Verificatum

User Manual for the Verificatum Mix-Net

VMN Version 3.1.0
2022-09-10

The Verificatum mix-net is an implementation of a provably secure El Gamal-based mix-net. This document describes how to use the mix-net. For information about the design and implementation, please visit `http://www.verificatum.org`.

Help us improve this document! The most recent version of this document can be found at `http://www.verificatum.com`. Report errors, omissions, and suggestions to `docs@verificatum.com`.

Contents

1 Introduction

The Verificatum mix-net (VMN) 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. The reader is expected to understand what an El Gamal-based mix-net is, but there is no need to understand the inner workings of the mix-net.

Throughout the document we mark advanced sections by an asterix. These are sections that target programmers or power users that must use special configurations. These sections can safely be ignored in a first reading, or if these features are not needed.

Components. All that is needed to execute the mix-net is to run a few commands in the right sequence. Thus, we hope that VMN is easy to use even for people who have limited knowledge of cryptography. The following are the basic commands.

Command	Description
`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 VMN. 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.
`vmnv`	Verifier of universally verifiable non-interactive zero-knowledge proofs correct execution based on the Fiat-Shamir heuristic.
`vmnc`	Tool used to convert public keys, ciphertexts, and plaintexts to/from an application dependent format from/to the raw format used internally in VMN.

Usage information. Usage information for each command can be printed by passing `-h` as an option to it. However, the usage information for the above commands are given as appendices Appendix H - Appendix L.

Worked example. Appendix F 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 described in Appendix E.

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 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 then generate a stub of a protocol file. To do that each operator invokes `vmni` as follows.

```
vmni -prot -sid "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2 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, which should be globally unique, 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.

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, such as private signature keys. The latter defines the public parameters of a party, e.g., its IP address and public signature key. Each party must share its local protocol info file with the other mix-servers using an *authenticated channel*. The protocol info stub file `stub.xml` can now be deleted.

3. **Merge protocol info files.** The operator now 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 common parameters and the public information about all parties. The output file does not depend on the order of the input protocol info files.

The generated configuration files are write-protected to avoid accidental modifications. In practice, the operators could organize a physical meeting to which they bring their laptops and execute the above steps (in a way that is secure against side-channel attacks to not leak their secret keys). Alternatively, a simple protocol could be implemented using any PKI to perform the above tasks.

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 hexadecimal encoded 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 name parameters 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 "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2
```

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 important options below. For a complete usage description use the following command, or generate info files as above and inspect the resulting files. A comprehensive comment is generated for each value.

```
vmni -h
```

Pre-computation. The `-maxciph` option can be passed to `vmni` to invoke a pre-computation phase as explained in Section 3.2. For example, to configure the mix-net to process 10000 ciphertexts with pre-computation the option below can be used. (This is only used as a default and can be overridden for individual mixing sessions.)

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

Changing the default key width. In some settings it is useful to have $\kappa > 1$ public keys, in which case we say that the *keywidth* is κ. Let the set of plaintexts be $\mathcal{M}_\kappa = G_q^\kappa$, the set of ciphertexts be $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$, and the space of randomness be $\mathcal{R}_\kappa = \mathbb{Z}_q^\kappa$. We can then define operations componentwise, i.e., if $a, b \in \mathcal{M}_\kappa$ and $e \in \mathcal{R}_\kappa$, then $ab = (a_1 b_1, \ldots, a_\kappa b_\kappa)$ and $a^e = (a_1^{e_1}, \ldots, a_\kappa^{e_\kappa})$. We say that an element $g \in \mathcal{M}_\kappa$ is a generator if the map $f : \mathcal{R}_\kappa \to \mathcal{M}_\kappa$ defined by $f(e) = g^e$ is a bijection.

With these conventions in place, we may generalize El Gamal in the natural way. A generator $g \in \mathcal{M}_\kappa$ is given. A secret key is $x \in \mathcal{R}_\kappa$ is randomly chosen and the public key is then defined as (g, y), where $y = g^x$. A plaintext $m \in \mathcal{M}_\kappa$ is encrypted as $(g^r, y^r m)$. Decryption of a ciphertext (u, v) is simply defined as $u^{-x}v$.

Changing the default width of ciphertexts. A long plaintext that can not be embedded into a single group element can be split up and embedded into a list of group elements. A slight generalization (of the already generalized version) of El Gamal can then be used to encrypt the resulting list of group elements. More precisely, we let the set of plaintexts be $\mathcal{M}_{\kappa,\omega} = \mathcal{M}_\kappa^\omega$, the set of ciphertexts be $\mathcal{C}_{\kappa,\omega} = \mathcal{M}_{\kappa,\omega} \times \mathcal{M}_{\kappa,\omega}$, and the space of randomness be $\mathcal{R}_{\kappa,\omega} = \mathcal{R}_\kappa^\omega$, and define encryption of a message $m \in \mathcal{M}_{\kappa,\omega}$ using a public key (g, y) and randomness $r \in \mathcal{R}_{\kappa,\omega}$ by $(u, v) = \big((g^{r_1}, \ldots, g^{r_w}), (y^{r_1}m_1, \ldots, y^{r_w}m_w)\big)$. By extending our notation we can even write this as $(u, v) = (g^r, y^r m)$. We say that the *width* of plaintexts and ciphertexts is w. The mix-net can process a list of ciphertexts of any width, assuming of course that all ciphertexts have the same width. The `-width` option can be passed to the `vmni` command to change the default width for individual sessions. For example, the following changes the key width to 2 and the default width to 3.

```
vmni -prot -sid "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2 -keywidth 2 -width 3 stub.xml
```

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 4 to generate a hexadecimal-encoded instance of a random source that can be used as a value with `-rand`. Below we give two examples.

```
vmni -party -name "Mix-Server 1" \
     -rand "$(vog -gen RandomDevice /dev/urandom)" \
     privInfo.xml localProtInfo.xml
```

```
vmni -party -name "Mix-Server 1" \
     -rand "$(vog -gen PRGElGamal -fixed 2048)" \
     -seed /dev/urandom \
     privInfo.xml localProtInfo.xml
```

Note that multiple sources can be combined into a single source using the combiner class named `com.verificatum.crypto.RandomSourceCombiner` such that the security of the combined sources is as strong as the strongest of the combined source. Combining a hardware generator, `/dev/urandom`, and PRG should satisfy the requirements of any auditor.

Using a different bulletin-board. The default bulletin board uses a stripped-down version of HTTP with mutual signatures, but it is easy to replace the default bulletin board by another bulletin board.

The simplest way to replace the bulletin board is to merely use any existing HTTP server and use the `-httptype external` flag to `vmni` to indicate that an external HTTP server is serving

the files published as part of the existing bulletin board protocol. The rest of the protocol would then be intact.

Alternatively a subclass of the abstract class `com.verificatum.com.BullBoardBasic` can be implemented. This class must contain the abstract methods: `start`, `stop`, `publish`, `unpublish`, and `waitFor`. These functions must behave as expected. They are used to start/stop the bulletin board and to publish or wait for something to be published by another party. Please consult the VMN source code for more details. Thus, it is easy to write a wrapper for an existing bulletin board implementation.

If a bulletin board is implemented in a class `FooBullBoard`, then there must also exist a subclass of `com.verificatum.com.BullBoardBasicGen` named `FooBullBoardGen` that specifies the configuration data needed by the bulletin board. If a wrapper is implemented, then the configuration data can of course consist of a single field that specifies an external configuration file of the underlying bulletin board.

Given a custom bulletin board `FooBullBoard`, the mix-net can be instructed to use it by the following command.

```
vmni -party -name "Mix-Server 1" -bullboard FooBullBoard \
     privInfo.xml localProtInfo.xml
```

Note that if you use this option when printing usage info, then information about the configuration of your own bulletin board will be printed as well. Use the following to try this.

```
vmni -h -bullboard FooBullBoard
```

Running multiple mix-servers on a single computer. When the default bulletin board is used for communication between the mix-servers, each mix-server allocates two ports for communication: one for its HTTP server and one for its *hint server*. (The hint servers are used to optimistically reduce the need for servers to poll each other.) 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 VMN, 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 behind a NAT. 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 F also illustrates the use of these options.

```
vmni -party -name "Mix-Server 1" \
     -http http://server.example.com:8040 \
     -hint server.example.com:4040 \
     privInfo.xml localProtInfo.xml
```

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: generate a public key, optionally perform pre-computation, and process a

list of ciphertexts to produce a list of plaintexts. The formats used to represent the public key, the ciphertexts, and the plaintexts are detailed in Appendix C. Before we continue we warn the user.

WARNING! On its own the mix-net provides no protection against Pfitzmann's attack (malleability attack). Only use this software under the supervison of somebody that understand this warning.

The reason why such protection is not provided is that it is inherently application dependent. The warning is meant to be a reminder and not an explanation. If you do not understand the warning or this remark, you need to ask a cryptographer for help before deploying the mix-net.

