User Manual and Developer's Guide

Authors: Martin Burkhart, Dilip Many, Manuel Widmer

`http://www.sepia.ee.ethz.ch`

Version 0.9.1,
May 4, 2012

Contents

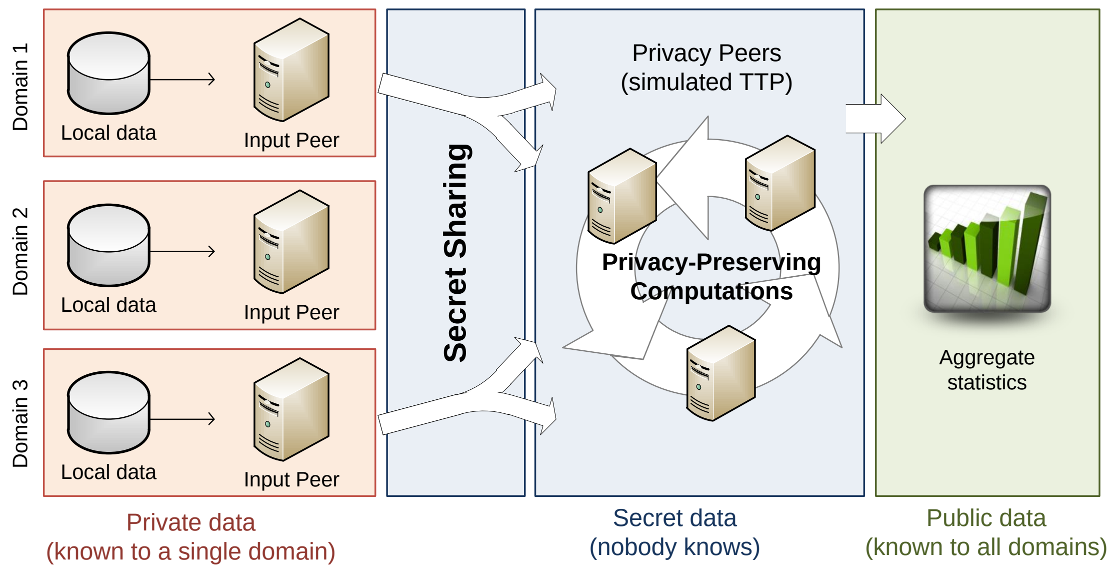


Figure 1: SEPIA deployment scenario. The privacy peers simulate a trusted third party (TTP).

1 Introduction

SEPIA[1] is a Java library for secure multiparty-computation (MPC). It is designed to achieve high performance for parallel execution of private operations.

A typical deployment scenario for SEPIA is depicted in Fig. 1. A number of independent network domains are observed by some type of network probe. The probe could, for instance, produce NetFlow records, SNMP inventory data, or IDS alert logs. Although the individual network data is sensitive and therefore kept secret, the networks could substantially benefit by *aggregating* their data with data from other networks for alert correlation, collaborative anomaly detection, multi-domain traffic engineering, or collecting network performance statistics. For example, they might be interested in whether other networks see similar intrusion alerts or not. Yet, they would never just hand over their alerts. But if a network operator knew that indeed, other networks are attacked by the same group of hackers, they might be willing to cooperate in order to come up with a more effective collaborative defense. What operators could do, is installing SEPIA *input peers* in their premises. Input peers take sensitive local data and share it over the group of SEPIA *privacy peers*. The privacy peers (PPs) together simulate a trusted third party (TTP). That is, each privacy peer gets only one share of each data item. From this share, the PP cannot derive any information about the original data. Only if a majority of PPs comes together and combines their shares, they can reconstruct the information. In addition, the PPs can perform any computation on the shared secrets without reconstructing intermediate values. When the computation has finished, the PPs reconstruct the final result (for instance all intrusion alerts reported by three or more networks) and reveal it to the input peers. These will then use this information for improving network management.

SEPIA's overall architecture is shown in Fig. 2. SEPIA consists of a library and a set of command-line (CLI) tools. The library implements basic operations on shared secrets, such as addition, multiplication and comparison of values. Furthermore, it provides functionality to connect participants of the computation. On top of the basic library, four CLI tools are implemented. These tools implement ready-to-use high-level protocols such as the aggregation of arbitrary network events and the computation of common network statistics.

[1]SEPIA stands for *Security through Private Information Aggregation*.

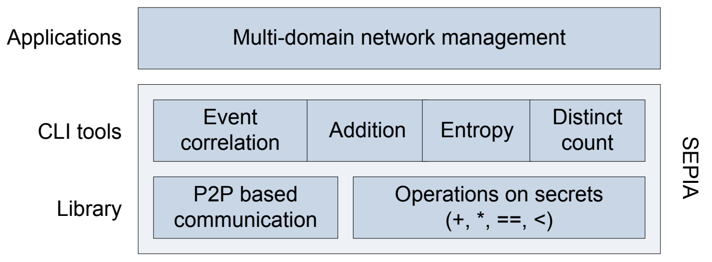


Figure 2: SEPIA architecture overview.

This document is organized as follows: The usage and configuration of the CLI protocols is described in Section 2. To facilitate the generation of configuration files and key material for large numbers of input and privacy peers, we created the configuration editor, which is documented in Section 3. Section 4 introduces programmers to the library with a step-by-step tutorial for creating a new SEPIA protocol. For more theoretical background and a comprehensive performance evaluation of SEPIA, please refer to [2].

2 Running SEPIA protocols

This section introduces eleven basic protocols provided by SEPIA:

Addition:	Allows the addition of arbitrary vector data. Input files could be histograms or a sequence of volume metrics, such as byte or packet counts for different protocols (TCP, UDP, etc.).
Entropy Computation:	Computes the Tsallis entropy (a generalization of Shannon's entropy) of an aggregate distribution. Input for this protocol is a local distribution, e.g., the distribution of the activity of IP addresses.
Unique Count Comp.:	Computes the number of distinct (unique) values of an attribute, e.g., port number or IP address. Input is again a local distribution.
Event Correlation:	Correlates arbitrary events defined by a key and a weight. It reconstructs only the events that are reported by a minimum number of input peers and have a minimum aggregated weight.
Top-K:	Computes the aggregated weight of events defined by a key and a weight to then compile a list of top k events with the k largest aggregated weights. It reconstructs only the top k events.
Set Intersection:	This protocol and the next four are based on bloom filters. It computes the intersection of multiple sets. Input is a set of items that are converted into a bloom filter representation. It reconstructs the bloom filter representation of the intersection set and outputs the elements in the intersection.
Set Union:	Computes the union of multiple sets. It reconstructs and outputs the bloom filter representation of the union set.
Set Intersection Cardinality:	Computes the cardinality of the set intersection.
Set Union Cardinality:	Computes the cardinality of the set union.
Bloom Filter Weighted Set Intersection:	Like the event correlation protocol it correlates arbitrary events defined by a key and a weight. It reconstructs only the events that are reported by a minimum number of input peers and have a minimum aggregated weight. This version uses bloom filters contrary to the event correlation protocol.
Benchmark:	Performs a number of basic MPC operations (multiplications or comparisons) in parallel and reports execution time and data volume statistics. This allows assessing the performance of a specific setup.

All protocols work on a continuous data stream partitioned into time windows. That is, for each time window, input peers expect a separate input file to be dropped in the input directory. If no file is available, they will poll the directory until the expected file arrives. As soon as all inputs are available for the current window, a new computation round is started. When the round has finished, the aggregated output for that time window is written to a file.

