DRAFT

User Manual for the Verificatum Mix-Net

Douglas Wikström
KTH Stockholm, Sweden
dog@csc.kth.se

January 19, 2012

Abstract

Verificatum, http://www.verificatum.org, is a free and open source implementation of an El Gamal-based mix-net. This document describes how to use the mix-net.

Help Us Improve This Document

This document is a draft. The most recent version of this document can be found at: http://www.verificatum.org/verificatum/vmnu.pdf.

Please help us improve the quality of this document by reporting errors, omissions, and suggestions to dog@csc.kth.se.

Contents

1 Introduction

The Verificatum mix-net [3] is an implementation of an El Gamal-based re-encryption mix-net. It can be configured in many ways, but all values have sensible defaults.

All that is needed to execute the mix-net is to run a few commands in the right sequence. Thus, we hope that the protocol is easy to use even for people who have no background in cryptography. The following commands are provided.

- `vmni` – Info file generator used to generate configuration files for the mix-net. Some optional configuration parameters are outputs from the object generator `vog` described below.
- `vmn` – Mix-server executing the Verificatum mix-net. The execution is parametrized by configuration files output from `vmni`.
- `vog` – Object generator of primitive cryptographic objects such as hash functions, keys, and pseudo-random generators. These can then, using `vmni`, be used to replace the default values.

Throughout the document we mark advanced sections by an asterix. These are sections that target programmers or users that must use special configurations.

Are you simply looking for an example? Appendix A contains a complete description of the commands executed by the respective operators in an execution with three mix-servers, including how to generate demo ciphertexts using the `vmnd` command.

2 Info File Generator

Before the mix-net is executed, the operators of the mix-servers must agree on a set of common parameters, generate their private and public parameters, and share their public parameters.

2.1 Basic Usage

The info file generator is used in three simple steps. Below we walk through an example with three mix-servers, where we describe the view of the operator of the first mix-server. The other operators execute corresponding commands.

1. **Agree on Common Parameters.** The operators agree on the name and session identifier of the execution, and the maximal number of ciphertexts to be processed, and then generate a stub of a protocol file. To do that each operator invokes `vmni` as follows.

   ```
   vmni -prot -sid "Session ID" -name "Swedish Election" \
        -nopart 3 -thres 2 -maxciph 10000 stub.xml
   ```

   The command produces a protocol info stub file `stub.xml`. The parties can then verify that they hold identical protocol info stub files by computing digests as described below.

   The session identifier can be used to separate multiple executions that logically should have the same name. In the example, the number of mix-servers is 3 of which 2 are

needed to decrypt ciphertexts encrypted with the joint public key produced during the key generation phase. The integer 10000 gives a bound on the number of ciphertexts that can be input to the mix-net and indicates that pre-computation is used. If this option is dropped, then no pre-computation takes place, and calling `vmn -precomp`, as explained in Section 3, Step 2, will fail and give a warning.

2. **Generate Individual Info Files.** Using the protocol info stub file `stub.xml` as a starting point, each operator generates its own private info file and protocol info file by invoking `vmni` as in the following example.

```
vmni -party -name "Mix-server 1" stub.xml \
     privInfo.xml localProtInfo.xml
```

   The command produces two files: a private info file `privInfo.xml` and an *updated* protocol info file `localProtInfo.xml`. The former contains private data, e.g., the signature key and the local directory storing the state of the mix-server. The latter defines the public parameters of a party, e.g., its IP address and public signature key. Each party shares its local protocol info file with the other mix-servers using an out-of-bound channel. The protocol info stub file `stub.xml` can now be deleted.

3. **Merge Protocol Info Files.** At this point the operator holds three protocol info files: its local file `localProtInfo.xml`, and the local files `protInfo2.xml`, and `protInfo3.xml` of the other parties. It merges these files using the following command.

```
vmni -merge localProtInfo.xml protInfo2.xml protInfo3.xml \
     protInfo.xml
```

   This produces a single *joint* protocol info file `protInfo.xml` containing all the public information about common parameters and all parties. The output file does not depend on the order of the input protocol info files (see Section 2.3 for more information).

In practice, the operators could, e.g., organize a physical meeting to which they bring their laptops and execute the above steps.

Computing Digests of Info Files. For convenience, hexadecimal encoded hash digests of files can be computed using `vmni` to allow all parties to check that they hold identical protocol info files at the end. In our example, a SHA-256 digest of the joint protocol info file can be computed as follows.

```
vmni -digest protInfo.xml
```

2.2 Execution in a Directory

In a typical application, each operator creates a directory and executes the above commands in this directory. For convenience, the `vmni` command allows the operator to drop many info file names when executing the commands in this way, in which case the file names default to the

file names used above. A similar convention is later used for the `vmn` command described in Section 3. More precisely, the commands below are equivalent to the above.