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

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
```

2. **Mix ciphertexts.** To start the mixing phase with a file `ciphertexts` containing ciphertexts to produce a file `plaintexts` containing plaintexts, the operator uses the following command.

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

If the info files were generated in a directory as explained in Section 2.2 and the above commands are executed in the same directory, then the names of the info files can be dropped here and below.

Deleting a session. 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 delete all information about the current mixing session.

```
vmn -delete privInfo.xml protInfo.xml
```

Note that this only deletes the mixing data and not any information about the distributed key generation.

Changing the set of active parties. For distributed bulletin boards, there is no practical way to determine if a party will deliver a message or if it simply delays messages, since the expected delays differ depending on the role of the mix-server, its hardware, its network connection, etc. Thus, the approach of VMN is to simply let each mix-server wait indefinitely for each message. If all mix-servers end up waiting for the same mix-server, then the operators may decide to interrupt

and delete the session and then deactivate the mix-server. If a mix-server is deactivated, then from the other mix-servers point of view it is as if it always publishes a fixed message whenever it is expected to publish a message. Thus, deactivated mix-servers emulate a particular type of corrupted mix-server and are handled as such in the subprotocols.

The following two commands can be used to display and define the set of active parties.

```
vmn -lact privInfo.xml protInfo.xml
```

```
vmn -sact privInfo.xml protInfo.xml "{1,3}"
```

The latter command sets the set of active servers to the first and third mix-servers. Both commands can be executed without explicitly giving the info files if previous commands were used in the same way.

3.2 Other Ways to Use the Mix-Net*

The mix-net can be used to do more than simply mixing. Pre-computation can be used to speed up the shuffling phase of the mix-net, ciphertexts can be shuffled without decrypting, and ciphertexts can be decrypted without shuffling. A public key can be imported to the mix-net and used to shuffle ciphertexts without decrypting them. In combination with the session handling described in Section 3.3 and the ability to override the keywidth as described in Section 2.3 and the width as described in Section 3.4 this turns the mix-net into a blackbox that provides all the functionality needed in virtually every electronic voting systems.

Pre-computation. Pre-computation can be used to speed up the mixing phase of the mix-net, but all mix-servers must agree on the number of ciphertexts for which pre-computation is performed. If the `-maxciph` option was used when generating the protocol info stub file, then there is a default number of ciphertexts for which pre-computation is performed and pre-computation followed by the mixing can be executed using the following commands.

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

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

The default number of ciphertexts for which pre-computation is performed can be overridden by using the `-maxciph` option, and this is necessary if the default was not set to a positive value as outlined above. All mix-servers must of course use the same value.

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.

Shuffling without decrypting. In some applications it is useful to shuffle the ciphertexts without decrypting. Use the following command to achieve this.

```
vmn -shuffle privInfo.xml protInfo.xml \
    ciphertexts ciphertextsout
```

Decrypting without shuffling. In other applications it is useful to simply decrypt ciphertexts. Strictly speaking this is not a mix-net functionality, but it is a natural functionality if we view the mix-net as a blackbox used to implement electronic voting systems. Use the following command to run the mix-net in this way.

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

Importing a public key. If protocols for key generation and decryption have already been implemented and the user prefers to use these, then the mix-net can still be used to shuffle ciphertexts without decrypting them. To use the mix-net in this way, the user simply sets the public key of the mix-net using the command below, instead of first running the key generation protocol. After this, the mix-net can of course only be used to shuffle ciphertexts as described above, but not to mix or decrypt ciphertexts.

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

3.3 Multiple Sessions*

In some applications it is useful to be able to *run the key generation phase only once* and then re-use the same public key for multiple executions of the mix-net. Each session is identified by an *auxiliary session identifier*, which must consist only of letters `A-Z`, `a-z`, and digits `0-9`.

Use the `-auxsid` option to specify a given auxiliary session identifier. For example, to simply mix a list of ciphertexts in a session with auxiliary session identifier `abc123` you can use the following.

```
vmn -mix -auxsid abc123 privInfo.xml protInfo.xml \
    ciphertexts plaintexts
```

To hide this functionality during basic usage, the auxiliary session identifier defaults to `default`, i.e., the following is equivalent to not specifying any auxiliary session identifier.

```
vmn -mix -auxsid default privInfo.xml protInfo.xml \
    ciphertexts plaintexts
```

3.4 Changing the width of ciphertexts.*

The default width of ciphertexts in the protocol info file can be overridden by using the `-width` option. This may be useful if the width is not known when the keys are generated or if different widths are used in different sessions. For example, to change the width to 3 in the execution of the mix-net, use the following command.

```
vmn -mix -width 3 privInfo.xml protInfo.xml \
    ciphertexts plaintexts
```

Similarly, to override the default width during pre-computation, use the following command.

```
vmn -precomp -width 3 privInfo.xml protInfo.xml
```

Note that all mix-servers must use the same width.

3.5 Rearranging public keys and ciphertexts.*(experimental)

Consider a set of ciphertexts of a given width. It may be useful, e.g., to only decrypt the first component of each ciphertext. The mix-net already supports decrypting ciphertexts of different widths, but a tool is needed to extract the first component. The command `vre` provides this functionality and much more. It allows combining components of ciphertexts from multiple lists of ciphertexts. The command can also extract subsets of a list of ciphertexts or concatenate lists of ciphertexts.

Furthermore, public keys keys may have different key widths. The command can extract components of public keys and combine these components into new public keys. The command provides the corresponding functionality for ciphertexts encrypted using the original public keys.

This command can, e.g., be used to: (1) let two distinct sets of mix-servers run two mix-nets that generate two public keys, (2) combine them into a new single public key, (3) encrypt messages using the combined public key, (4) shuffle the combined ciphertexts, (5) extract the components of each ciphertext encrypted using the first public key, and (6) decrypt the extracted components.

This is only one example, the command `vre` is very powerful and the best way to understand the functionality is reading the usage information and try it out. The usage information is found in Appendix 3.5

This is still experimental functionality where some options may change.

4 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 4.2.

4.1 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 an 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 `com.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 3072)"            \
    "$(vog -gen PRGHeuristic                        \
       "$(vog -gen HashfunctionHeuristic SHA-256)" \
    )"
```

4.2 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.

```
vog -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 `com.verificatum.crypto.PRG`, then the hexadecimal encoding of its seed is stored in the file `.verificatum_random_seed`, or if the environment variable `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.

4.3 Custom Objects*

To allow `vog` to instantiate a custom subclass `CustomClass`, e.g., of `PGroup`, a separate subclass `CustomClassGen` of `com.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 the 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.

5 Demonstrator

The demo directory of the VMN 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 many options of the mix-net. It is also easy to configure it to remotely orchestrate an execution on other servers. For more information we refer the reader to the `README` file and the configuration file `conf` found in the demo directory.

Concrete Example. Appendix F also contains a simple worked example of the commands executed by the operators including how to generate demo ciphertexts using the `vmnd` command.

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 a subdirectory of the `nizkp` subdirectory of the working directory of `vmn`. The subdirectory is named using the auxiliary session identifier of the mixing/shuffling/decryption session, by default this it is named `default` (since this is the default auxiliary session identifier). The contents of this directory can then be verified by anybody who has the necessary knowledge to implement a verification algorithm. Thus, VMN is said to be *universally verifiable*.

6.1 Verificatum Mix-Net Verifier

Let `protInfo.xml` be a protocol info file and let `default` be a directory containing an intermediate values and non-interactive zero-knowledge proof. Then the following commands can be used to verify the correctness of proofs of mixing/shuffling/decryption.

The command `vmnv` is completely silent by default and halt with exit status 0 upon success, and otherwise halt with a non-zero exit status. The `-v` option can be used to turn on verbose output displaying progress and the verifications performed. Syntax errors or incorrect usage gives error messages even if the `-v` option is not used.

Proof of Mixing. The validity of a proof of a mixing can be verified using the following command.

```
vmnv -mix protInfo.xml default
```

This command gives a usage error if the proof is not even a proof of a mixing. The `-nopos` option turns off verification of proofs of shuffles. The `-nodec` option turns off verification of the proof of decryption. This allows verifying a part of a proof.

If pre-computation was used during the execution of the mix-net, then the `-noposc` option can be used to turn off verification of proofs of shuffles of commitments, and the `-ccpos` option can be used to turn off verification of the commitment-consistent proofs of shuffles. These options gives a usage error if pre-computation was not used during the execution producing the proof. It is quite natural to use `-noccpos` and `-nodecr` to verify that the pre-computation phase was performed correctly, but nothing else, and then later using `-noposc` to verify the commitment-consistent proofs of shuffles and the decryption, but not the proofs of shuffles of commitments.

Proof of Shuffling. If the mix-net only shuffled the ciphertexts without decrypting, then the proofs of shuffles can be verified using the following command.

```
vmnv -shuffle protInfo.xml default
```

If pre-computation was used, then the options `-noposc` and `-noccpos` can be used as above. These options gives a usage error if the proof is not the result of an execution with pre-computation.

Proof of Decryption. If the mix-net was only used to decrypt a list of ciphertexts, then the corresponding proof can be verified using the following command.