Please note that you can find complete sample scenarios for all protocols (except for Benchmark) in the archive `sampleConfigurations.zip`, which can be downloaded from the SEPIA web page. Each scenario contains one folder per input and privacy peer with all the necessary input data, configuration files and startup scripts. All peers are configured to run on `localhost` on a unique port. In order to distribute the peers, just copy the corresponding folder to a remote machine and update the peer's address in all config files (see Section 2.2). For instance, in order to run the scenario for unique count computation, just enter the following commands:

```
$ cd "sampleConfigurations/uniqueCount"
$ ./runAll.sh
```

This will start the input and privacy peers. After a connection discovery phase, the peers perform the unique count protocol for 5 rounds and each input peer stores the resulting aggregated output in its local "output" folder, one file per round. More protocol-specific information, such as input file format and configuration can be found in Section 2.3.

```
# Peer
peers.mypeerid=peer01
peers.minpeers=3
peers.timeout=10000
peers.numberoftimeslots=5
peers.numberofitemsintimeslot=5
peers.activeprivacypeers=pp01:localhost:50001;pp02:localhost:50002;pp03:localhost:50003

# Connection
connection.keystore=peer01KeyStore.jks
connection.keystorepassword=peer01StorePass
connection.keypassword=peer01KeyPass
connection.keystorealias=peer01alias

# MPC
mpc.inputdirectory=input
mpc.outputdirectory=output
mpc.inputtimeout=300
mpc.peerclass=mpc.additive.AdditivePeer
mpc.privacypeerclass=mpc.additive.AdditivePrivacyPeer
mpc.minpeers=3
mpc.field=3775874107000403461
mpc.skipinputverification=false
mpc.maxelement=5400000000
```

Figure 3: Example configuration file for an input peer in the addition protocol. For more examples, see the scenarios in folder ”sampleConfigurations“.

2.1 Starting Peers

The Syntax for starting SEPIA input and privacy peers from command line is as follows:

```
java -cp "sepia.jar:protocol.jar" MainCmd [-options]
```

"protocol.jar" contains the protocol implementation and varies depending on the scenario. For instance, the addition, entropy and distinct count protocols are stored in "statistics.jar". Note that on Linux, the character for separating entries in the classpath option (-cp) is the colon (':') while on Windows it is the semicolon (';'). Make sure to use a Java VM with version at least 1.5. The following options are available:

`-v`	Enable verbose logging mode. This creates quite big log files.
`-help`	Show usage information.
`-p <peerType>`	Specifies the peer type (0: input peer: 1: privacy peer).
`-c <configFile>`	Path to the configuration file.

Thus, an input peer with verbose logging can be started by the following command:

```
java -cp "sepia.jar:application.jar" MainCmd -v -p 0 -c MyConfig.properties
```

Note that more information is needed to configure SEPIA peers. SEPIA reads the detailed configuration from the configuration file (parameter -c). All configuration options are described in the next sections.

2.2 Configuring Peers

SEPIA configuration files are organized as a sequence of ”<key> = <value>“ lines. The key identifies a *property* to set. Fig. 3 shows a complete example of a configuration file for an input peer

Key (`peers.*`)	Description
`mypeerid`	A peer's unique ID (String value).
`minpeers`	The minimun number of *input* peers that must be present before the computation starts.
`timeout`	Before each round, a phase of connection discovery is performed. This is the active timeout (in milliseconds) when waiting for new connections.
`numberoftimeslots`	Number of rounds (timeslots or windows) to process.
`numberofitemsin-timeslot`	Specifies the size of the fixed-length input data vectors. For instance, for complete port histograms this is 65,536.
`activeprivacypeers`	Sequence of all privacy peer addresses in the form <adr1>:<adr2>:...:<adrN>. Each address has the form <ID>;<host>;<port>.

Table 1: Basic parameters for input and privacy peers (`peers.*`).

Key (`connection.*`)	Description
`keystore`	Name of the Java KeyStore (JKS) where the private key and trusted certificates of other peers are stored (used for SSL connections). For details on setting up the KeyStore read Section 2.5.
`keystorepassword`	Password for opening the KeyStore.
`keystorealias`	Alias of this peer's private key in the KeyStore.
`keypassword`	Password for using this peer's private key.
`usecompression`	Enables automatic compression of data (e.g., the shares) sent over connections (`true`/`false`, default: `false`). This makes sense if the field size is much smaller than the datatype holding the shares. Shares are stored and transmitted in `long` variables (64 bits). If the field size is 20 bits, compression saves roughly 2/3 of the data volume.

Table 2: Connection parameters for input and privacy peers (`connection.*`).

participating in the addition protocol. This section describes the general properties for all input and privacy peers. Protocol-specific configuration is discussed in the next section. Properties with a default value are optional, whereas the remaining are mandatory.

There are three kinds of properties. First, basic properties configuring the input and privacy peers with prefix `peers.*` are described in Table 1. Secondly, properties configuring the connection with prefix `connection.*` are listed in Table 2. Lastly, properties configuring the MPC protocol with prefix `mpc.*` are shown in Table 3.

2.3 Protocol-Specific Usage and Configuration

In the following, we describe protocol-specific configuration parameters, as well as input and output file formats for the five protocols. To see the protocols in action with concrete data, please refer to the sample configurations. For entropy and distinct count computation, the examples use input vectors with a size of 65,535 elements, using port histograms as input.

2.3.1 Vector Addition Protocol

The vector addition protocol simply takes a list of integer values as input. All the participating input peers contribute a vector of comma-separated values in each computation round. The goal of the computation is to privately compute the sum of the vectors as shown in Table. 4. That is, no peer learns the input values of other peers, but every peer learns the sum of the input vectors. Each vector element could for instance be an additive metric, such as the local flow or byte count for different protocols and/or directions.

Key (`mpc.*`)	**Description**
`peerclass`	Specifies the class which implements input peer functionality. The class must be available in the classpath. The input and privacy peer classes for the different protocols are listed below: Addition: `mpc.additive.AdditivePeer/AdditivePrivacyPeer` Entropy: `mpc.entropy.EntropyPeer/EntropyPrivacyPeer` Unique Count: `mpc.uniqueCount.UniqueCountPeer/UniqueCountPrivacyPeer` Event Correlation: `mpc.weightedSetIntersection.WsipPeer/WsipPrivacyPeer` Benchmark: `mpc.benchmark.BenchmarkPeer/BenchmarkPrivacyPeer`
`privacypeer-class`	Specifies the class which implements privacy peer functionality. The class must be available in the classpath.
`inputdirectory`	Path to the input directory (default: ”input/“). Only read on input peers.
`outputdirectory`	Path to the output directory (default: ”output/“). Only read on input peers.
`inputtimeout`	The time (in seconds) peers wait for new input files in case no file is available for the next round (default: 300). Only read on input peers.
`field`	Size of the field over which the secrets are shared. This must be a prime number of sufficient size to represent all intermediate computation results (default: $2^{63} - 25$).
`degree`	The degree of the polynomials used for Shamir's secret sharing. Default is $\lfloor (m-1)/2 \rfloor$, where m is the number of connected privacy peers.
`minpeers`	Minimum number of *privacy* peers that have to be present for the computation to start.
`skipinput-verification`	Defines whether input verification should be skipped (`true`) or not (`false`). Default: `false`. The type of input verification that is performed depends on the protocol. If verification is enabled, input peers with non-compliant data are excluded.
`paralleloper-ationscount`	The maximum number of MPC operations that are performed in parallel. For no limit, specify 0 (default: 0).

Table 3: MPC parameters for input and privacy peers (`mpc.*`).

Input Peer 1:	421706,	6393885,	4262205881,	554130,	6522044
Input Peer 2:	517974,	1234433,	7947344550,	345443,	6345454
Input Peer 3:	238220,	5002015,	4899900381,	200033,	7653329
Output:	1177900,	12630333,	17109450812,	1099606,	20520827

Table 4: Example round of the addition protocol.

Table 5 gives an overview of all the protocol-specific properties of the vector addition protocol. If input verification is enabled, it is possible to require each vector element to be below a maximum value. Notice that if too many input peers are disqualified due to invalid inputs, the output file will contain the result vector of the input verification rather than the vector addition result vector (which wasn't computed). These two vectors don't have the same length and can therefore be easily distinguished. Also the log file will contain a warning.