1. **Agree on Common Parameters.** The following creates a stub info file `stub.xml`.

```
vmni -prot -sid "Session ID" -name "Swedish Election" \
     -nopart 3 -thres 2 -maxciph 10000
```

2. **Generate Individual Info Files.** The following assumes that there is a stub file named `stub.xml` in the working directory and then creates a private info file `privInfo.xml` and a protocol info file `localProtInfo.xml`.

```
vmni -party -name "Mix-server 1"
```

3. **Merge Protocol Info Files.** The following creates a protocol info file `protInfo.xml`.

```
vmni -merge localProtInfo.xml protInfo2.xml protInfo3.xml
```

2.3 Additional Configuration Options*

The command `vmni` used to generate info files accepts a large number of options which allows defining various parameters of the mix-net. In our example we have simply used the default values, but we discuss a few of the options below. For a complete usage description use the following command, or generate info files as above and inspect the resulting files (a comment is generated for each value).

```
vmni -h
```

Running Multiple Mix-servers on a Single Computer. Each mix-server allocates two ports for communication: one for its HTTP server and one for its "hint server". By default these port numbers are 8040 and 4040, and this typically works well when running a single mix-server. However, if there is a need to run several mix-servers on the same computer, e.g., when trying out Verificatum, then the mix-servers must be assigned distinct port numbers. The `-http` and `-httpl` options are used to define the external and local URL's for the HTTP server. These can be distinct e.g., if port forwarding is used. If only `-http` is given, then the local port number defaults to the same value. If only `-httpl` is given then the external port number remains 8040. Similarly, `-hint`, and `-hintl` options are used to define the socket address of the hint server. Below we give an example, but Appendix A also illustrates the use of these options.

```
-http http://server.example.com:8040
-hint server.example.com:4040
```

Secure Source of Randomness. The default source of randomness is the `/dev/urandom` device. This is often a reasonable choice, but on machines with few system events, this may not give sufficient entropy. The `-rand` option can be used to either use a different device, e.g., a hardware random generator mounted as a device, or a pseudo-random generator. In the latter case, the `-seed` option must also be used to provide the name of a file containing a relatively short truly random seed. Use the `vog` tool described in Section 5 to generate a hexadecimal-encoded instance of a random source that can be used as a value with `-rand`. Below we give two examples.

```
-rand "$(vog -gen RandomDevice /dev/urandom)"
-rand "$(vog -gen PRGElGamal -fixed 2048)" -seed /dev/urandom
```

Mix-Net Interface. From an abstract point of view, the mix-net first outputs a file containing a joint public key package with an embedded description of the underlying group. Then it takes a file containing ciphertexts as input and produces a file containing plaintexts as output. The format used for these files depend on the application and can be changed using the `-inter` option. The value can be one of the strings `native`, `raw`, or `helios`, or the name of a subclass of `verificatum.protocol.mixnet.MixNetElGamalInterface`. Section 4 describes these formats and how to implement such a subclass.

Roles of the Mix-servers. The roles of the mix-servers are not entirely symmetric. If there are k mix-servers of which λ are needed to decrypt, then only the first λ of the mix-servers take part in the shuffling, but all take part in the decryption. The order in which the public information of the mix-servers appear in the protocol info file determines the roles of the mix-servers. When the local protocol info files are merged the sections belonging to the different parties are sorted before the result is output as a joint protocol info file. The parties are sorted based on the `-srtbyrole` parameter and their names. If two parties have the same name, then merging fails, so this is gives an unambiguous ordering of the parties.

3 Mix-Net

When the info files of all mix-servers have been generated the mix-net can be executed in two (or three) simple steps.

WARNING! It is the responsibility of the user to make sure that the input ciphertexts are distinct and unrelated to any ciphertexts submitted by honest senders. Failure to do so allows a fully practical attack on the privacy of any sender.

3.1 Basic Usage

We complete the example from Section 2 by describing the sequence of commands executed by the operator to actually run the mix-net.