```
vmnv -decrypt protInfo.xml default
```

Sloppy Verification. It is also possible to verify the correctness of a proof of a mixing/shuffling/decryption without stating which (or the auxiliary session identifier or actual width as described below). This option should *not* be used in real applications, since it is dangerous to not explicitly state the expected type of proof.

```
vmnv -sloppy protInfo.xml default
```

Non-default Parameters. For a given mixing/shuffling/decryption session both the auxiliary session identifier that specifies the session, as well as the width of ciphertexts can be changed. To verify a proof of such a session these values must be given explicitly to the verifier.

SPECIFYING THE AUXILIARY SESSION IDENTIFIER. The auxiliary session identifier is `default` by default in the above calls (it is not derived from the file name). To use a given session identifier `abc123` use the following command instead (and similarly for verifying only shuffling or only decryption).

```
vmnv -mix -auxsid abc123 protInfo.xml abc123
```

CHANGING THE WIDTH OF CIPHERTEXTS. If the default width of ciphertexts specified in the protocol info file was overridden when executing the mix-net, then the width must be passed to the verifier as well. For example, use the following to verify the default session with a non-default width equal to 3.

```
vmnv -mix -width 3 protInfo.xml default
```

6.2 Completing a Verification Using the Verificatum Verifier

The formats of the public key file handed to senders, the file containing the input ciphertexts, and the output file containing decoded plaintexts (or output file containing ciphertexts) may be different from VMN's internal format used to represent the corresponding objects as files in the proof directory.

Thus, to verify the overall correctness of an execution in an application, it must be ensured that the representations are consistent. There is obviously no general way to do this, since the formats used are inherently application dependent. However, one possibility is to use the `vmnc` as explained in Section 7 to convert the objects and then simply use `diff` to compare the output with the expected result in the proof directory.

6.3 Independent Stand-alone Mix-Net Verifiers*

Universal verifiability is of course more interesting if *independent* parties implement stand-alone verifiers. These verifiers should preferably share no code with VMN itself. We suggest that the reader takes a moment to consider the depth of this requirement.

In a companion document [2] targeting programmers of such verifiers, the formats of the files in a proof directory and the algorithms that must be implemented are described in detail. Independent verifiers must of course implement a conversion tool for completing the verification independently as explained in Section 6.2.

7 Converting Public Keys, Ciphertexts, and Plaintexts

The public key resulting from the key generation phase, the input ciphertexts, and the output plaintexts (or output ciphertexts), are represented using the internal format of VMN. This format is detailed in Appendix C.

In practice, the public key and the input ciphertexts may be represented in different ways in different applications, and the output plaintexts (our outputs) are typically decoded into strings somehow. VMN comes with the tool `vmnc` that can be used to perform the needed conversion

or decoding to/from these formats. A custom format can be used[1] by implementing a subclass of `com.verificatum.protocol.ProtocolElGamalInterface`, but there are also a few pre-defined formats described in Appendix D.

The mix-net outputs a public key or accepts an externally generated public key as input, it accepts ciphertexts as input and may also output ciphertexts, but it only outputs plaintexts. Thus, it is possible to specify the input and output formats when converting public keys and ciphertexts, but only the output format when decoding plaintexts. The default value of each interface is `raw`, which is the internal format of VMN.

For example, use the following commands to: (1) convert a public key output by the mix-net to native format, (2) convert input ciphertexts in native format to ciphertexts in raw format for the mix-net, and (3) decode plaintext group elements from the mix-net in raw format to plaintexts in native format.

```
vmnc -pkey -outi native protInfo.xml publicKey publicKeyNative
```

```
vmnc -ciphs -ini native protInfo.xml \
     ciphertextsNative ciphertexts
```

```
vmnc -plain -outi native protInfo.xml \
     plaintexts plaintextsNative
```

8 Acknowledgments

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

References

[1] Digital signature standard (DSS). *Digital signature standard (DSS)*. National Institute of Standards and Technology, Washington, 2000. URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 186-2.

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

[1] If you want to try out your own format in the demo, you can instead implement your own subclass of `com.verificatum.protocol.ProtocolElGamalInterfaceDemo`. This merely adds functionality for generating demo ciphertexts using your format.

A Byte Trees

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

A.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, but the third is not, is

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

A.2 Representation as an Array of Bytes

We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the $8k$-bit two's-complement representation of n in big endian byte order, where n is given in decimal notation. We also use hexadecimal notation for constants, e.g., `0A` means $\mathsf{bytes}_1(10)$. A byte tree is represented by an array of bytes as follows.

- A leaf $\mathsf{leaf}(d)$ is represented by 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 by 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 l, 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
```

A.3 ASCII Strings

ASCII strings are identified with the corresponding byte arrays. No ending symbol is used to indicate the length of the string, since the length of the string is stored in the leaf.

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

A.4 Hexadecimal Encodings

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 a. We denote by $\mathsf{unhex}(s)$ the reverse operation that converts an ASCII string s of an even number of digits `0-9` and `A-F` into the corresponding array of bytes.

B Representations of Arithmetic Objects

Every arithmetic object in VMN is represented as a byte tree. In this section we pin down the details of these representations. We describe how to represent elements in product rings and product groups, as well as arrays of such elements. These products are basically lists of elements and operations are applied element wise.

B.1 Basic Objects

Integers. A multi-precision integer n is represented by $\mathsf{leaf}(\mathsf{bytes}_k(n))$ for the smallest possible integer k.

Example 4. 263 is represented by `01 00 00 00 02 01 07`.

Example 5. -263 is represented by `01 00 00 00 02 FE F9`.

Arrays of booleans. An array $(a_1, \ldots, a_l)$ of booleans is represented as $\mathsf{leaf}(b)$, where b is an array $(b_1, \ldots, b_l)$ of bytes where b_i equals `01` if a_i is true and `00` otherwise.

Example 6. The array $(true, false, true)$ is represented by $\mathsf{leaf}(\texttt{01 00 01})$.

Example 7. The array $(true, true, false)$ is represented by $\mathsf{leaf}(\texttt{01 01 00})$.

B.2 Prime Order Fields

Field element. An element a in a prime order field $\mathbb{Z}_q$ is represented by $\mathsf{leaf}(\mathsf{bytes}_k(a))$, where a is identified with its integer representative in $[0, q-1]$ and k is the smallest possible k such that q can be represented as $\mathsf{bytes}_k(q)$. In other words, field elements are represented using *fixed size* byte trees, where the fixed size only depends on the order of the field.

Example 8. $258 \in \mathbb{Z}_{263}$ is represented by `01 00 00 00 02 01 02`.

Example 9. $5 \in \mathbb{Z}_{263}$ is represented by `01 00 00 00 02 00 05`.

Array of field elements. An array $(a_1, \ldots, a_l)$ of field elements is represented by a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of a_i.

Example 10. The array $(1, 2, 3)$ of elements in $\mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 03
  01 00 00 00 02 00 01
  01 00 00 00 02 00 02
  01 00 00 00 02 00 03
```

B.3 Product Rings

Product ring element. An element $a = (a_1, \ldots, a_k)$ in a product ring is represented by a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_k})$, where $\overline{a_i}$ is the byte tree representation of the component a_i. Note that this representation keeps information about the order in which a product group is formed intact (see the second example below).

Example 11. The element $(258, 5) \in \mathbb{Z}_{263} \times \mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 02
  01 00 00 00 02 01 02
  01 00 00 00 02 00 05
```

Example 12. The element $((258, 6), 5) \in (\mathbb{Z}_{263} \times \mathbb{Z}_{263}) \times \mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 02
  00 00 00 00 02
    01 00 00 00 02 01 02
    01 00 00 00 02 00 06
  01 00 00 00 02 00 05
```

Array of product ring elements. An array $(a_1, \ldots, a_l)$ of elements in a product ring 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.

Thus, the structure of the representation of an array of ring elements mirrors the representation of a single ring element. This seemingly contrived representation turns out to be convenient in implementations.

Example 13. The array $\big((1, 4), (2, 5), (3, 6)\big)$ of elements in $\mathbb{Z}_{263} \times \mathbb{Z}_{263}$ is represented as