2.3.2 Entropy Protocol

The Tsallis entropy is a generalization of Shannon's entropy that has applications in anomaly detection. The 1-parametric Tsallis entropy is defined as:

$$H_q(Y) = \frac{1}{q-1}\Big(1 - \sum_k (p_k)^q\Big). \tag{1}$$

Key	Description
`mpc.maxelement`	Specifies the maximum value of a vector element that is accepted as valid input (default: 2^{30}). This property is only checked if input verification is not skipped (see property `mpc.skipinputverification` in Table 3).

Table 5: Specific properties: Vector Addition Protocol.

Input Peer 1:	1,	0,	7,	10,	5
Input Peer 2:	0,	2,	5,	0,	0
Input Peer 3:	2,	2,	0,	0,	5
Intermediate (secret):	3,	4,	12,	10,	10
Output ($q = 2$):	0.7573964				

Table 6: Example round of the entropy protocol.

and has a direct interpretation in terms of moments of order q of the distribution. In particular, the Tsallis entropy is a generalized, non-extensive entropy that, up to a multiplicative constant, equals the Shannon entropy for $q \to 1$.

The entropy protocol takes a local distribution as input, which is given in a comma-separated list of item counts, similarly to the addition protocol. In a first step, the input distributions are privately summed to derive an aggregate distribution. Up to this point, the protocol is identical to the vector addition protocol. But after this point, instead of reconstructing the aggregate distribution, the counts for each item k are used to privately derive the probabilities used in (1) and eventually the Tsallis entropy H_q of the aggregate distribution (See [2] for more details). Table 6 shows an example with concrete input and output values. In this protocol, the output is a single value, H_q.

The specific configuration properties for the entropy protocol are listed in Table 7. Since the entropy protocol is built on top of the addition protocol, also the properties from Table 5) apply.

2.3.3 Distinct Count Protocol

The distinct count protocol computes the number of distinct values of an attribute or, in other words, the number of items in the aggregate data with a non-zero count. It accepts local distributions as input but the input peers internally only share a single bit (1 for "item present" and 0 for "item not present"). Table 8 shows a simple example for one round.

The distinct count protocol does not have specific properties. If input verification is enabled, the privacy peers verify that each element shared is indeed a bit-value (0 or 1).

2.3.4 Event Correlation Protocol

The event correlation protocol allows private aggregation of arbitrary network events. An event e is defined by a key-weight pair $e = (k, w)$.[2] This notion is generic in the sense that keys can be defined to represent arbitrary types of network events which are uniquely identifiable. The key k could for instance be the source IP address of packets triggering IDS alerts, or the source address concatenated with a specific alert type or port number. It could also be the hash value of extracted malicious payload or represent a uniquely identifiable object, such as popular URLs, of which the peers want to compute the total number of hits. The weight w reflects the impact (count) of this event (object), e.g., the frequency of the event in the current time window, or a classification on a severity scale.

[2]Note that in the code, the protocol is called *weighted set intersection protocol* and event keys are called features. This is also reflected in configuration properties.

Key	Description
`mpc.entropy-.tsallisexponent`	Sets the Tsallis exponent q used in (1) (default: 2). Currently SEPIA supports integer values greater than 1.

Table 7: Specific properties: Entropy Protocol.

Input Peer 1:	23,	0,	0,	0,	5
Input Peer 2:	0,	0,	0,	7,	0
Input Peer 3:	0,	0,	0,	75,	12
(Intermediate:)	1,	0,	0,	1,	1
Output:	3				

Table 8: Example round of the distinct count protocol.

Each peer shares at most s local events per time window. The goal of the protocol is to reconstruct an event if and only if a minimum number of input peers T_c report the same event *and* the aggregated weight is at least T_w. The rationale behind this definition is that an input peer does not want to reconstruct local events that are unique in the set of all input peers, exposing sensitive information asymmetrically. But if he knew that for example three other input peers report the same event, e.g., a specific intrusion alert, he would be willing to contribute its information and collaborate with the other peers reporting the same event. Likewise, a peer might only be interested in reconstructing events of a certain impact, having a non-negligible aggregated weight. For more details on the protocol, please refer to [2].

For input verification, it is possible to have the privacy peers verify that each input peer reports unique event keys (i.e., it reports each event only once). In addition, the peers can specify a maximum event weight. Each of these checks is enabled by a separate configuration option. The general property `mpc.skipinputverification` is ignored. Table 9 lists the protocol-specific configuration options. The field size (property *mpc.field*) is overridden and determined internally based on the maximum event key (`maxfeature`) and the maximum expected sum of weights (number of input peers times the maximum weight `maxweight`).

For an example of the event correlation protocol please look at the sample configurations. The protocol supports two input file formats selected by the `inputType` property. The protocol parses input files in the `weightByCount` format as follows: Each row contains comma-separated values. The first value in each row is used as the event key. The number of lines in which this key occurs is used as the weight (count) of the event. Input files in the `keyWeightData` format the protocol parses as follows: Again each row contains comma-separated values. The first value in each row is used as the event key and the second is used as the weight (count) of the event. Irrespective of the format, the s events with biggest weights are used as input of the corresponding input peer.

Output files contain one reconstructed event per line with the following format:
```
<event key> <aggregated weight> <peer count> <list of peers reporting the event>
```

2.3.5 Top-K Protocol

The top-k protocol (described in detail in [1]) privately and probabilistically computes the top-k items. An item e is defined by a key, weight tuple $e = (k, w)$.

The input peers create hash tables of a fixed size h. For each item they store the key and weight at the position given by the hash of the key. If local collisions occur, the node only reports key and value of the item with the bigger value. They then share these hash tables.

The PPs aggregate the items of each slot of the hash tables with the same key. In case of different items being in the same slot, a subalgorithm computes the key with the largest weight for that slot.

Key (`mpc.weighted-setintersection.*`)	**Description**
`maxweight`	Defines the maximum acceptable event weight in input verification (default: 256).
`maxfeature`	Defines the maximum value that is used for event keys. (default: 1401085391). If this is big enough to represent the maximum weight sum, it is used as the field size.
`featurecount-perpeer`	The number of events s per input peer and round (default: 10).
`mintotalweight-threshold`	The minimum total weight T_w an event needs to have in order to be reconstructed (default: 100).
`minfeaturecount-threshold`	The minimum total peer count T_c an event needs to have in order to be reconstructed (default: 2).
`featureduplicates-check`	Enables the input verification for duplicate event keys (`true`/`false`, default: `true`).
`maxweightcheck`	Enables the input verification for maximum weights (`true`/`false`, default: `true`).
`inputType`	Selects the input file format (`weightByCount`/`keyWeightData`, default: `weightByCount`).

Table 9: Specific properties: Event Correlation Protocol.

Key (`mpc.topk.*`)	**Description**
`k`	Defines the number of top features to compute.
`s`	Defines the number of hash arrays used.
`h`	The size of the hash arrays.
`seed`	The common seed used for the hash functions.
`maxtau`	The maximum value that is considered in the binary search for the value tau (separating the k-th from the (k+1)-th value).
`inputType`	Selects the input file format (`BasicMetricsBinary`/`TopkKeyWeightData`, default: `BasicMetricsBinary`).

Table 10: Specific properties: Top-k Protocol.

The PPs then reconstruct the k keys with the largest aggregate weights and their weights and send it to the IPs.

Table 10 lists the protocol-specific configuration options. The special parameter `maxtau` indicates the upper bound on the range ($[0, \texttt{maxtau}]$) in which the algorithm searches for the value τ separating the k-th from the (k+1)-th weight value. Adjusting this parameter speeds up the protocol execution. But a too low value will cause the protocol to output an incorrect result!

The protocol supports two input file formats selected by the `inputType` property. For backwards compatibility the default input format is `BasicMetricsBinary`. This file format is used by a proprietary tool for binary port and full IPv4 distributions as input. Usage of the `TopkKeyWeightData` format is recommended. Input files in the `TopkKeyWeightData` format the protocol parses as follows: Each row contains comma-separated values. The first value in each row is used as the event key and the second is used as the weight of the event. (The only difference to the event correlation protocols `keyWeightData` format is that the `TopkKeyWeightData` format only supports integer keys and weights while the former supports longs.)