1. **Generate Public Key.** The operators execute the joint key generation phase of the protocol which outputs a joint public key to a file `publicKey` to be used by senders when computing their ciphertexts.

```
vmn -keygen privInfo.xml protInfo.xml publicKey
```

   We stress that this command invokes a protocol, so all operators must execute this command roughly at the same time.[1]

   The format used to output the public key to the file `publicKey` is defined by the *mix-net interface*. See Section 2.3 for how to choose which interface to use, and see Section 4 for details on the available interfaces and how implement your own.

2. **Pre-compute (optional).** If the `-maxciph` option was used when generating the protocol info stub file, then pre-computation can (optionally) be executed as a separate step to speed up the mixing phase that follows. To start the pre-computation phase, the operator executes the following command.

```
vmn -precomp privInfo.xml protInfo.xml
```

   We stress that the the mix-servers execute a protocol during the pre-computation phase. Thus, all of the operators must execute this command roughly at the same time.

3. **Mix Ciphertexts.** To start the mixing phase with a file `ciphertexts` containing ciphertexts, the operator uses the following command.

```
vmn -mix privInfo.xml protInfo.xml ciphertexts plaintexts
```

   We stress that this command invokes a protocol, so all operators must execute this command roughly at the same time.

   The formats of the files `ciphertexts` and `plaintexts` are defined by the mix-net interface. See Section 2.3 for how to choose which interface to use, and see Section 4 for details on the available interfaces and how implement your own.

3.2 Resetting Securely

If there is a fatal error during the execution, or if an operator interrupts an execution by mistake, then all operators can execute the following command to securely reset the states of all the mix-servers to their states right after the end of the key generation step. Then the options `-precomp` and `-mix` can be used to execute the mix-net again.

```
vmn -reset privInfo.xml protInfo.xml
```

[1] Currently the implementation of the bulletin board is not robust, so there is essentially no time-out.

If pre-computation was carried out for too few ciphertexts, then the operators must manually edit their protocol info files and set the maximal number of ciphertexts to zero. Then they can use the `-mix` option to execute the mix-net without pre-computation.

Please note that resetting deletes all values computed during pre-computation. This is required to be able to securely reset from an arbitrary point in the protocol.

3.3 Execution in a Directory

If the `vmni` command was used within a directory as explained in Section 2.2, then the info file names can be dropped in the calls to `vmn` above. More precisely, the following calls can be used instead. Each command assumes that there is a private info file named `privInfo.xml` and a protocol info file `protInfo.xml` in the working directory.

1. **Generate Public Key.**

```
vmn -keygen publicKey
```

2. **Pre-compute (optional).**

```
vmn -precomp
```

3. **Mix Ciphertexts.**

```
vmn -mix ciphertexts plaintexts
```

Securely Resetting. The info files can of course be dropped also when executing the reset command.

```
vmn -reset
```

4 Interfaces to the Mix-Net

The interface of a mix-net is the format used to represent: the joint public key, the input ciphertexts, and the output plaintexts. The representation of the joint public key also embeds a representation of the underlying group. The Verificatum mix-net is configured to use a particular interface using the `-inter` option to `vmni`. As explained in Section 2.3, parameter to this option can be either the short name of a built-in interface (`native`, `raw`, or `helios`), or the name of a class implementing the interface. Below we describe the built-in interfaces in terms of byte trees and the representations of group elements defined in Appendix B and Appendix C (and in a different context in [2]), and then describe how to implement a custom interface. Here G_q denotes the underlying group, $\mathcal{M}$ denotes the group of plaintexts (some fixed power of G_q), and $\mathcal{C} = \mathcal{M} \times \mathcal{M}$ denotes the group of ciphertexts.

4.1 Native Interface

The native interface converts the binary objects of the raw interface to their hexadecimal representation and does not require the number of ciphertexts in the input file to be available. It also decodes the output plaintext group elements into strings according to the encoding scheme of the underlying group. More precisely, $\mathsf{hex}(\mathsf{node}(\mathsf{marshal}(G_q), \overline{pk}))$ is output on the public key file, where pk denotes the public key in $G_q \times G_q$. For each line in the file of input ciphertexts an attempt is made to interpret it as $\mathsf{hex}(\overline{w})$ for some ciphertext $w \in \mathcal{C}$. Lines for which this fails are ignored. The array m of plaintext group elements in $\mathcal{M}$ are decoded element-wise into strings using the decoding scheme of the underlying group. Any occurrence of a newline or carriage return character in a string is deleted before the strings are output on file separated by newline characters.