```
00 00 00 00 02
  00 00 00 00 03
    01 00 00 00 02 00 01
    01 00 00 00 02 00 02
    01 00 00 00 02 00 03
  00 00 00 00 03
    01 00 00 00 02 00 04
    01 00 00 00 02 00 05
    01 00 00 00 02 00 06
```

B.4 Multiplicative Groups Modulo Primes

Group. A subgroup G_q of prime 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 encoded as four bytes determines how a string is encoded into a group element and can be ignored for the purpose of this document. We stress that $\overline{g}$ is the byte tree representation of g viewed as a group element as defined below.

Group element. An element $a \in G_q$, where G_q is a subgroup of prime 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 14. 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`.

Example 15. Let G_q be the subgroup of order $q = 131$ in $\mathbb{Z}_{263}^*$. Then $3 \in G_q$ is represented by `01 00 00 00 02 00 03`.

B.5 Standard Elliptic Curves over Prime Order Fields

Group. A standard elliptic curve group named FooCurve is represented by the byte tree leaf("FooCurve").

Example 16. The group P-256 from FIPS 186-3 [1] is represented by leaf("P-256").

The following curves are currently implemented by VMN, each defining the order of the underlying field, the curve equation, the order of the group, and the standard generator.

P-192	brainpoolp192r1	prime192v1	secp192k1
P-224	brainpoolp224r1	prime192v2	secp192r1
P-256	brainpoolp256r1	prime192v3	secp224k1
P-384	brainpoolp320r1	prime239v1	secp224r1
P-521	brainpoolp384r1	prime239v2	secp256k1
	brainpoolp512r1	prime239v3	secp256r1
		prime256v1	secp384r1
			secp521r1

Group element. Let the curve be defined over a prime order field $\mathbb{Z}_p$ and let k be the smallest integer such that p can be represented as $\mathsf{bytes}_k(p)$. Then an affine point $P = (x, y)$ on the curve is represented by $\mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_k(x)), \mathsf{leaf}(\mathsf{bytes}_k(y)))$ and the point at infinity is represented by $\mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_k(-1)), \mathsf{leaf}(\mathsf{bytes}_k(-1)))$. Note that a fixed-size representation of -1 is used.

B.6 Arrays of Group Elements and Product Groups

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

Product group element. An element $a = (a_1, \ldots, a_k)$ in a product group is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_k})$, where $\overline{a_i}$ is the byte tree representation of a_i.

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.

B.7 Marshalling Groups

When objects are converted to byte trees in VMN, they do not store the name of the Java class of which they are instances. Thus, to recover an object from such a representation additional information must be available. Java serialization would not be language independent. Furthermore, only a few objects must be converted, so we use a simplified scheme where a group G_q represented by an instance of a Java class PGroupClass in VMN is marshalled into a byte tree

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

This byte tree in turn is converted into a byte array which is coded into hexadecimal and prepended with an ASCII comment. The comment and the hexadecimal coding of the byte array are separated by double colons. The resulting ASCII string is denoted by $s = \mathsf{marshal}(G_q)$ and the group G_q recovered from s by removing the comment and colons, converting the hexadecimal string to a byte array, converting the byte array into a byte tree, and converting the byte tree into a group G_q. This is denoted by $G_q = \mathsf{unmarshal}(s)$.

Groups in VMN. Currently, there are two implementations of groups in VMN:

Implementation	Description
`com.verificatum.arithm.ModPGroup`	Multiplicative groups.
`com.verificatum.arithm.ECqPGroup`	Standard elliptic curve groups over prime order fields.

Example 17. The standard NIST curve P-256 [1] is marshalled into

$$\mathsf{node}(\mathsf{leaf}(\texttt{"com.verificatum.arithm.ECqPGroup"}), \mathsf{leaf}(\texttt{"P-256"})) \ .$$

C Raw Format

The internal representation of groups and group elements is used to represent the public key (with an embedded representation of the underlying group), the input ciphertexts, and output plaintext group elements. These objects depend on the underlying group G_q of order q, the key width κ, and the ciphertext width ω. Let $\mathcal{M} = G_q$ be the underlying group, let $\mathcal{M}_\kappa = \mathcal{M}^\kappa$ be the plaintext group, let $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$ be the public key group, and let $\mathcal{C}_{\kappa,\omega} = \mathcal{M}_\kappa^\omega \times \mathcal{M}_\kappa^\omega$ be the ciphertext group.

The public key file contains $\mathsf{node}(\mathsf{marshal}(\mathcal{M}_\kappa), \overline{pk})$, where $pk \in \mathcal{C}_\kappa$ is the public key in the product group. Note that the public key does not depend on the width of the ciphertexts. The array L_0 of input ciphertexts in $\mathcal{C}_{\kappa,\omega}$ is represented on file as $\overline{L_0}$ and the array m of output plaintext group elements in $\mathcal{M}_{\kappa,\omega}$ is represented on file as $\overline{m}$. We stress that the output group elements are not decoded into strings.

The `vbt` command can be used to display the contents of a byte tree in a structured way.

D Alternative Formats

The `vmnc` tool can be used to convert public keys and ciphertexts to and from an application dependent format as well as decoding plaintext group elements in an application dependent way. There are a few built-in format, but it is also easy to implement a custom format and use it with `vmnc`.

D.1 Native Format

The native format converts the binary objects of the raw format to their hexadecimal representation as ASCII strings and does not require the number of ciphertexts in the input file to be available at the start of parsing the file. It also decodes the output plaintext group elements into strings according to the encoding scheme of the underlying group.

More precisely, $\mathsf{hex}(\mathsf{node}(\mathsf{marshal}(\mathcal{M}_\kappa), \overline{pk}))$ is output on the public key file, where pk denotes the public key in $\mathcal{C}_\kappa$. For each line in the file of input ciphertexts an attempt is made to interpret it as $\mathsf{hex}(\overline{w})$ for some ciphertext $w \in \mathcal{C}_{\kappa,\omega}$. Lines for which this fails are ignored. The array m of plaintext group elements in $\mathcal{M}_{\kappa,\omega}$ is 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.

D.2 JSON Format

The JSON format assumes that a subgroup of a multiplicative group modulo a prime is used and that $\kappa = \omega = 1$. i.e., that the underlying group is represented in VMN as an instance of `com.verificatum.arithm.ModPGroup` and that each plaintext is a single group element. 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 $\mathbb{Z}_p^*$. Let $(g, y) = (3, 13)$ be a public key in $G_q \times G_q$. 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 format.

D.3 Custom Format

The custom format of the mix-net is captured by a subclass of `ProtocolElGamalInterface`, found in the `com.verificatum.protocol` package. This class requires every subclass to implement the following methods.

Method	Description
`writePublicKey`	Writes a public key to file, including the underlying group in marshalled form.
`readPublicKey`	Reads a public key from file, including the underlying group in marshalled form.
`writeCiphertexts`	Writes ciphertexts to file.
`readCiphertexts`	Reads ciphertexts from file.
`decodePlaintexts`	Decodes plaintext group elements and writes the result to file.
`standardRandomSource`	Creates a source of randomness, which is needed to be able to check the correctness of some inputs probabilistically.

Please consider the source of, e.g., `ProtocolElGamalInterfaceNative` (in the same package), for an example of a subclass.

E Additional Command Line Tools

VMN comes with a number of additional tools used for debugging the mix-net itself or a standalone verifier. Use each command with the `-h` flag for more information.

Command	Description
vmnd	Generates plaintext group elements.
vbt	Command used to print a byte tree on file as a recursive JSON array with data printed as hexadecimal strings.
vhttp	Command used to run the built-in HTTP server as a stand-alone server. This is mostly used to test the ability of the mix-net to use an external mix-server as its basis of a bulletin board.
vtest	Command used to unit test basic classes.
vdemo	Command used to run demos for subprotocols, where an accompanying demo class exists. This is used for debugging subprotocols.
vspt	Lists safe primes in a given interval of integers.

F Worked Example with Three Mix-Servers

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

mkdir -p mydemo/01
mkdir -p mydemo/02
mkdir -p mydemo/03

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

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

cd mydemo/01
vmni -prot -sid "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2

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

vmni -party -name "Mix-server 01" \
-http http://localhost:8041 \
-hint localhost:4041

# Copy protocol info files using out-of-bound channel.
cp localProtInfo.xml protInfo01.xml
cp protInfo01.xml ../02/
cp protInfo01.xml ../03/

#### 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 -ciphs publicKey 100 ciphertexts
cp ciphertexts ../02/
cp ciphertexts ../03/

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

```
vmn -mix ciphertexts plaintexts


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

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

cd mydemo/02
vmni -prot -sid "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2

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

vmni -party -name "Mix-server 02" \
-http http://localhost:8042 \
-hint localhost:4042

# Copy protocol info files using out-of-bound channel.
cp localProtInfo.xml protInfo02.xml
cp protInfo02.xml ../01/
cp protInfo02.xml ../03/

#### 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/03
vmni -prot -sid "SessionID" -name "Swedish Election" \
     -nopart 3 -thres 2

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

vmni -party -name "Mix-server 03" \
-http http://localhost:8043 \
-hint localhost:4043

# Copy protocol info files using out-of-bound channel.
cp localProtInfo.xml protInfo03.xml
cp protInfo03.xml ../01/
cp protInfo03.xml ../02/