Output files contain one reconstructed item per line with the following format:
`<key>; <aggregated weight>`

Key (bloomfilter.*)	Description
iscounting	Defines whether counting Bloom filters shall be used (true) or not (false, for binary). (Not used by Bloom filter based weighted set intersection protocol.)
hashcount	The number of hash functions to be used for the Bloom filter.
size	The size of the Bloom filter to be used.

Table 11: Properties used for Set Operation Protocols.

2.3.6 Set Intersection Protocol

SEPIA includes a suite of private set operations protocols based on bloom filters. In these protocols the sets are represented by bloom filters. A bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_r\}$ of r elements is an array BF of s bits initially set to 0. The bloom filter uses k independent hash functions $h_1, \ldots, h_k$ with range $1, \ldots, s$. For each element $x \in S$, the bits at positions $h_i(x)$ are set to 1 for $1 \leq i \leq k$. Counting Bloom Filters (CBF) are a generalization of bloom filters, which use integer arrays instead of bit arrays enabling representation of multisets. (For more details please see [3].)

The set intersection protocol takes as input a file describing a set of items with one set element per line. From the input set the IPs compute their bloom filter representation of their set. The filters are then sent to the privacy peers. The privacy peers then compute the bloom filter representing the intersection set position-wise. That is, for all positions u with $1 \leq u \leq s$ they perform the logical AND:

$$[BF_\wedge(u)] := [BF_1(u)] \wedge [BF_2(u)] \wedge \ldots \wedge [BF_n(u)]. \tag{2}$$

For the multiset case the equation is as follows:

$$[BF_\wedge(u)] := min(min(min([BF_1(u)], [BF_2(u)]), [BF_3(u)]), \ldots, [BF_n(u)]) \tag{3}$$

All positions of $[BF_\wedge]$ are then reconstructed and sent back to the input peers.

The input peers then check for each item of their input set if it is in the intersection set. Each such element gets written to the output file (one per line).

The configuration options of this protocol can be seen in Table 11.

2.3.7 Set Union Protocol

The set union protocol takes as input a file describing a set of items with one set element per line just like the intersection protocol. From the input set the IPs compute their bloom filter representation of their set. The filters are then sent to the privacy peers. The privacy peers then compute the bloom filter representing the intersection set position-wise. That is, for all positions u with $1 \leq u \leq s$ they perform the logical OR:

$$[BF_\vee(u)] := [BF_1(u)] \vee [BF_2(u)] \vee \ldots \vee [BF_n(u)] \tag{4}$$

For the multiset case the equation is as follows:

$$[BF_\vee(u)] := [BF_1(u)] + [BF_2(u)] + \ldots + [BF_n(u)] \tag{5}$$

All positions of $[BF_\vee]$ are then reconstructed and sent back to the input peers.

The output files of the input peers contain the union bloom filter. The first line is a human readable text describing the next three lines. The following 3 lines contain the configurable properties of a bloom filter. The second line is a Boolean indicating if the filter is a counting bloom filter, the third contains the number of hash functions and the fourth the filter length. These lines are directly followed by the content of the filter, one line per position.

The configuration options of this protocol can be seen in Table 11.

2.3.8 Set Intersection Cardinality Protocol

The input of this protocol is the same as for the set intersection protocol. It also first computes the set intersection but without reconstructing the filter representing the intersection set. The PPs take this filter and add up the individual positions (bits or counters). The resulting sum is reconstructed and sent back to the IPs.

The IPs then compute an estimate of the intersection set cardinality. The output text file contains in the first line a human readable text describing the contents of the next three lines. The following lines contain the sum of the input filter positions, the sum of the intersection filter positions and the intersection set cardinality estimate.

The configuration options of this protocol can be seen in Table 11.

2.3.9 Set Union Cardinality Protocol

The input of this protocol is the same as for the set union protocol. It also first computes the set intersection but without reconstructing the filter representing the union set. The PPs take this filter and add up the individual positions (bits or counters). The resulting sum is reconstructed and sent back to the IPs.

The IPs then compute an estimate of the union set cardinality. The output text file contains in the first line a human readable text describing the contents of the next three lines. The following lines contain the sum of the input filter positions, the sum of the union filter positions and the union set cardinality estimate.

The configuration options of this protocol can be seen in Table 11.

2.3.10 Bloom Filter Weighted Set Intersection Protocol

The set intersection protocol takes as input a file describing the key set and their weights with one set element and associated weight per line. The key can be any string not containing a semi-colon terminated by a semi-colon. The weight is an integer right after the semi-colon. The keys are stored in a counting bloom filter CFK_i and the weights are stored in CFW_i by inserting each key as many times as the value of its associated weight. Both filters CFK_i and CFW_i are then shared among the PPs.

From this the PPs compute a bloom filter with each position u satisfying:

$$\begin{cases} F_{WSI}(u) = \sum CFW_i(u) & \text{if } \sum CFK_i(u) \geq t_c \text{ and } \sum CFW_i(u) \geq t_w \\ F_{WSI}(u) = 0 & \text{otherwise} \end{cases} \tag{6}$$

If the weights of the keys satisfying the above requirement shall not be learnt, the resulting filter is computed to satisfy:

$$\begin{cases} F_{WSI}(u) = 1 & \text{if } \sum CFK_i(u) \geq t_c \text{ and } \sum CFW_i(u) \geq t_w \\ F_{WSI}(u) = 0 & \text{otherwise} \end{cases} \tag{7}$$

Key (`mpc.bfwsi.*`)	**Description**
`keythreshold`	The minimum total peer count an event needs to have in order to be reconstructed.
`weightthreshold`	The minimum total weight an event needs to have in order to be reconstructed.
`learnweights`	Defines whether the weights shall be reconstructed (`true`) or not (`false`).

Table 12: Specific properties: Bloom Filter Based Weighted Set Intersection Protocol.

Key (`mpc.benchmark.*`)	**Description**
`operationtype`	Chooses the basic MPC operation to be benchmarked: `multiply`, `equal`, `lessthan`, or `smallintervaltest`. default: `multiply`.
`lowerbound`	Lower bound used for the `smallintervaltest` operation. (default: 1).
`upperbound`	Upper bound used for the `smallintervaltest` operation. (default: 10).

Table 13: Specific properties: Benchmark Protocol.

The resulting filter is then written into a file. The format is the same as for the set union protocol (Section 2.3.7).

The configuration options of this protocol can be seen in Tables 11 (property `iscounting` is not supported) and 12.

2.3.11 Benchmark Protocol

The benchmark protocol is used to assess the performance of parallel MPC operations, such as multiplications, equality, and less-than comparisons. It does not have input and output files, but input data is generated randomly. Statistics for each round are written to the log file. It measures the number of parallel operations per second and the data volume sent over the network.

Table 13 describes the supported configuration options. The number of operations performed is given by the property `mpc.numberofitemsintimeslot`.

2.4 Controlling Privacy of a Computation

In the above sections, we described the parameters `peers.minpeers` for setting a minimum number of *input peers* and `mpc.minpeers` for setting a minimum number of privacy peers. These parameters can be used to control the privacy of an ongoing computation. When a new protocol round starts, the peers perform a connection discovery phase until enough input/privacy peers are available.

Since version 0.9, SEPIA is robust against peer failures[3]. That is, it can continue computation in case input/privacy peers fail and disconnect. When a peer disconnects during an ongoing computation, SEPIA checks if the minimum conditions are still met. If not enough peers are available, computation is stopped.

Number if Input Peers. By requesting a minimum number of input peers, it is guaranteed that the final result is sufficiently aggregated and that individual contributions are “hidden” in the aggregate result. Consider, for example, if you ran the vector addition protocol with just two input peers. Then each input peer can derive the other peer’s input data from the final result and its own input data. If you set the minimum number of input peers to 5, however, this is not possible.