4.2 Raw Interface

The raw interface uses the internal representation of the public key (with an embedded representation of the underlying group), the input ciphertexts, and output plaintext group elements. More precisely, at the end of the key generation phase, the public key file contains $\mathsf{node}(\mathsf{marshal}(G_q), \overline{pk})$, where pk is the public key in the product group $G_q \times G_q$. The array L_0 of input ciphertexts in $\mathcal{C}$ is represented on file as $\overline{L_0}$ and the array m of output plaintext group elements in $\mathcal{M}$ is represented on file as $\overline{m}$. We stress that the output group elements are not decoded into strings.

4.3 Helios Interface

The Helios interface assumes that a modular group is used and that $\mathcal{M} = G_q$, i.e., that the underlying group must be an instance of `verificatum.arithm.ModPGroup` and that each plaintext is a single group elements. A modular group is used by `vmni` by default. The format of the public key file is best explained by an example. Suppose that $p = 23$, $q = 11$, and that G_q is the subgroup of order q in $\mathbb{Z}_p^*$. Furthermore, let $g = 3$ and let $y = 13$. Then the public key is represented as

```
{"g":"3","p":"23","q":"11","y":"13"} .
```

Similarly, a ciphertext $(12, 16) \in G_q \times G_q$ is represented as

```
{"alpha":"12","beta":"16"}
```

and the input file of ciphertexts contains a single such line for each ciphertext without any additional delimiters. The output file of plaintexts is constructed exactly like in the native interface.

4.4 Custom Interface*

The interface of the mix-net is captured by a subclass of `MixNetElGamalInterface` (found in the `verificatum.protocol.mixnet` package). This class requires every subclass to implement five methods.

- `writePublicKey` – Writes a public key to file.

- `readPublicKey` – Reads a public key from file, including the underlying group.
- `writeCiphertexts` – Writes ciphertexts to file.
- `readCiphertexts` – Reads ciphertexts from file.
- `decodePlaintexts` – Decodes plaintext group elements and writes the result to file.

Please consider the source of, e.g., `MixNetElGamalInterfaceNative`, for an example.

5 Object Generator

Some of the option parameters passed to `vmni` can be complex objects, i.e., a provably secure pseudo-random generator may be based on a computational assumption that must be part of the encoding. The object generator `vog` is used to generate representations of such objects. Before any objects are generated, the source of randomness of the object generator must be initialized, but we postpone the discussion of this to Section 5.2.

5.1 Listing and Generating Objects

The main usage of `vog` is to list all suitable subclasses of some class specified as a valid parameter to `vmni`, and then to generate an instance of such a class.

Browse Library. Consider an option to `vmni` that is parametrized by instance of a subclass of a class `AbstractClass`. Then the set of subclasses of `AbstractClass` that can be instantiated using `vog` can be listed using the following command.

```
vog -list AbstractClass
```

For example, the source of randomness used by a protocol can be chosen by passing an instance of a subclass of `verificatum.crypto.RandomSource` to `vmni` using the `-rand` option. (Use `vmni -h` to find out which type of object can be passed as a parameter with each option.) To list all suitable subclasses, the following command can be used.

```
vog -list RandomSource
```

As illustrated by the example it suffices to give the unqualified class name when this is not ambiguous.

Generate Object. To generate an instance of `ConcreteClass` the `-gen` option is used along with the name of the class, but each class requires its own set of parameters. To determine the correct set of parameters the following command can be used.

```
vog -gen ConcreteClass -h
```

This prints usage information as if `vog -gen ConcreteClass` was a command on its own. An instance is then generated by passing the correct parameters, e.g., to generate an instance of `HashfunctionHeuristic` that represents SHA-512, the following command can be used.

```
vog -gen HashfunctionHeuristic SHA-512
```

For some classes, the parameters passed to `vog` must in turn be generated using `vog` itself, e.g., `PRGHeuristic` optionally takes the representation of a hash function as input as illustrated in the following.

```
vog -gen PRGHeuristic \
    "$(vog -gen HashfunctionHeuristic SHA-256)"
```

This approach of constructing parametrized complex objects is quite powerful. We can for example construct an instance of `PRGCombiner` that combines a random device and two pseudo-random generators using the following command.

```
vog -gen PRGCombiner                                \
    "$(vog -gen RandomDevice /dev/urandom)"         \
    "$(vog -gen PRGElGamal -fixed 1024)"            \
    "$(vog -gen PRGHeuristic                        \
       "$(vog -gen HashfunctionHeuristic SHA-256)" \
    )"
```