#### 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


################## Verify Zero-Knowledge Proof ######################

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

G Usage Information for `vtm`

```
Usage:
 vtm -h
 vtm vog|vmni|vmn|vmnv|vre|vmnc <params>...
 vtm vbt|vmnd <params>...
 vtm -version


Description:

Executes the provided commands. Each command can be executed on its own as
well. A brief description of each command follows. Please execute any
particular command with "-h" to get the full usage information.

 vog  - Provides a uniform interface to all primitive objects that can be
        generated and used in initialization files of protocols or as inputs
        to other calls to this tool.
 vmni - Generates configuration files for the Verificatum Mix-Net.
 vmn  - Executes the Verificatum Mix-Net.
 vmnv - Verifies the overall correctness of an execution of the Verificatum
        Mix-Net.
 vre  - Re-arranges inputs and outputs of the mix-net in various ways.
 vmnc - Plug-in based tool for converting public keys, ciphertexts, and
        plaintexts from/to custom formats.
 vbt  - Reads byte tree data and prints it as a nested JSON array or reads
        data and verifies that it is a valid byte tree.
 vmnd - Generates demo ciphertexts for the given interface.
```

H Usage Information for `vmni`

```
Usage:
 vmni -h
      [-bullboard <value>]
 vmni -prot -sid <value> -name <value> -nopart <value> -thres <value>
      [-bullboard <value>] [-cerr] [-corr <value>] [-descr <value>] [-e]
      [-ebitlen <value>] [-ebitlenro <value>] [-keywidth <value>]
      [-maxciph <value>] [-pgroup <value>] [-prg <value>] [-rohash <value>]
      [-statdist <value>] [-vbitlen <value>] [-vbitlenro <value>]
      [-width <value>]
      <protInfoOut>
 vmni -party -name <value>
      [-arrays <value>] [-cerr] [-cert <value>] [-descr <value>]
      [-dir <value>] [-e] [-hint <value>] [-hintl <value>] [-http <value>]
      [-httpdir <value>] [-httpl <value>] [-httptype <value>] [-keygen <value>]
      [-nizkp <value>] [-pkey <value>] [-rand <value>] [-seed <value>]
      [-skey <value>] [-srtbyrole <value>]
      <protInfoIn> <privInfo> <protInfoOut>
 vmni -merge
      [-cerr] [-e]
      <protInfoIn> ... <protInfoOut>
 vmni -digest
      [-hash <value>]
      <file>
 vmni -schema <type>
 vmni -version


Description:

This command is used to generate the configuration file of a protocol in
three simple steps:

   (1) Each party generates a stub protocol info file with the
       global parameters.
   (2) Each party generates private and protocol info files.
   (3) Each party merges all protocol info files into a single
       protocol info file.

The options "-prot", "-party", and "-merge" must appear as the first option.
Use "-prot" to generate the global stub file containing only the global
parameters that all parties agree on. For example:

   vmni -prot -sid "SID" -name "Execution" -nopart 3 -thres 2 stub.xml

Use "-party" to generate: (i) a private info file containing your private
parameters, e.g., your secret signing key, and (ii) a new protocol info file
based on the input protocol info stub file, where all your public
information has been added. For example:

   vmni -party -name "Santa Claus" stub.xml privInfo1.xml protInfo1.xml

(If you use a PRG as the "-rand" (random source), then use the "-seed"
option as well and specify a seed file (or device) containing a seed of
suitable length.)

Use "-merge" to generate a single protocol info file from several protocol
info files with identical joint parameters. Assuming that the ith party
names its info files as above:
```

```
   vmni -merge protInfo1.xml protInfo2.xml protInfo3.xml protInfo.xml
```

All optional values have reasonable defaults, i.e., you can actually use the above commands provided that /dev/urandom contains good randomness. Please generate dummy files to investigate exactly what these defaults are. It is unwise to touch the defaults unless you know exactly what you are doing.

The stub filename can be dropped when the "-prot" option is used, in which case it defaults to "stub.xml". Similarly, the filenames can be dropped when using the "-party" option, in which case they default to "stub.xml", "privInfo.xml", and "localProtInfo.xml". The name of the output joint protocol info file can also be dropped when using the "-merge" option, in which case it defaults to "protInfo.xml"

WARNING!

All basic inputs are verified to be of the right form and within reasonable ranges already at a syntactical XML level, but due to the flexibility of the protocols certain combinations of inputs can make the protocol:

(1) Insecure in an application due to too small parameters. The default parameters are conservative, but the user is responsible for checking that these, or any custom parameters provides the expected level of security.

(2) Computationally infeasible to execute due to too large parameters. In this case the software will stall without output.

Furthermore, some inputs such as paths, URLs, and plugin classes can not be fully validated at a syntactical XML level, i.e., only upper bounds on input lengths and alphabets used can be verified.

They are of course validated at the application level, but since these objects can be instantiations of custom classes implemented by third parties the responsibility for this validation may fall outside the scope of this software. All such inputs are clearly marked with "WARNING!" in the descriptions below.

Parameters:

```
   <file>           - Info file.
   <privInfo>       - Private info output file.
   <protInfoIn>     - Protocol info file containing joint parameters and
                      possibly some party info entries.
   <protInfoOut>    - Protocol info output file.
```

Options:

```
   -arrays <value>  - Determines if arrays of group/field elements and integers
                      are stored in (possibly virtual) RAM or on file. The
                      latter is only slighly slower and can accomodate larger
                      arrays ("ram" or "file").
-bullboard <value>  - Name of bulletin board implementation used, i.e., a
                      subclass of com.verificatum.protocol.com.BullBoardBasic.
                      WARNING! This field is not validated syntactically.
     -cerr          - Print error messages as clean strings without any error
                      prefix or newlines.
     -cert <value>  - Certainty with which probabilistically checked parameters
                      are verified, i.e., the probability of an error is
                      bounded by 2^(-cert). This must be a positive integer at
                      most equal to 256.
     -corr <value>  - Determines if the proofs of correctness of an execution
                      are interactive or non-interactive. Legal valus are
```

```
                    "interactive" or "noninteractive".
    -descr <value>  - Description of this protocol execution. This is merely a
                      longer description than the name of the protocol
                      execution. It must satisfy the regular expression |[A-Za-
                      z][A-Za-z0-9:;?!.()\[\] ]{0,4000}.
   -digest          - Compute hexadecimal encoded digest of file.
      -dir <value>  - Working directory of this protocol instance. WARNING!
                      This field is not validated syntactically.
       -e           - Print exception trace upon error.
  -ebitlen <value>  - Bit length of each component in random vectors used for
                      batching.
-ebitlenro <value>  - Bit length of each component in random vectors used for
                      batching in non-interactive random-oracle proofs.
        -h          - Display usage information
     -hash <value>  - Name of an algorithm from the SHA-2 family, i.e., SHA-
                      256, SHA-384, or SHA-512, used to compute a digest of an
                      info file. (Default is SHA-256.)
     -hint <value>  - Socket address given as <hostname>:<port> or <ip address>:
                      <port> to our hint server. A hint server is a simple UDP
                      server that reduces latency and traffic on the HTTP
                      servers.
    -hintl <value>  - Socket address given as <hostname>:<port> or <ip address>:
                      <port>, where our hint server listens for connections,
                      which may be different from the address used to access
                      it, e.g., if it is behind a NAT.
     -http <value>  - URL to the HTTP server of this party.
  -httpdir <value>  - Root directory of HTTP server. WARNING! This field is not
                      validated syntactically.
    -httpl <value>  - URL where the HTTP-server of this party listens for
                      connections, which may be different from the HTTP address
                      used to access it, e.g., if it is behind a NAT.
 -httptype <value>  - Decides if an internal or external HTTP server is used.
                      Legal values are "internal" or "external".
   -keygen <value>  - Determines the key generation algorithm used to generate
                      keys for the CCA2-secure cryptosystem with labels used in
                      subprotocols. An instance of com.verificatum.crypto.
                      CryptoKeyGen. WARNING! This field is not validated
                      syntactically.
 -keywidth <value>  - Width of El Gamal keys. If equal to one the standard El
                      Gamal cryptosystem is used, but if it is greater than
                      one, then the natural generalization over a product group
                      of the given width is used. This corresponds to letting
                      each party holding multiple standard public keys.
  -maxciph <value>  - Maximal number of ciphertexts for which precomputation is
                      performed. Pre-computation can still be forced for a
                      different number of ciphertexts for a given session using
                      the "-maxciph" option during pre-computation.
    -merge          - Merge several protocol info files with identical joint
                      parameters into a single protocol info file.
     -name <value>  - Name of this protocol execution. This is a short
                      descriptive name that is NOT necessarily unique, but
                      satisfies the regular expression [A-Za-z][A-Za-z0-9_ ]
                      {1,255}.
    -nizkp <value>  - Destination directory for non-interactive proof. Paths
                      are relative to the working directory or absolute.
                      WARNING! This field is not validated syntactically.
   -nopart <value>  - Number of parties taking part in the protocol execution.
                      This must be a positive integer that is at most 25.
    -party          - Generate private and protocol info files based on the
                      given protocol info stub file.
   -pgroup <value>  - Group over which the protocol is executed. An instance of
```

```
                    a subclass of com.verificatum.arithm.PGroup.
      -pkey <value>  - Public signature key (instance of subclasses of com.
                       verificatum.crypto.SignaturePKey). WARNING! This field is
                       not validated syntactically.
       -prg <value>  - Pseudo random generator used to derive random vectors for
                       batchingfrom jointly generated seeds. This can be "SHA-
                       256", "SHA-384", or "SHA-512", in which case com.
                       verificatum.crypto.PRGHeuristic is instantiated based on
                       this hashfunction, or it can be an instance of com.
                       verificatum.crypto.PRG. WARNING! This field is not
                       validated syntactically.
      -prot          - Generate protocol info stub file containing only joint
                       parameters.
      -rand <value>  - Source of randomness (instance of com.verificatum.crypto.
                       RandomSource). WARNING! This field is not validated
                       syntactically and it is impossible to verify that a
                       random device points to a source of randomness suitable
                       for cryptographic use, or that a pseudo-random generator
                       has been initialized with such randomness
    -rohash <value>  - Hashfunction used to implement random oracles. It can be
                       one of the strings "SHA-256", "SHA-384", or "SHA-512", in
                       which case com.verificatum.crypto.HashfunctionHeuristic
                       is instantiated, or an instance of com.verificatum.crypto.
                       Hashfunction. Random oracles with various output lengths
                       are then implemented, using the given hashfunction, in
                       com.verificatum.crypto.RandomOracle.
                       WARNING! Do not change the default unless you know
                       exactly what you are doing. This field is not validated
                       syntactically.
    -schema <type>   - Output the XML schema definition of private or protocol
                       files. Legal values are "private" and "protocol".
      -seed <value>  - Seed file for pseudo-random generator of this party.
       -sid <value>  - Session identifier of this protocol execution. This must
                       be globally unique and satisfy the regular expression [A-
                       Za-z][A-Za-z0-9]{1,1023}.
      -skey <value>  - Pair of public and private signature keys (instance of
                       com.verificatum.crypto.SignatureKeyPair). WARNING! This
                       field is not validated syntactically.
 -srtbyrole <value>  - Sorting attribute used to sort parties with respect to
                       their roles in the protocol. This is used to assign roles
                       in protocols where different parties play different roles.
  -statdist <value>  - Statistical distance from uniform of objects sampled in
                       protocols or in proofs of security. This must be a non-
                       negative integer at most 256.
     -thres <value>  - Threshold number of parties needed to violate the privacy
                       of the protocol, i.e., this is the number of parties
                       needed to decrypt. This must be positive, but at most
                       equal to the number of parties.
   -vbitlen <value>  - Bit length of challenges in interactive proofs.
 -vbitlenro <value>  - Bit length of challenges in non-interactive random-oracle
                       proofs.
   -version          - Print the package version.
     -width <value>  - Default width of ciphertexts processed by the mix-net. A
                       different width can still be forced for a given session
                       by using the "-width" option.
```