[3]This must not be confused with security against malicious adversaries. In SEPIA, adversaries are assumed to be semi-honest and a failure is not intentional. Failures are, e.g., caused by a server crash or network disruption.

Number of Privacy Peers. Likewise, privacy of input data is better protected if more privacy peers participate in the computation. SEPIA is secure as long as the majority of the privacy peers is honest. If there are only 3 privacy peers, the system is broken as soon as any two collude and exchange their information. However, if we configure 9 privacy peers, a group of 5 privacy peers must collude to achieve the same. So by setting a minimum number of required privacy peers, we can adjust the collusion threshold.

Polynomial Degree. Another parameter useful for controlling data privacy is `mpc.degree`. It controls the degree t of the polynomials used in Shamir's secret sharing. By default, it is set to $\lfloor (m-1)/2 \rfloor$ with m being the number of privacy peers. This setting yields the above privacy properties. In general, any set of $t+1$ or more privacy peers can interpolate the secret. This means, if we set $t = m-1$, then *all* privacy peers must be available for interpolating a secret and no subgroup of colluding privacy peers could break the scheme. While this raises the collusion threshold, it makes private multiplication impossible. For private multiplication to work, $m \geq 2t+1$ must hold [2].

In summary, $t = \lfloor (m-1)/2 \rfloor$ is the highest collusion threshold we can choose if the protocol requires multiplication (or any operation built on top of multiplication, such as comparison). However, if the protocol only requires addition, the collusion threshold can be set arbitrarily.

2.5 Creation of SSL Keys and Certificates

This section describes how key pairs, certificates and symmetric keys can be created. For an SSL connection the server needs to authenticate, whereas client authentication is optional. In SEPIA protocols, both parties need authentication. Several ways are possible for generating private/public key pairs and certificates, e.g., a trusted certification authority (CA) might create certificates.

For reasons of simplicity, self-signed certificates can be used and exported. Each party who wants to communicate must export its public key and certificate and the other party needs to import them as trusted certificates.

The following describes how key pairs and certificates are created and exchanged using the JDK's keytool. A privacy peer (privacypeer01) generates and exports the keys and the certificate and an input peer (peer01) imports it. The configuration editor described in Section 3 automates these steps and generates a full PKI for all the configured input and privacy peers.

NOTE: To reduce the key generation effort for simple tests with many input and privacy peers, all peers can use the same key. The sample configuration for the event correlation protocol contains a KeyStore ("pKeyStore.jks") which trusts its own key. Thus, all input and privacy peers can use the same KeyStore file. Note however, that this is not secure! In production environments, make sure that keys and certificates are properly configured.

Creation of key pairs. Using the keytool a self-signed private/public key pair is generated:

```
keytool -genkey -v
-keystore privacypeer01KeyStore.jks
-storepass privacypeer01StorePass
-alias privacypeer01Alias
-keypass privacypeer01KeyPass
-keyalg RSA -keysize 2048
```

The user will then be prompted to specify his name, organization,...

Export to a certificate file. The self-signed certificate is then exported to a certificate file. This can be given to the other party.

```
keytool -export -alias privacypeer01Alias
-storepass privacypeer01StorePass
-file privacypeer01Certificate.crt
-keystore privacypeer01KeyStore.jks
```

Import trusted certificate. Other parties' certificate can be imported to a local keystore and labeled as trusted. In the example bellow, the certificate of privacy peer 1 is imported to the keystore of privacy peer 2:

```
keytool -import -v -trustcacerts
-alias privacypeer01Alias
-file privacypeer01Certificate.crt
-keystore peer02KeyStore.jks
-storepass privacypeer02StorePass
```

The following command shows a list of all key pairs and trusted certificates stored in a key store:

```
keytool -list -keystore peer01KeyStore.jks
-storepass peer01StorePass
```

Command to Run Programs that Use Keystores. The truststore used can be set by setting the system property:

```
System.setProperty("javax.net.ssl.trustStore", "peer01KeyStore.jks");
```

If, however, several instances run on the same machine (e.g., for testing reasons) the truststore will have to be set in the command line:

```
java -Djavax.net.ssl.trustStore=peer01KeyStore.jks <peer arguments>
```

3 SEPIA Configuration Editor

Creating configuration files and certificates/keystores for large numbers of input and privacy peers can be tedious, especially if various configurations with different parameters need to be evaluated. To facilitate the creation of configurations, we implemented the SEPIA configuration editor. Table 14 gives an overview of the files involved.

3.1 Quickstart

To get started right away just run:

`winStartGUI.bat` on a Windows OS
`startGUI.sh` on a Linux/Unix OS

There are two modes for the configuration editor, one mode is command line only and the second is the graphical user interface. The Syntax is as follows:

```
$ java -jar MakeConfig.jar g|c [some.properties]
```

Filename	Description
config.properties	Main configuration file. Change settings here if you use the tool from command line.
IPhosts.txt	List of all possible input peer hosts
PPhosts.txt	List of all possible privacy peer hosts
MakeConfig.jar	Java archive of this Tool
sampleKeytoolInput.txt	Input file needed if a full PKI is generated
testConfigKeyStore.jks	Default keystore (all peers use the same private/public key)
startGui.sh	Shell script to start the GUI version of the tool
winStartGUI.bat	Script to start the GUI version on Windows
.sepia_logo.png	Logo needed for the about dialog
sepia.jar	Sepia Library (must be named exactly this way)
yourprotocolhere.jar	The jar file of your custom protocol (can have any name)

Table 14: Files required by the configuration editor.

The parameter "g" selects the graphical interface whereas "c" chooses command-line operation. The last parameter specifies the configuration file to be used.

By default "config.properties" is used to configure the settings. The default configuration generates a set of 5 input and 3 privacy peers running the tutorial protocol (see Section 4). There are many comments in the file to help you with the configuration. You can also specify another config file to be used. The command would then look as follows:

```
$ java -jar MakeConfig.jar c someOther.properties
```

3.2 Graphical User Interface

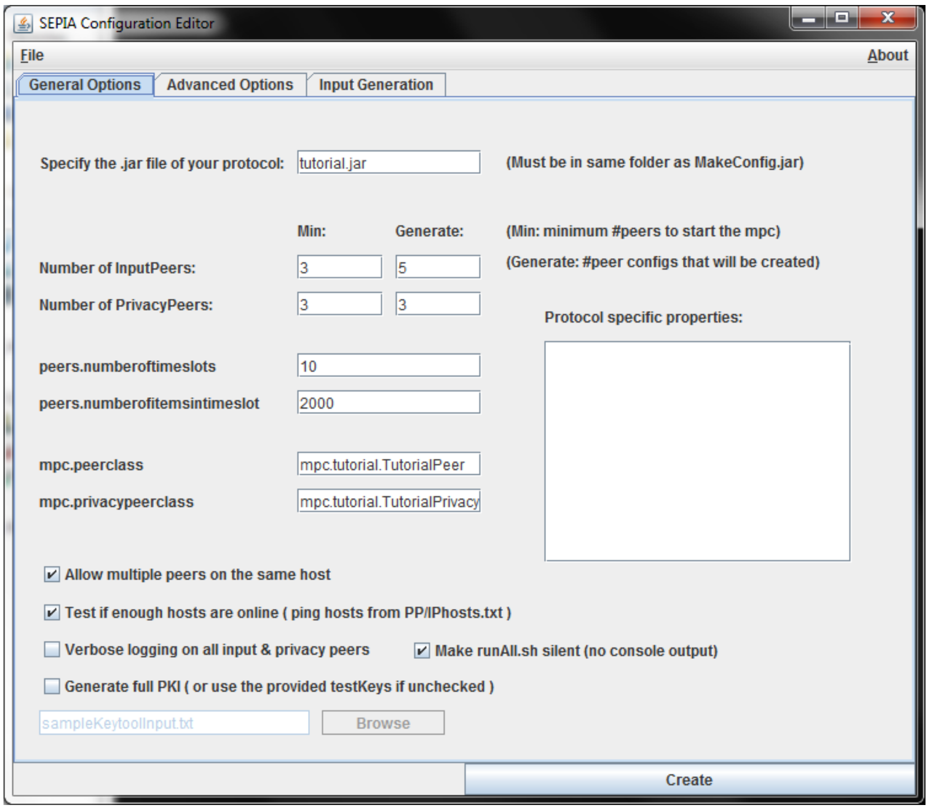
Figure 4 shows a screenshot of the GUI. The file menu allows saving and opening different configurations. To generate a configuration click the "Create" button at the bottom of the window. The "Advanced Options" tab allows you to configure additional properties. The "Input Generation" tab allows the automatic generation of random input data for protocols. The number of files produced will match the "peers.numberoftimeslots" property and there will be "peers.numberofitemsintimeslot" entries in one file. There are 2 options for generating input:

1. "Set Generation" will produce sets of strings formatted like IPv4 addresses and separated by newlines.
2. "Random Numbers" will just produce random numbers which are separated by the specified delimiter.