5.2 Initializing the Random Source

Before `vog` is used to generate any objects, its source of randomness must be initialized. This is only done once. The syntax is almost identical to the syntax to generate an instance of a subclass of `RandomSource`, except that it is mandatory to provide a seed if a pseudo-random generator is used. We give two examples. The first example initializes the random source to be the random device `/dev/urandom`. Any device can of course be used, e.g., a hardware random generator mounted as a device. To avoid accidental reuse of randomness, this option should *never* be used with a normal file.

```
vmn -rndinit RandomDevice /dev/urandom
```

The second example shows how to initialize the random source as a pseudo-random generator where the seed is read directly from `/dev/urandom`.

```
vog -seed /dev/urandom -rndinit \
    PRGHeuristic "$(vog -gen HashfunctionHeuristic SHA-256)"
```

A representation of the random source is stored in the file `.verificatum_random_source` in the home directory of the user. If the environment variable `VERIFICATUM_RANDOM_SOURCE` is defined, then it is taken to be the name of a file to be used instead. If the random source

is a pseudo-random generator, i.e., a subclass of `verificatum.crypto.PRG`, then the hexadecimal encoding of its seed is stored in the file `.verificatum_random_seed`, or if `VERIFICATUM_RANDOM_SEED` is defined it is interpreted as a file name to be used instead. Note that the seed is automatically replaced with part of the output of the pseudo-random generator in each invocation to avoid accidental reuse of the seed.

5.3 Custom Objects*

To allow `vog` to instantiate a custom subclass `CustomClass`, e.g., of `PGroup`, a separate subclass `CustomClassGen` of `verificatum.ui.gen.Generator` must also be implemented. Please note the naming convention where `Gen` is added as a suffix to the class name. The generator class provides the command-line interface to `CustomClass`, i.e., it prints usage information, and interprets command-line parameters and returns an instance of `CustomClass`. See source for `HashfunctionHeuristicGen` for a simple example.

There are two ways to make `vog` aware of such custom classes: (1) a colon-separated list of classes can be provided as a parameter with the `-pkgs` option, or (2) the environment variable `VERIFICATUM_VOG` can be initialized to such a list. Each class identifies a package to be considered by `vog`, so it suffices to provide a single class from each package to be considered.

6 Universally Verifiable Proof of Correctness

By default, `vmni` generates a protocol info stub file such that all zero-knowledge proofs computed during the execution of `vmn` are made *non-interactive* using the Fiat-Shamir heuristic. When `vmn` is executed in this mode it stores all the relevant intermediate results along with the non-interactive zero-knowledge proofs in the `roProof` subdirectory of the working directory of `vmn`. The contents of this directory can then be verified by anybody that has the necessary knowledge to implement a verification algorithm. Thus, the Verificatum mix-net is said to be *universally verifiable*.

6.1 The Verificatum Mix-Net Verifier

The Verificatum built-in verifier command `vmnv` can handle all the built-in mix-net interfaces. Let `protInfo.xml` be a protocol info file and let `roProof` be a directory containing the intermediate values and non-interactive zero-knowledge. Then the consistency of these can be verified using the following command.

```
vmnv protInfo.xml roProof
```

A given mix-net interface is of course used during the execution of `vmn` and the public key file `publicKey` handed to senders, the file `ciphertexts` containing input ciphertexts, and the output plaintext file `plaintexts` are represented using this interface. To verify that these files correspond correctly to the public key, input ciphertexts, and the output plaintexts stored in the raw format within the `roProof` the following command can be used (which also verifies the contents of `roProof`).

```
vmnv protInfo.xml roProof publicKey ciphertexts plaintexts
```

6.2 Independent Stand-alone Mix-Net Verifier*

Universal verifiability is of course more interesting if independent parties implement stand-alone verifiers. These verifiers should preferably share no code with Verificatum itself.

In a companion document [2] targeting programmers of such verifiers, the formats of the files in the `roProof` directory and the algorithms that must be implemented are described in detail. There is also a simple reference implementation written in Python[1] that follows the notation in [2] closely.

7 Demonstrator

In the `demo/mixnet` subdirectory of the installation directory of the Verificatum package contains a number of demo scripts. The following simple command runs the demo with the default options.

```
./demo
```

The demo can be configured to illustrate almost every option of the mix-net. It is also easy to configure it to orchestrate an execution on multiple computers remotely. For more information we refer the reader to the `README` file and the configuration file `conf` found in the `demo/mixnet` subdirectory.

Concrete Example. For a more concrete example, Appendix A contains a worked example of the commands executed by the operators including how to generate demo ciphertexts using the `vmnd` command.

