced as output by tool will have the same format as an input observed trace and can therefore be used as such. Another example is actual trace files output by the artificial trace generator, which can naturally be used as input actual trace file for subsequent runs of the tools because the format is the same.

Metrics output files do not correspond to possible input files. We now look at the output format of the metric operations.

There are seven metrics currently implemented in the tool: `Anonymity`, `Entropy`, `Distortion`, `MostLikelyLocationDistortion`, `MostLikelyTraceDistortion`, `Density`, `MeetingDisclosure`. Each metric defines its own output format since metric operations are expected to themselves write their output to file.

That said, similar output format patterns are shared by different metrics. Indeed, many metrics output a set of results for each user. In such a case, the output is composed of `numUsers` rows each with the format (which we call the "user-values" format): `<user ID>: <value 1>, <value 2>, ..., <last value>`, where the number of values is metric-dependent.

A mathematical description of each metric can be found in [1, 2]. Here, we describe the format without going into the specifics of how the results are computed.

- `Anonymity` measures the success of the adversary in de-anonymizing users. The output is in the "user-values" format, with exactly one value per user. The value is 0 if the adversary was able to de-anonymize the user, 1 otherwise (the user's anonymity was preserved).
- `Entropy` measures the normalized entropy of the pdf of the location-privacy of the user at each time instance. The output is in the "user-values" format, with exactly `numTimes` values per user.
- `Distortion` measures the location-privacy of the user at each time instance (according to a distance function). The output is in the "user-values" format, with exactly `numTimes` values per user.
- `MostLikelyLocationDistortion` measures the location-privacy of the user at each time instance (according to a distance function), with respect to the most likely location of the user at each time instance. The output is composed of the most likely location at each time instances. This is in the "user-values" format, with exactly `numTimes` values per user (each value is a locationstamp). Following this, it additionally contains the distortion for each user at each time instance in the "user-values" format, with exactly `numTimes` values per user.
- `MostLikelyTraceDistortion` measures the location-privacy of the user at each time instance (according to a distance function), with respect to the most likely trace of the user at each time instance. The output is composed of the location of the user in the most likely trace at each time instances. This is in the "user-values" format, with exactly `numTimes` values per user (each value is a locationstamp). Following this, it additionally contains the distortion for each user at each time instance in the "user-values" format, with exactly `numTimes` values per user.
- `Density` measures the expected number of users present at a given location at a given time instance. The output is matrix with `numLoc` rows and `numTimes` columns.
- `MeetingDisclosure` measures the expected number of meetings between a pair of users, for all pairs of users. A meeting between two users is when the two users are in the same locationstamp at the same timestamp. The output format is composed of one row for each unique pair of users and one value (the expected number of meetings) for each of those rows. The pair is composed of two user IDs, separated by a comma, whereas the value is separated from the pair by a colon. Note that this metric does not make sense if the tool is run for a single user (since in this case, there cannot be any meetings).

A Archive Contents

A.1 Binaries

Archives containing the binaries contain three main folders: *include*, *lib*, and, *examples*. The first two contain the header files and the library binary file (`libLPM.a`), respectively. The examples folders contain the examples presented in this guide, each in a separate sub-folder. Those examples can be compiled directly using the provided *makefiles* from the directory to which the archive is extracted. That is typing "make" in the examples folder will compile the code (one can use "make clean" to remove the compiled files). Each example can then be executed from within its associated sub-folder by running the binary (i.e. typing "./main"). Note that each sub-folder already contains the required input files (actual trace, knowledge, etc.).

A.2 Source

Archives containing the source code of the library contain three main folders: *include*, *source*, and, *makefiles*. The first two contain the C++ header files and C++ sources files, respectively. The last contain automatically generated makefiles to compile the library. Those makefiles are split into four sub-folders depending on the build type and target architecture as follows: *Debug-32* (debug 32-bits build); *Debug* (debug 64-bits build); *Release-32* (release 32-bits build); *Release* (release 64-bits builds). To build the library (e.g., 32-bits debug build) from the directory to which the archive was decompressed, the following command can be used: "cp -R makefiles/Debug-32 .; cd Debug-32/; make clean; make". The `libLPM.a` binary file will be created in the current directory.

References

[1] Shokri, R., Theodorakopoulos, G., Danezis, G., Hubaux, J.-P., and Le Boudec, J.-Y. Quantifying Location Privacy: The Case of Sporadic Location Exposure. In *The 11th Privacy Enhancing Technologies Symposium (PETS)* (2011).

[2] Shokri, R., Theodorakopoulos, G., Le Boudec, J.-Y., and Hubaux, J.-P. Quantifying Location Privacy. In *IEEE Symposium on Security and Privacy (S&P)* (2011).