All Options of the GUI can also be directly changed in the "config.properties" file.

Protocol-specific properties When you open a configuration with protocol-specific properties they won't be displayed in the GUI, but they will be considered for the configuration. There is no need to enter them again. If you need to change some just enter the key/value pair(s) into the provided box and the loaded property will be overwritten when creating or saving the current configuration.

Full PKI If you choose to generate a full PKI you need to make sure that the keytool of the java installation is in the path. The configuration editor will read input for the keytool from the provided input file. Please look at the comments in that file for further instructions on the required input. Unfortunately, keytool is language-dependent, so the very last input of the tuple which is a



Specify the .jar file of your protocol: tutorial.jar (Must be in same folder as MakeConfig.jar)

	Min:	Generate:
Number of InputPeers:	3	5
Number of PrivacyPeers:	3	3

(Min: minimum #peers to start the mpc)
(Generate: #peer configs that will be created)

peers.numberoftimeslots 10
peers.numberofitemsintimeslot 2000
mpc.peerclass mpc.tutorial.TutorialPeer
mpc.privacypeerclass mpc.tutorial.TutorialPrivacy

Protocol specific properties:

[✓] Allow multiple peers on the same host
[✓] Test if enough hosts are online (ping hosts from PP/IPhosts.txt)
[ ] Verbose logging on all input & privacy peers
[✓] Make runAll.sh silent (no console output)
[ ] Generate full PKI (or use the provided testKeys if unchecked)
sampleKeytoolInput.txt

Figure 4: Screenshot of the graphical user interface of the SEPIA configuration editor.

confirmative "yes" must be supplied in the correct language (e.g., "ja" for the German version) or the keytool will hang. Every host will then have his own private key and a keystore where the public keys of the other hosts are saved. Passwords for keystores and keys can be found in the config file ("final.properties") of the respective configuration.

3.3 The Generated Configuration

Table 15 shows the files produced when the configuration was successfully created.

Filename	Description
hostnameIP/PP	folder containing the files below:
→ input	folders containing input data of an input peer
→ final.properties	properties that will be used to start the peer
→ perfomance.sh	script to log top during runtime (CPU & memory consumption)
→ startup.bat	Windows script to start the peer
→ startup.sh	Linux script to start the peer
distributeConfigs.sh	Script to copy the configurations to the corresponding hosts
extractStatistics.sh	Script to extract information about the protocol runtime from log files after a run
RunAll.bat	Script to start a (localhost)config on Windows
runAll.sh	Script to start a configuration on Linux systems (distributed or local)

Table 15: Files generated by the SEPIA configuration editor.

First you need to distribute the configuration to all the hosts. If you are working in a Linux environment use "distributeConfigs.sh" for that purpose, it will automatically copy all necessary files to the correct hosts using SCP. Else you need to copy the files sepia.jar, yourprotocol.jar, testConfigKeyStore.jks, .toprc, *ConfigurationFolder* to the corresponding hosts.

Starting a Configuration To start a configuration you can use the runAll.sh script. On Windows RunAll.bat will only work if all peers run on the same computer (localhost). Another way to get your protocol running is to start each peer manually using startup.bat or startup.sh in the configuration folder.

4 Tutorial: Create Your Own Protocol

SEPIA distinguishes between input and privacy peers. In this section, the term *peer* will refer to both input and privacy peers as a collective term. Basic operations on Shamir shares are implemented in separate classes in the package `mpc.protocolPrimitives.operations`. There are more than a dozen operations, including the most important ones, namely multiplication, equality testing and less-than comparison. For a complete overview of the different classes please refer to the Javadoc. These operations are used via the `Primitives` class. To make usage easier the `mpc.protocolPrimitives` package provides some helper classes. This section explains how to implement an MPC protocol using all these classes.

The general starting procedure of a SEPIA protocol is as follows: The `PeerStarter` class creates, initializes, and runs the configured peer instances. Then, the instances connect to other peers and perform the implemented MPC protocol.

SEPIA uses the Observer design pattern to handle messages and to signal important events, e.g., final results or exceptions. Upon receiving a network message, a protocol thread sends a notification (which contains the message) to its observers. The peer (observer) then handles the message. In the following, a *message* is received over a network connection, while a *notification* is sent by observables to their observers.

SEPIA protocols are separated into several classes. For example, in the `mpc.tutorial` package we have the following main classes:

1. TutorialPeer
2. TutorialPrivacyPeer
3. TutorialProtocolPeer
4. TutorialProtocolPrivacyPeerToPeer
5. TutorialProtocolPrivacyPeerToPP
6. TutorialMessage

A peer class (TutorialPeer) in general holds the state of an input peer and starts the protocols between itself and the privacy peers. A privacy peer class (TutorialPrivacyPeer) stores the state of a privacy peer and starts the protocols between the privacy peer and the other input or privacy peers. The three protocol classes (TutorialProtocolPeer, TutorialProtocolPrivacyPeerToPeer, and TutorialProtocolPrivacyPeerToPP) simply contain the run functions that start the protocol steps in the appropriate order. Each of the protocol classes governs a different part of the communication. The first protocol class (TutorialProtocolPeer) handles the communication of the input peer with a privacy peer. At the other end, the second protocol class (TutorialProtocolPrivacyPeerToPeer) handles the communication of the privacy peer with an input peer. Lastly, the third protocol class (TutorialProtocolPrivacyPeerToPP) handles the communication between two privacy peers. For each type of connection, the corresponding instance is run in a separate thread. This minimizes synchronization delays and allows the parallelization of the communication and local computation for each peer. Fig. 5 illustrates the role of the different classes. Note that input peers do not directly communicate with each other.

The methods implementing the protocol steps are mainly implemented in the peer classes, because they require access to the state information and sometimes need to be synchronized among the different protocol threads. Additionally, a message class (TutorialMessage) holds data to be sent between input and privacy peers, such as the initial shares or the final computation result.

The following steps are required to implement a new protocol:

1. Create the input peer class.

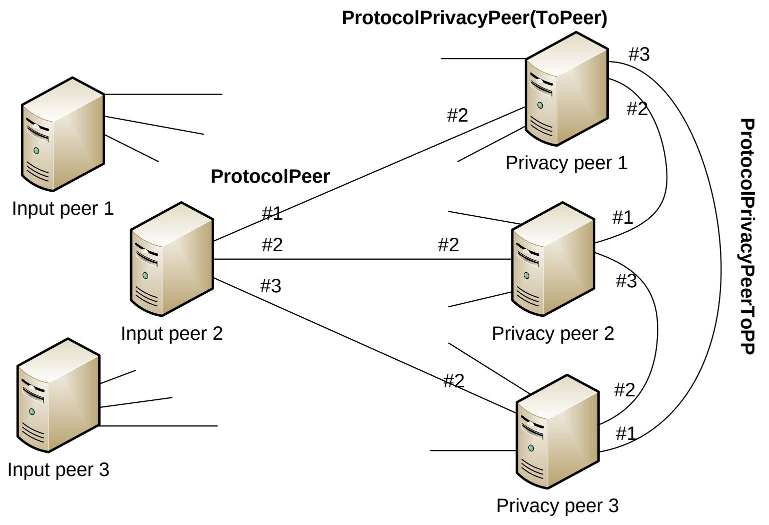


Figure 5: Different protocol classes in action. Each line corresponds to a socket connection. Each protocol instance is run in a separate thread. That is, a privacy peer runs one thread for the communication with each input peer and each privacy peer.