8 Troubleshooting

- If you get a "Connection refused." error, then it could be that the HTTP-server resolves your hostname, e.g., server.example.org to `127.0.0.1`, i.e., localhost. This is not a problem in Verificatum. The problem is that your server resolves your hostname incorrectly. Often it is possible to solve this problem by editing `/etc/hosts`. Google to figure out how this is done on your OS.
- If you get an "Invalid socket address!" error, then it could be that the port number you are trying to use is already allocated by an other mix-server (or other server). See Section 2.3 for how to run multiple mix-servers on a single computer.

9 Acknowledgments

The suggestions of Shahram Khazaei and Gunnar Kreitz have improved this document.

References

[1] Python Verificatum mix-net verifier. http://www.verificatum.org/pvmnv, November 2011.

[2] D. Wikström. How to implement a stand-alone verifier for the Verificatum mix-net. Manuscript, 2011. Available at http://www.verificatum.org.

[3] D. Wikström. The Verificatum mix-net. Manuscript, 2011. In preparation.

A Commands for an Execution with Three Mix-servers

```
###################### Set up directories ############################

mkdir -p mydemo/1
mkdir -p mydemo/2
mkdir -p mydemo/3

#################### Executed by Operator 1 ##########################

#### Step 0 #### Generate stub file.

cd mydemo/1
vmni -prot -sid "Session ID" -name "Swedish Election" \
     -nopart 3 -thres 2

#### Step 1 #### Generate private info and protocol info files.

vmni -party -name "Mix-server 1" \
-http http://localhost:8041 \
-hint localhost:4041

cp localProtInfo.xml protInfo1.xml
cp protInfo1.xml ../2/
cp protInfo1.xml ../3/

#### Step 2 #### Merge protocol files and generate public key.

vmni -merge protInfo?.xml
vmn -keygen publicKey

#### Step 3 #### Generate demo ciphertexts (in reality by senders).

vmnd publicKey 100 ciphertexts
cp ciphertexts ../2/
cp ciphertexts ../3/

#### Step 4 #### Mix the ciphertexts.

vmn -mix ciphertexts plaintexts


#################### Executed by Operator 2 ##########################
```

```
#### Step 0 #### Generate stub file.

cd mydemo/2
vmni -prot -sid "Session ID" -name "Swedish Election" \
     -nopart 3 -thres 2

#### Step 1 #### Generate private info and protocol info files.

vmni -party -name "Mix-server 2" \
-http http://localhost:8042 \
-hint localhost:4042

# Copy protocol info files using out-of-bound channel.
cp localProtInfo.xml protInfo2.xml
cp protInfo2.xml ../1/
cp protInfo2.xml ../3/

#### Step 2 #### Merge protocol files and generate public key.

vmni -merge protInfo?.xml
vmn -keygen publicKey

#### Step 3 #### Wait for demo ciphertexts (generated by Operator 1).

#### Step 4 #### Mix the ciphertexts.

vmn -mix ciphertexts plaintexts


#################### Executed by Operator 3 ##########################

#### Step 0 #### Generate stub file.

cd mydemo/3
vmni -prot -sid "Session ID" -name "Swedish Election" \
     -nopart 3 -thres 2

#### Step 1 #### Generate private info and protocol info files.

vmni -party -name "Mix-server 3" \
-http http://localhost:8043 \
-hint localhost:4043

# Copy protocol info files using out-of-bound channel.
cp localProtInfo.xml protInfo3.xml
cp protInfo3.xml ../1/
cp protInfo3.xml ../2/

#### Step 2 #### Merge protocol files and generate public key.

vmni -merge protInfo?.xml
vmn -keygen publicKey

#### Step 3 #### Wait for demo ciphertexts (generated by Operator 1).
```

```
#### Step 4 #### Mix the ciphertexts.

vmn -mix ciphertexts plaintexts


#################### Executed by Observer ############################

# Verify internal consistency of Fiat-Shamir proof.
vmnv protInfo.xml dir/roProof

# Verify internal consistency of Fiat-Shamir proof and correspondence
# with public key, input ciphertexts, and ouput plaintexts.
vmnv protInfo.xml dir/roProof publicKey ciphertexts plaintexts
```

B Byte Trees

We use a byte-oriented format to represent objects on file and to turn them into arrays of bytes that can be input to a hash function. The goal of this format is to be as simple as possible.

B.1 Definition

A byte tree is either a *leaf* containing an array of bytes, or a *node* containing other byte trees. We write $\mathsf{leaf}(d)$ for a leaf with a byte array d and we write $\mathsf{node}(b_1, \ldots, b_l)$ for a node with children $b_1, \ldots, b_l$. Complex byte trees are then easy to describe.