I Usage Information for vmn

```
Usage:
 vmn -h
 vmn -keygen
     [-cerr] [-e] [-s]
     <privInfo> <protInfo> <publicKey>
 vmn -mix
     [-auxsid <sid>] [-cerr] [-e] [-maxciph <value>] [-s] [-width <value>]
     <privInfo> <protInfo> <ciphertexts> <plaintexts>
 vmn -delete
     [-auxsid <sid>] [-cerr] [-e] [-f] [-s]
     <privInfo> <protInfo>
 vmn -lact
     <privInfo> <protInfo>
 vmn -sact
     <privInfo> <protInfo> <indices>
 vmn -precomp
     [-auxsid <sid>] [-cerr] [-e] [-maxciph <value>] [-s] [-width <value>]
     <privInfo> <protInfo>
 vmn -setpk
     [-cerr] [-e] [-s]
     <privInfo> <protInfo> <publicKey>
 vmn -shuffle
     [-auxsid <sid>] [-cerr] [-e] [-s] [-width <value>]
     <privInfo> <protInfo> <ciphertexts> <ciphertextsout>
 vmn -decrypt
     [-auxsid <sid>] [-cerr] [-e] [-s] [-width <value>]
     <privInfo> <protInfo> <ciphertexts> <plaintexts>
 vmn -version


Description:

Executes the various phases of a mix-net.

In all commands, info file names can be dropped in which case they are
assumed to be "privInfo.xml" and "protInfo.xml" and exist in the current
working directory.

Use "-keygen" to execute the joint key generation phase of the mix-net. This
results in a joint public key. All other invocations of the mix-net are tied
to a particular session as determined by the "-auxsid" option (or lack
thereof in which case it defaults to "default").

Use "-setpk" to only use the mix-net for shuffling using an externally
generated public key.

Use "-mix" to shuffle and decrypt the input ciphertexts, i.e., the output is
a list of randomly permuted plaintexts.

Use "-shuffle" to shuffle the input ciphertexts without decrypting, i.e.,
the output is a list of re-encrypted and permuted ciphertexts.

If pre-computation was used, then these commands invoke the faster process
using the pre-computed values.

Use "-decrypt" to only execute the decryption phase of the mix-net, i.e., no
mixing takes place and the output is a list of plaintexts.
```

```
The shuffling and decryption options can also be used to separate the two
phases of the mixing process. Together with the "-delete" option described
below this gives a way to implement milestones after the pre-computation and
after shuffling to avoid redundant processing in the event of a failure or
corruption of a mix-server.

WARNING!
If the mix-net is used in this way, then the user must ensure by other means
that the input to the decryption phase is shuffled by mix-servers of which a
sufficient number are guaranteed to be uncorrupted.

Use "-delete" to completely delete data about a session.

WARNING!
This removes all data permanently. There is no way to recover deleted data.
(You can not keep the pre-computed data for the shuffling, since this is not
necessarily secure to re-use.) To remain faithful to cryptographic theory
YOU MUST CHANGE THE AUXILIARY SESSION IDENTIFIER to make sure that no
encrypted messages are re-used.

Use "-precomp" to pre-compute as much as possible of the shuffling. Note
that this requires interacting with the other mix-servers, so all operators
must do this simultaneously.

To deal with the case of crashed, corrupted, or isolated mix-servers due to
accidents, misuse, or corruption, individual mix-servers can be deactivated
by the remaining mix-servers. This allows completing an execution in a
secure way without these mix-servers. Deactivated mix-servers can be re-
activated before a later session. Thus, the set of active mix-servers can
vary throughout the life time to ensure maximum robustness.

Use "-lact" to print the set of indices of currently active mix-servers.
Use "-sact" to set the list of indices of currently active mix-servers.

The "-width" option can be used to set the ciphertext width of a session and
otherwise it defaults to the width from the protocol info file. This
corresponds to the number of ciphertexts processed as a unit in naively
implementations.

The "-maxciph" option can be used to set the number of ciphertexts for which
pre-computation is performed and otherwise it defaults to the corresponding
value in the protocol info file.

Unless the "-s" option is used, each invocation of the protocol prints
logging information not only to the log file in the working directory of the
mix-server, but also to stdout. The time entries of each line in the log
file must be interpreted with great care, since certain operations take
place at the same time in separate threads and some operations are pre-
computed in this way. Time measurements are printed at the end of the
logging information.

Parameters:
   <ciphertexts>    - Ciphertexts to be mixed.
   <ciphertextsout> - Mixed ciphertexts.
   <indices>        - The value must be a set described as a braced comma-
                      separated list of distinct indices, where an index i is
                      an integer 1<=i<=k and k is the total number of parties.
   <plaintexts>     - Resulting plaintexts from mixnet.
   <privInfo>       - Private info file.
   <protInfo>       - Protocol info file.
   <publicKey>      - Destination of public key.
```

```
Options:
   -auxsid <sid>    - Auxiliary session identifier used to distinguish
                      different sessions of the mix-net. This must consist of
                      letters a-z, A-Z, and digits 0-9. If this option is not
                      used, then the auxiliary session identifier defaults to
                      "default". Thus, there is a session identifier for every
                      execution.
     -cerr          - Print error messages as clean strings without any error
                      prefix or newlines.
  -decrypt          - Decrypt the input ciphertexts without mixing.
   -delete          - Delete the given session. WARNING! There is no way to
                      recover the data once it has been deleted.
        -e          - Print exception trace upon error.
        -f          - Force an interactive option to become non-interactive by
                      silently assuming an affirmative response from the user.
        -h          - Print usage information.
   -keygen          - Execute joint key generation.
     -lact          - List indices of currently active servers.
  -maxciph <value>  - Maximal number of ciphertexts for which pre-computation
                      is performed. This defaults to the value given in the
                      protocol info file.
      -mix          - Mix the input ciphertexts using the given session. If pre-
                      computation was used previously, then the pre-computed
                      values are used to speed up the mixing.
  -precomp          - Perform joint pre-computation for a given session.
        -s          - Silent mode, i.e., do not print any output on stdout.
     -sact          - Set the set of active mix-servers.
    -setpk          - Set an externally generated public key to be used during
                      shuffling (without decrypting). The key must be given in
                      the raw format for the group specified in the info file
                      and with the proper key width. Consider using the vmnc
                      command to convert public keys in other formats.
  -shuffle          - Shuffle the input ciphertexts without decrypting. If pre-
                      computation was used previously, then the pre-computed
                      values are used to speed up the shuffling.
  -version          - Print the package version.
    -width <value>  - Number of ciphertexts shuffled as a single block. This
                      defaults to the value in the protocol info file.
```