2. Create the privacy peer class.
3. Create the protocol classes.
4. Create a new message class for exchanging the initial shares, the final computation result, and any other protocol-specific information between the input and privacy peers.
5. If your protocol needs to store more information about the peers than the standard PeerInfo class, derive a new one extending PeerInfo.

In this tutorial you will build an MPC protocol where the input peers (a, b, c, ...) share inputs $(a_1, a_2, a_3, ...; b_1, b_2, ...)$ with the privacy peers and the privacy peers compute the products of the inputs $(a_1 * b_1 * c_1 * ..., a_2 * b_2 * ...)$. You should be able to complete this tutorial within 1-2 hours. The complete code of the tutorial is available in the `mpc.tutorial` package. Also, the sample configurations contain a pre-configured set of input and privacy peers running the tutorial protocol. Alternatively, you can use the default settings of the SEPIA configuration editor (see Section 3) to generate configuration files for 5 input and 3 privacy peers running the tutorial protocol.

At some points in this tutorial your IDE (or compiler) might complain that some classes (or their methods) are unknown. To resolve these problems just add the appropriate imports (e.g: `import events.FinalResultEvent`).

4.1 Basics

The following pages guide you through step-by-step instructions that develop a full-blown protocol starting from the stub classes in the `mpc.skeleton` package (see "tutorial.jar"). All code fragments shown are available in the corresponding classes of the `mpc.tutorial` package.

Step 1: Base Classes First choose a name for your protocol, e.g., Tutorial. The package for the tutorial protocol classes should then be named "tutorial". The class names should start with "Tutorial". Copy the classes from the `mpc.skeleton` package to your own package and rename them appropriately.

Afterwards do case sensitive replacements (or renaming with refactoring) within all files:

- "Skeleton" > "Tutorial"
- "skeleton" > "tutorial"
- "SKELETON" > "TUTORIAL"

Step 2: Starting the protocol Your new protocol implementation will likely reside in a separate jar file. Therefore, you need to tell SEPIA how to instantiate the input and privacy peer classes. This can be done by configuring the peer classes in the config files:

```
mpc.peerclass=mpc.tutorial.TutorialPeer
mpc.privacypeerclass=mpc.tutorial.TutorialPrivacyPeer
```

Make sure that the sepia.jar and the classes `TutorialPeer` and `TutorialPrivacyPeer` are available in the classpath. To include all the necessary files in the classpath you can use the '-cp' option of the java VM and then start each SEPIA peer via the `MainCmd` class:

```
$ java -cp "sepia.jar:tutorial.jar" MainCmd ...
```

Your new protocol is already running, but at this point it does not do anything useful yet.

Step 3: Variable Initialization To track the state of the computation and hold local data, we need some variable definitions. Add the following definitions to the `TutorialPeer` class:

```
/** indicates if the initial shares were generated yet */
private boolean initialSharesGenerated = false;
/** array containing the input data of this peer for all time slots; format: [inputIndex] */
protected long[] inputData = null;
/** array containing my initial shares; dimensions: [numberOfPrivacyPeers][numberOfItems] */
private long[][] initialShares = null;
```

Then, add the following variable definition to the `TutorialBase` class:

```
/** contains the final results */
protected long[] finalResults = null;
```

To initialize these variables, add the following code to the `initializeNewRound` method of the `TutorialPeer` class (only shown in parts):

```
initialSharesGenerated = false;
initialShares = null;
finalResultsToDo = numberOfPrivacyPeers;
finalResults = null;
```

The privacy peer class also needs a variable definition:

```
/** number of initial shares that the privacy peer yet has to receive */
private int initialSharesToReceive = 0;
```

These variables are initialized in the `initializeNewRound` method of the privacy peer class (only shown in parts):

```
// init counters
initialSharesToReceive = numberOfInputPeers;
finalResultsToDo = numberOfInputPeers;
finalResults = null;
```

4.2 Read Input Data, Generate, Send and Receive Shares

Step 4: Read Input Data Besides the configuration data your protocol certainly needs some input data. Therefore you should implement the `readDataFromFile` method in the input peers class to read the input data from a file. For our tutorial protocol we just add some code to generate the data instead of reading it from a file:

```
inputData = new long[numberOfItems];
for(int inputIndex = 0; inputIndex < numberOfItems; inputIndex++) {
  inputData[inputIndex] = inputIndex;
}
return true;
```

Step 5: Generate and Send Shares The input peers have to generate Shamir shares of their inputs. For that we add one method to generate and one to retrieve the initial shares in the `TutorialPeer` class:

```
public synchronized void generateInitialShares() {
  if(!initialSharesGenerated) {
    initialSharesGenerated = true;
    initialShares = mpcShamirSharing.generateShares(inputData);
  }
}
```

```
protected long[] getInitialSharesForPrivacyPeer(int privacyPeerIndex) {
  return initialShares[privacyPeerIndex];
}
```

In the `TutorialProtocolPeer` class we then implement the method `createInitialSharesMessage`, which uses `generateInitialShares` and `getInitialSharesForPrivacyPeer` to create the message with initial shares for the privacy peer(s):

```
private synchronized void createInitialSharesMessage() {
  logger.log(Level.INFO, "Creating message for first round (send initial shares)...");
  inputPeer.generateInitialShares();
  messageToSend = new TutorialMessage(inputPeer.getMyPeerID(), myPeerIndex);
  messageToSend.setSenderIndex(myPeerIndex);
  messageToSend.setTimeSlotCount(timeSlotCount);
  messageToSend.setMetricCount(metricCount);
  messageToSend.setIsInitialSharesMessage(true);
  messageToSend.setShares(inputPeer.getInitialSharesForPrivacyPeer(privacyPeerIndex));
}
```

The `createInitialSharesMessage` method we just implemented is called from the `run` method of the `TutorialProtocolPeer` class to create the shares before sending them away.

Step 6: Receive Shares The privacy peers now obviously need the capabilities to receive these shares. The `run` method of the `TutorialProtocolPrivacyPeerToPeer` class calls the `receive-Message()` method which again forwards the received message to its observer, i.e., the privacy peer class. Therefore we have to add code to the `notificationReceived`[4] method of the `Tutorial-PrivacyPeer` class to actually handle the message with the initial shares (comments and logging omitted):

```
if (object instanceof TutorialMessage) {
  TutorialMessage msg = (TutorialMessage) object;
  if (msg.isDummyMessage()) {
    msg.setIsInitialSharesMessage(true);
  }

  if (msg.isInitialSharesMessage()) {
```

[4]The `update` method of Java's Observer design pattern is defined in the `TutorialBase` class and calls this method whenever it receives a `TutorialMessage`.

```
    TutorialPeerInfo peerInfo = getPeerInfoByPeerID(msg.getSenderID());
    peerInfo.setInitialShares(msg.getInitialShares());

    initialSharesToReceive--;
    if (initialSharesToReceive <= 0) {
      startNextPPProtocolStep();
    }

  } else {
    // create exception; see source code for details
  }
}
```