Example 1. The byte tree containing the data `AF`, `03E1`, and `2D52` (written in hexadecimal) in three leaves, where the first two leaves are siblings is

$$\mathsf{node}(\mathsf{node}(\mathsf{leaf}(\texttt{AF}), \mathsf{leaf}(\texttt{03E1})), \mathsf{leaf}(\texttt{2D52})) \ .$$

B.2 Representation as an Array of Bytes

We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the two's-complement representation of n in big endian byte order truncated to the k least significant bytes. We also use hexadecimal notation for constants, e.g., `0A` means $\mathsf{bytes}_1(10)$.

A byte tree is represented as an array of bytes as follows.

- A leaf $\mathsf{leaf}(d)$ is represented as the concatenation of: a single byte `01` to indicate that it is a leaf, four bytes $\mathsf{bytes}_4(l)$, where l is the number of bytes in d, and the data bytes d.
- A node $\mathsf{node}(b_1, \ldots, b_l)$ is represented as the concatenation of: a single byte `00` to indicate that it is a node, four bytes $\mathsf{bytes}_4(l)$ representing the number of children, and $\mathsf{bytes}(b_1) \mid \mathsf{bytes}(b_2) \mid \cdots \mid \mathsf{bytes}(b_l)$, where $\mid$ denotes concatenation and $\mathsf{bytes}(b_i)$ denotes the representation of the byte tree b_i as an array of bytes.

Example 2 (Example 1 contd.). The byte tree is represented as the following array of bytes.

```
00 00 00 00 02
  00 00 00 00 02
    01 00 00 00 01 AF
    01 00 00 00 02 03 E1
  01 00 00 00 02 2D 52
```

It is a good idea to hand (an upper bound of) the expected recursive depth to a parser to let it fail in a controlled way if the input does not represent a valid byte tree.

ASCII strings are converted to byte trees in the natural way, i.e., a string s (an array of bytes) is converted to $\mathsf{leaf}(s)$.

Example 3. The string `"ABCD"` is represented by $\mathsf{leaf}(\texttt{65666768})$.

Sometimes we store byte trees as the hexadecimal encoding of their representation as an array of bytes. We denote by $\mathsf{hex}(a)$ the hexadecimal encoding of an array of bytes. We abuse notation and simply write a instead of $\mathsf{bytes}(a)$ when the context is clear, e.g., if H is a hash function we write $H(a)$ instead of $H(\mathsf{bytes}(a))$.

B.3 Backus-Naur Grammar.

The above description should suffice to implement a parser for byte trees, but for completeness we give their Backus-Naur grammar. We denote the n-fold repetition of a symbol $\langle \textit{rule} \rangle$ by $n\langle \textit{rule} \rangle$. The grammar then consists of the following rules for $n = 0, \ldots, 2^{31} - 1$, where $|$ denotes choice.

$$
\begin{aligned}
\langle \textit{bytetree} \rangle &::= \langle \textit{leaf} \rangle \mid \langle \textit{node} \rangle \\
\langle \textit{leaf} \rangle &::= \texttt{01}\langle \textit{uint}_n \rangle \langle \textit{data}_n \rangle \\
\langle \textit{node} \rangle &::= \texttt{00}\langle \textit{uint}_n \rangle \langle \textit{bytetrees}_n \rangle \\
\langle \textit{uint}_n \rangle &::= \mathsf{bytes}_4(n) \\
\langle \textit{data}_n \rangle &::= n\langle \textit{byte} \rangle \\
\langle \textit{bytetrees}_n \rangle &::= n\langle \textit{bytetree} \rangle \\
\langle \textit{byte} \rangle &::= \texttt{00} \mid \texttt{01} \mid \texttt{02} \mid \cdots \mid \texttt{FF}
\end{aligned}
$$

C Representations of Groups and Group Elements

Group elements are represented as byte trees. In this section we pin down the details of these representations.

- **Modular Group.** A subgroup G_q of order q of the multiplicative group $\mathbb{Z}_p^*$, where $p > 3$ is prime, with standard generator g is represented by the byte tree $\mathsf{node}(\overline{p}, \overline{q}, \overline{g}, \mathsf{bytes}_4(e))$, where the integer $e \in \{0, 1, 2\}$ determines how a string is encoded into a group element. We denote by $\mathsf{ModPGroup}(b)$ the group recovered from such a byte tree b.