J Usage Information for vog

```
Usage:
 vog -h
 vog -list
     [-pkgs <names>]
     [<classname>]
 vog -gen
     [-cerr] [-e] [-pkgs <names>]
     <classname>
     [<parameters>]
 vog -tem
     <shellcmd>
 vog -rndinit
     [-seed <file>]
     <classname>
     [<parameters>]
 vog -version


Description:

This command provides a uniform interface to all objects that can be
generated and used in initialization files of protocols or as inputs to
other calls to this tool, e.g., cryptographic keys, collision-free hash
functions, etc.

The two most important options are: "-list" which lists, for a given class,
all its sub-classes/interfaces, and "-gen" which invokes the generator of
the given class. For such a class the option "-h" should give a usage
description. For example, the following describes the possible ways of
generating subgroups of multiplicative groups.

   vog -gen ModPGroup -h

Some classes require an instance of another class as input. Using shell-
quoting it is possible to write any such invokation as a single shell
command. In Bash you can quote with "$(" and ")" and generate a instance of
Pedersen's collision-free hash function as follows.

   vog -gen HashfunctionPedersen -width 2 \
           "$(vog -gen ModPGroup -fixed 2048)"

The "-rndinit" option can only be used once. It initializes the source of
randomness used by the Verificatum command line tools in all future
invokations. If this option has not been used at all, then the calls that
needs a random source complains, but all other calls complete without
errors. The random source can either be a wrapper of a device, e.g.,
/dev/urandom or a hardware randomness generator mounted as a device, or it
can be a software PRG. There are pros and cons of each. A notable advantage
of the latter is that it is platform and JVM independent, whereas, e.g.,
/dev/urandom, is not.

We intentionally force the user to make an explicit choice and we never rely
on the builtin java.security.SecureRandom for security critical operations,
since this is both platform and JVM dependent and poorly documented.

That said, the following, which uses the /dev/urandom device as a source of
bits, is usually a reasonable default, but please make sure that this is the
case on your platform before you use it.
```

```
   vog -rndinit RandomDevice /dev/urandom

Similarly, the random source can be initialized as SHA-256 in "counter mode"
as follows, given a seed file.

   vog -rndinit -seed seedfile PRGHeuristic

Note that the seed file should contain hexadecimal encoded random bytes. The
seed file itself will be deleted when the random source is initialized to
avoid accidental reuse.

On many Un*xes such a seed file can be generated as follows by reading from
a device /dev/mydevice, but rolling a die is perfectly practical, since it
is only done once.

   head -c 2048 < /dev/mydevice | hexdump -e '"%x"' > seedfile

Some usage examples:
   vog -list PRG                       # Sub-classes/interfaces of PRG.
   vog -gen PRGHeuristic               # SHA-256 with counter.
   vog -gen ModPGroup -fixed 1024      # Squares modulo safe prime.

Parameters:
   <classname>      - Name of class that allows generation.
   <parameters>     - Parameters of generator of class named <classname>.
   <shellcmd>       - Shell command turned into template.

Options:
     -cerr          - Print error messages as clean strings without any error
                      prefix or newlines.
        -e          - Print exception trace upon error.
      -gen          - Invoke generator of class <classname>.
        -h          - Print usage information.
     -list          - List subclasses of class <classname> with descriptions.
     -pkgs <names>  - Packages searched. Given as a colon-separated list of
                      full class names; one class/interface contained in each
                      package to be searched.
  -rndinit          - Initialize the random source used by this command.
     -seed <file>   - File containing truly random bits (master seed).
      -tem          - Make a template for the given parameters, i.e., a shell
                      command (only for debugging).
  -version          - Print package version.
```

K Usage Information for `vmnv`

```
Usage:
 vmnv -h
 vmnv -c
 vmnv -th
 vmnv -mix
      [-a <value>] [-auxsid <value>] [-e] [-noccpos] [-nodec] [-nopos]
      [-noposc] [-t <names>] [-v] [-wd <dir>] [-width <value>]
      <protInfo> <nizkp>
 vmnv -shuffle
      [-a <value>] [-auxsid <value>] [-e] [-noccpos] [-noposc] [-t <names>]
      [-v] [-wd <dir>] [-width <value>]
      <protInfo> <nizkp>
 vmnv -decrypt
      [-a <value>] [-auxsid <value>] [-e] [-t <names>] [-v] [-wd <dir>]
      [-width <value>]
      <protInfo> <nizkp>
 vmnv -sloppy
      [-a <value>] [-e] [-t <names>] [-v] [-wd <dir>]
      <protInfo> <nizkp>
 vmnv -mc
      <command>
      [<flags>]
 vmnv -version


Description:

Verifies the overall correctness of an execution using the intermediate
results and the zero-knowledge proofs of correctness using the Fiat-Shamir
heuristic in the given proof directory. The verification of certain parts
can be turned off to simplify a limited form of online verification and
simplify debugging of other verifiers.

WARNING!
Using this in a real election gives SOME assurance, but it does NOT
eliminate the need for an independently implemented verifier according to
the human-readable description of the universally verifiable proof resulting
from an execution of the mix-net. This document is available at https://www.
verificatum.org.

The main motivations of this tool are to:
(a) debug the description of the universally verifiable proof,
(b) benchmark the running time of verifiers,
(c) serve as a reference implementation to implementors of
    their own verifiers, and
(d) check the compatibility of independent verifiers with
    the requirements of the description of the universally
    verifiable proof.

For this purpose it provides a feature-rich way to print test vectors of
intermediate results and express compatibility.

Parameters:
   <command>        - Command name of independent verifier. The name may not
                      contain any "-" characters.
   <flags>          - A comma-separated list of option flags to be removed
                      from the compatibility usage information. The following
                      flags are available:
```

```
                      -nopre -mix -shuffle -decrypt -width
                      -nopos -nodec -noposc -noccpos
   <nizkp>          - Directory containing the non-interactive zero-knowledge
                      proof of correctness using the Fiat-Shamir heuristic.
   <protInfo>       - Protocol info file.

Options:
        -a <value>  - Determines if file based arrays are used or not. Legal
                      values are "file" or "ram" and the default is "file".
   -auxsid <value>  - Verify that the given auxiliary session identifier
                      matches that in the proof. This is required when the
                      auxiliary session identifier in the proof is not
                      "default".
         -c         - Print compatibility usage information.
  -decrypt          - Check proof of decryption.
         -e         - Show stack trace of an exception.
         -h         - Print usage information.
        -mc         - Print modified compatibility usage information. This can
                      be used by others to print the usage information that
                      their own verifiers must provide. Partial implementations
                      can remove certain functionality using flags.
       -mix         - Check proof of mixing.
  -noccpos          - Turn off verification of commitment-consistent proofs of
                      shuffles. This is only possible if pre-computation was
                      used during execution.
    -nodec          - Turn off verification of proof of decryption.
    -nopos          - Turn off verification of proofs of shuffles. If pre-
                      computation is used, this turns off verification of both
                      proofs of shuffles of commitments and commitment-
                      consistent proofs of shuffles.
   -noposc          - Turn off verification of proofs of shuffles of
                      commitments. This is only possible if pre-computation was
                      used during execution.
  -shuffle          - Check proof of shuffle.
   -sloppy          - Check proof of mixing/shuffle/decryption depending on
                      what is specified in the proof itself using the auxiliary
                      session identifier and width specified in the proof
                      itself. WARNING! If these values are not verified using
                      other means, then this does not constitute a complete
                      verification.
        -t <names>  - Print the given comma-separated test vectors. The "-th"
                      option can be used to list the available test vectors.
        -th         - List the available test vectors. The names are chosen to
                      be easily related to the notation used in the document
                      that describes the non-interactive zero-knowledge proof
                      of correctness. In particular for programmers that are
                      familiar with LaTeX.
         -v         - Verbose output, i.e., turn on output.
  -version          - Print the package version.
       -wd <dir>    - Directory for temporary files (default is a unique
                      subdirectory of /tmp/com.verificatum). This directory is
                      deleted on exit.
    -width <value>  - Verify that the given width matches that in the proof.
                      This is required when the width in the proof is different
                      from the width in the protocol info file.
```