4.3 Computing Functions on Secrets

Step 7: Compute Functions At this point we get to the most interesting part: computing a function on the shared secrets. The code for the computation goes into the `TutorialPrivacyPeer` class. Here is the code for computing the product of all the input peers' data items:

```
1  public void startProductComputations() {
2      int activeInputPeers = connectionManager.getNumberOfConnectedPeers(false, true);
3      initializeNewOperationSet(numberOfItems);
4      operationIDs = new int[numberOfItems];
5      long[] data = null;
6      for(int operationIndex = 0; operationIndex < numberOfItems; operationIndex++) {
7        operationIDs[operationIndex] = operationIndex;
8        data = new long[activeInputPeers];
9        int dataIndex=0;
10       for(int peerIndex = 0; peerIndex < numberOfInputPeers; peerIndex++) {
11         long[] initialShares = getPeerInfoByIndex(peerIndex).getInitialShares();
12         if(initialShares!=null) { // only consider active input peers
13           data[dataIndex++] = initialShares[operationIndex];
14         }
15       }
16       primitives.product(operationIndex, data);
17     }
18 }
```

This method is executed once per round. When using the protocol primitives with the helper classes, we first have to initialize a new operation set with the number of operations we want to run in parallel (line 3). After that, we create a new integer array with the operation IDs. The IDs start at 0 and are consecutive. These IDs are later used for fetching the results of each operation. On line 8, the `data` array is created for holding each input peer's share. The actual `product` operations are scheduled within the for-loop by invoking the respective method on the `primitives` with the operation's ID and input data (line 16).

The above method is called from the `run` method of the `TutorialProtocolPrivacyPeerToPP` class (shown only in parts):

```
  if (ppThreadsBarrier.await()==0) {
    // compute and reconstruct products
    privacyPeer.startProductComputations();
  }

  ppThreadsBarrier.await();
  if(!doOperations()) {
    logger.severe("Computing products failed; returning...");
    return;
  }
```

The `ppThreadsBarrier` is provided by the privacy peer to synchronize all threads. One thread always prepares the computations (calls `startProductComputations`) and all of them then perform the computation (`doOperations`).

After the computation, shares of the product can be retrieved using the `primitives.getResult` method. As the last step of the overall computation, explicit reconstruction operations have to be scheduled. For this, we add the following code to the privacy peer class:

```java
  public void startFinalResultReconstruction() {
    long[] result = new long[operationIDs.length];
    for(int i = 0; i < operationIDs.length; i++) {
      result[i] = primitives.getResult(operationIDs[i])[0];
    }
    initializeNewOperationSet(result.length);
    operationIDs = new int[result.length];
    long[] data = null;
    for(int i = 0; i < result.length; i++) {
      operationIDs[i] = i;
      data = new long[1];
      data[0] = result[i];
      primitives.reconstruct(operationIDs[i], data);
    }
  }
```

The code is very similar to the code of the `startProductComputations` method. The main differences are that we use the `getResult` method first to get the result from the completed operations using their IDs and that we create reconstruct operations instead of multiplication operations. Note that retrieving the results must always be done before initializing a new operation set. Otherwise the results are lost!

This method is then called from the `run` method of `TutorialProtocolPrivacyPeerToPP`:

```java
  if (ppThreadsBarrier.await()==0) {
    privacyPeer.startFinalResultReconstruction();
  }

  ppThreadsBarrier.await();
  if(!doOperations()) {
    logger.severe("Final result reconstruction failed; returning...");
    return;
  }
```

After the results are reconstructed, we store them in an array (on the `TutorialPrivacyPeer`) and let the protocol threads, which communicate with the input peers, broadcast the results:

```java
1  public void setFinalResult() {
2    finalResults = new long[operationIDs.length];
3    for(int i = 0; i < operationIDs.length; i++) {
4      finalResults[i] = primitives.getResult(operationIDs[i])[0];
5    }
6    startNextPeerProtocolStep();
7  }
```

The call of `startNextPeerProtocolStep()` in line 6 opens the barrier at which all the `TutorialProtocolPrivacyPeerToPeer` threads are waiting (see the respective `run` method).

Again, this method has to be called by the `run` method of `TutorialProtocolPrivacyPeerToPP`:

```java
if (ppThreadsBarrier.await()==0) {
  privacyPeer.setFinalResult();
  logger.log(Level.INFO, "Tutorial protocol round completed");
}
```

4.4 Send, Receive and Write the Final Result to a File

Step 8: Send Final Results Next, we complete the `sendFinalResults` method in the `TutorialProtocolPrivacyPeerToPeer` class by setting the final results in the message:

```java
  // ...
  messageToSend.setResults(privacyPeer.getFinalResult());
  // ...
```

The `getFinalResult` method of the privacy peer class is simply a getter method:

```
public long[] getFinalResult() {
  return finalResults;
}
```

The `sendFinalResult` method is called from the `run` method (class `TutorialProtocolPrivacyPeer-ToPeer`) as soon as all results are ready.

After the final results were sent to all input peers, the final result is reported to the observers and the method checks if a new round should be started or if all protocols should be stopped.

Step 9: Receive Final Results Of course, the input peers need some code to handle the reception of the final results. After receiving a protocol message, the protocol thread notifies the observers, which in this case is the `TutorialPeer` class. Therefore, message handling is implemented in the `notificationReceived` method (comments and logging ommitted):

```
 1  if (object instanceof TutorialMessage) {
 2    TutorialMessage tutorialMessage = (TutorialMessage) object;
 3    if(tutorialMessage.isDummyMessage()) {
 4      tutorialMessage.setIsFinalResultMessage(true);
 5    }
 6
 7    if(tutorialMessage.isFinalResultMessage()) {
 8      finalResultsToDo--;
 9      if (finalResults == null && tutorialMessage.getResults() != null) {
10        finalResults = tutorialMessage.getResults();
11      }
12      if(finalResultsToDo <= 0) {
13        VectorData dummy = new VectorData();
14        FinalResultEvent finalResultEvent = new FinalResultEvent(this, myAlphaIndex, getMyPeerID(),
              tutorialMessage.getSenderID(), dummy);
15        finalResultEvent.setVerificationSuccessful(true);
16        sendNotification(finalResultEvent);
17
18        // See next step
19        writeOutputToFile();
20
21        if(currentTimeSlot < timeSlotCount) {
22          currentTimeSlot++;
23          initializeNewRound();
24        } else {
25          protocolStopper.setIsStopped(true);
26        }
27      }
28    } else {
29      // send exception event
30    }
31  }
```

From what you know by now, the code should be self-explanatory.

Step 10: Write Results to File The last thing left to do is writing the results to a file. To that end, we create the `writeResultToFile` method in `TutorialPeer` and call it from within the `notificationReceived` method (line 19 in the above code):

```
  protected void writeOutputToFile() throws Exception {
    String fileName = outputFolder + "/" + "tutorial_output_"
        + String.valueOf(getMyPeerID()).replace(":", "_") + "_round"
        + currentTimeSlot + ".csv";

    StringBuilder line = new StringBuilder();
    for (int resultIndex = 0; resultIndex < finalResults.length; resultIndex++) {
      line = line.append(finalResults[resultIndex]).append(", ");
    }
    Services.writeFile(line.substring(0, line.length() - 2), fileName);
  }
```

This code just writes the products separated by commas in one line to the file. The file name includes the input peer's ID and the current round number.

A final note: If your data is vector-oriented, as in the example above, you might want to use the class `mpc.VectorData` as a container for both input and output data. It also provides methods for reading and writing to/from files. In combination with the class `services.DirectoryPoller`, it allows to easily implement flexible file reading functionality, see for example the method `AdditivePeer.readNextInputFile()`.

References

[1] Martin Burkhart and Xenofontas Dimitropoulos. Privacy-preserving distributed network troubleshooting - bridging the gap between theory and practice. *ACM Trans. Inf. Syst. Secur.*, 14(4):31:1–31:30, December 2011.

[2] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *19th USENIX Security Symposium*, August 2010.

[3] Dilip Many, Martin Burkhart, and Xenofontas Dimitropoulos. Fast Private Set Operations with SEPIA. Technical report, ETH Zurich, April 2012.