  We may only have $e = 1$ if $p = 2q + 1$, and we may only have $e = 2$ if $p = tq + 1$ and the difference between the bit lengths of p and q is less than 2^{10}.

  The three encoding schemes are defined as follows.

  - If $e = 0$, then at most 3 bytes can be encoded. Let H denote SHA-256.

    Encode. To encode a sequence m of $0 \leqslant b_m \leqslant 3$ bytes into an element a, find an element $a \in G_q$ such that the first $b_m + 1$ bytes of $H(\overline{a})$ are of the form $\mathsf{bytes}_1(l) \mid m$ for some l such that $l \bmod 4 = b_m$.

    Decode. To decode an element $a \in G_q$ into a message m, let $\mathsf{bytes}_1(l')$ be the first byte of $H(\overline{a})$ and define $l = l' \bmod 4$. Then let m be the next l bytes of $H(\overline{a})$.

– If $e = 1$, then at most $b = \lfloor (n-2)/8 \rfloor - 4$ bytes can be encoded, where n is the bit length of p.

Encode. To encode a sequence m of $0 \leqslant b_m \leqslant b$ bytes as an element $a \in G_q$ do as follows. If $b_m = 0$, then set $m' = \texttt{01} \mid \mathsf{bytes}_{b-1}(0)$ and otherwise set $m' = m \mid \mathsf{bytes}_{b-b_m}(0)$. Then interpret $\mathsf{bytes}_4(b_m) \mid m'$ as a positive integer k. Then let a be k or $p - k$ so that $a \in G_q$.

Decode. To decode an element $a \in G_q$ to a message m, set k equal to a or $p - a$ depending on if $a < p - a$ or not. Then interpret $k \bmod 2^{8(b+4)}$ as $\mathsf{bytes}_4(l) \mid m'$, where m' is a sequence of b bytes. If $l < 0$ or $l > b$, then set $l = 0$. Then let m be the l first bytes of m'.

– If $e = 2$, then at most $b = \lfloor n/8 \rfloor - n' - 4$ bytes can be encoded, where n is the bit length of p and $n' = \lceil t/8 \rceil + 1$.

Encode. To encode a sequence m of $0 \leqslant b_m \leqslant b$ bytes as an element $a \in G_q$, interpret $\mathsf{bytes}_4(b_m) \mid m \mid \mathsf{bytes}_{b-b_m}(0)$ as an integer k. Then let a be the smallest integer of the form $i2^{8(b+4)} + k$ in G_q for some non-negative i.

Decode. To decode an element $a \in G_q$ to a message m, interpret $a \bmod 2^{8(b+4)}$ as $\mathsf{bytes}_4(l) \mid m'$, where m' is a sequence of b bytes. If $l < 0$ or $l > b$, then set $l = 0$. Then let m be the l first bytes of m'

- **Element of Modular Group.** An element $a \in G_q$, where G_q is a subgroup of order q of $\mathbb{Z}_p^*$ for a prime p is represented by $\mathsf{leaf}(\mathsf{bytes}_k(a))$, where a is identified with its integer representative in $[0, p-1]$ and k is the smallest integer such that p can be represented as $\mathsf{bytes}_k(p)$.

  Example 4. Let G_q be the subgroup of order $q = 131$ in $\mathbb{Z}_{263}^*$. Then $258 \in G_q$ is represented by `01 00 00 00 02 01 02`.

- **Array of Group Elements.** An array $(a_1, \ldots, a_l)$ of group elements is represented as $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of a_i.

- **Product Group Element.** An element $a = (a_1, \ldots, a_l)$ in a product group is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of a_i. This is similar to the representation of product rings.

- **Array of Product Group Elements.** An array $(a_1, \ldots, a_l)$ of elements in a product group, where $a_i = (a_{i,1}, \ldots, a_{i,k})$, is represented by $\mathsf{node}(\overline{b_1}, \ldots, \overline{b_k})$, where b_i is the array $(a_{1,i}, \ldots, a_{l,i})$ and $\overline{b_i}$ is its representation as a byte tree. This is similar to the representation of arrays of elements in a product ring.

Marshalling Groups. When objects convert themselves to byte trees in Verificatum, they do not store the name of the Java class of which they are instances. Thus, to recover an object from such a representation, information about the class must be otherwise available. Thus, a group G_q is marshalled into a byte tree

$$\mathsf{node}(\mathsf{leaf}(\texttt{"PGroupClass"}), \overline{G_q}) \ ,$$

where G_q is an instance of a Java class `PGroupClass`, and this is denoted $\mathsf{marshal}(G_q)$.