L Usage Information for vmnc

```
Usage:
 vmnc -h
 vmnc -pkey
      [-cerr] [-e] [-ini <name>] [-outi <name>] [-sloppy] [-wd <value>]
      <protInfo> <in> <out>
 vmnc -ciphs
      [-cerr] [-e] [-ini <name>] [-outi <name>] [-sloppy] [-wd <value>]
      [-width <value>]
      <protInfo> <in> <out>
 vmnc -plain
      [-cerr] [-e] [-outi <name>] [-sloppy] [-wd <value>] [-width <value>]
      <protInfo> <in> <out>
 vmnc -version


Description:

Converts public keys, ciphertexts, and plaintexts from one representation to
another. The input and output representations are determined by the "-ini"
and "-outi" options.

Possible values of the input and output interfaces are are "raw", "native",
or "json", or the name of a subclass of com.verificatum.protocol.
ProtocolElGamalInterface.

Parameters:
   <in>             - Source file containing object to convert.
   <out>            - Destination of converted object.
   <protInfo>       - Protocol info file.

Options:
     -cerr          - Print error messages as clean strings without any error
                      prefix or newlines.
    -ciphs          - Convert ciphertexts.
        -e          - Print stack trace for exceptions.
        -h          - Print usage information.
      -ini <name>   - Mix-net interface used to represent the input. This
                      defaults to the "raw" interface.
     -outi <name>   - Mix-net interface used to represent the output. This
                      defaults to the "raw" interface.
     -pkey          - Convert public key.
    -plain          - Decode plaintexts.
   -sloppy          - This changes the behavior such that if the input and
                      output formats are identical, then the contents are
                      copied blindly.
  -version          - Print the package version.
       -wd <value>  - Working directory used for file based arrays. This
                      defaults to a uniquely named subdirectory of /tmp/com.
                      verificatum.
    -width <value>  - Number of ciphertexts considered as a single block. This
                      option overrides the corresponding value in the protocol
                      info file.
```

M Usage Information for `vre`

```
Usage:
 vre -h
 vre -pkeys -noin <value> -format <string>
     [-cerr] [-e] [-wd <string>]
     <pkey> ...
 vre -sub -inter <string>
     [-cerr] [-ciphs] [-e] [-plain] [-wd <string>] [-width <value>]
     <pkey> <file> ...
 vre -cat
     [-cerr] [-ciphs] [-e] [-plain] [-wd <string>] [-width <value>]
     <pkey> <file> ...
 vre -shallow -widths <string> -format <string>
     [-cerr] [-ciphs] [-e] [-plain] [-wd <string>]
     <pkey> <file> ...
 vre -deep -noin <value> -width <value> -format <string>
     [-cerr] [-ciphs] [-e] [-plain] [-wd <string>]
     <pkey> <file> ...
 vre -version


Description:

WARNING! EXPERIMENTAL COMMAND AT THE MOMENT

Recall that VMN can process ciphertexts of any keywidth and width. This
command allows manipulating public keys, ciphertexts, and plaintexts to
change the keywidth and width and to remove elements or combine components
from ciphertexts and plaintexts.

For example, one may want to: (a) combine two different public keys, (b)
shuffle ciphertexts encrypted under the combined key, (c) extract only the
part encrypted under the first key, and (d) decrypt it. The decryption of
the other part may be delayed or conditional This is easily done using calls
to VMN and this command with a suitable format.

Public Keys.

Using the "-pkeys" option, components of public keys can be combined to form
new public keys. A standard public key is an element (g,y) in G x G, but VMN
can also use generalized public keys in G^k x G^k for some keywidth k. This
can also be viewed as a list of independent standard public keys. See
examples of different public keys below. The components of an element of G^k
are indexed from zero.

Arrays of elements of the same type, but different lengths.

The following functionality is available for either arrays of ciphertexts or
plaintexts depending on if the "-ciphs" or "-plain" flag is used. All arrays
must contain, or are generated to contain, elements from the same group.
Thus, these functions operate on complete plaintexts or ciphertexts.

(1) Using the "-sub" option, multiple subarrays can be extracted from
    a single source array by providing intervals of indices. The
    intervals are allowed to intersect.

(2) Using the "-cat" option arrays can be concatenated, i.e., the
    contents of multiple source arrays with elements of the same
    type are output as a single array.
```

Arrays of elements of different types, but identical lengths.

Standard plaintexts are elements in G, but VMN can handle more complex plaintexts that match the generalized public keys mentioned above, i.e., plaintexts may be contained in G^k for some keywidth k and the corresponding ciphertexts are contained in G^k x G^k. As explained above this corresponds to bundling ciphertexts for multiple public keys during processing. One often also want to process a list of ciphertexts as a unit. Thus, this is further generalized to plaintexts in (G^k)^w and ciphertexts in (G^k)^w x (G^k)^w for some width w. This is illustrated below. The following functionality is available:

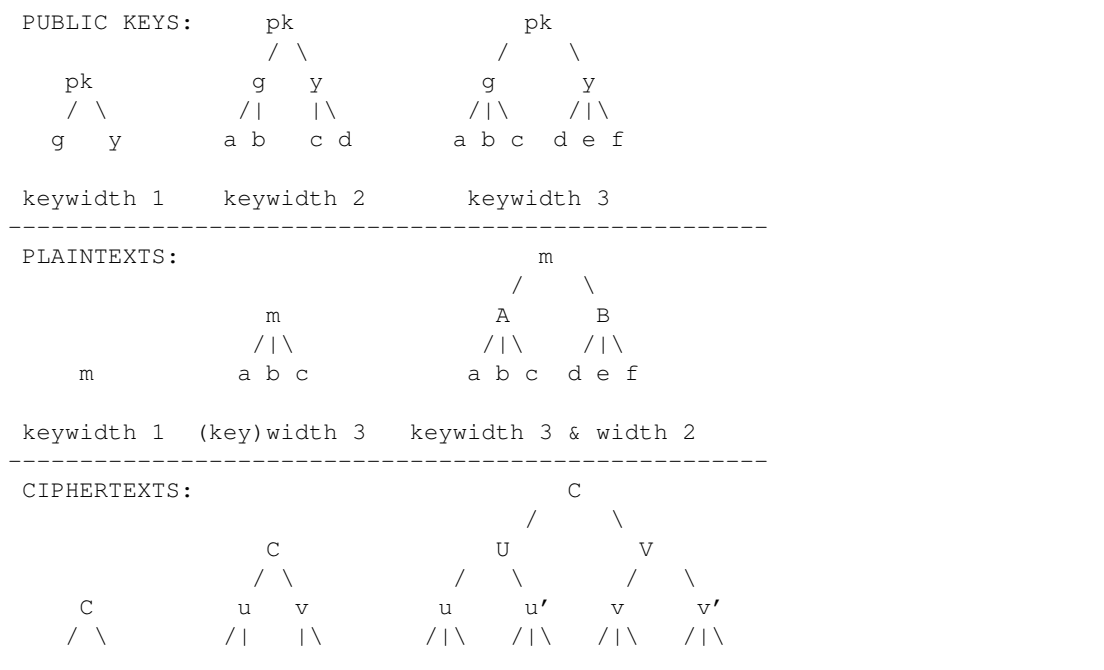
(1) At the shallow level, we think of G^k as a group H and the components of an element in H^w is indexed from zero. Suppose we are given arrays of identical lengths containing elements of the same keywidth, but not necessarily the same widths. Then the "-shallow" option can be used to form a new array by picking components at the shallow level from the input arrays element-wise and forming elements element-wise.

(2) At the deep level, we consider arrays of elements with identical widths, but not necessarily the same keywidth. Using the "-deep" option, arrays can be formed by picking components at the deep level from the input arrays element-wise and forming elements element-wise. We stress that this operation is mapped element-wise to the deepest level. Elements at the deep level are indexed from zero in the same way as for public keys

Thus, the deep level represents the use of multiple public keys in parallel, and the shallow level represents lists of ciphertexts and plaintexts that are processed as a unit by the mix-net (which may of course be defined over multiple public keys used in parallel).

ILLUSTRATION OF PLAINTEXTS AND PUBLIC KEYS

Due to the power and complexity of this command we remind the user of how public keys, ciphertexts, and plaintexts are represented. Consider the following examples:



```
   u   v      a b   c d     a b c d e f g h i j k l

 keywidth 1   keywidth 2     keywidth 3 & width 2
-----------------------------------------------------
Suppose that the basic prime order group is G. Then in the first example the
public keys are defined over: G, G^2, and G^3, respectively. Similarly, the
plaintexts in the second example are contained in: G, G^3, and (G^3)^2,
respectively. Finally, the ciphertexts are contained in: G x G, G^2 x G^2,
and (G^3)^2 x (G^3)^2, respectively.

Parameters:
   <file>           - Input files and output files in that order. Each file
                      contains a public key, a list of ciphertexts, or a list
                      of plaintexts, depending on which usage form is
                      executed.
   <pkey>           - Marshalled public key used to determine the group to
                      which ciphertexts or plaintexts belongs.

Options:
      -cat          - Concatenate input arrays. This requires that all input
                      arrays are defined relative the same group and width. One
                      of the options "-ciphs" or "-plain" is required.
     -cerr          - Print error messages as clean strings without any error
                      prefix or newlines.
    -ciphs          - Re-arrange ciphertexts.
     -deep          - Re-arrange input arrays according to the given format at
                      the deep level. One of the options "-ciphs" or "-plain"
                      is required and the width (number of elements or
                      ciphertexts processed in parallel) must be the same for
                      all inputs.
        -e          - Print stack trace for exceptions.
   -format <string> - Describes which components of the objects stored on the
                      input files should be chosen and how they are combined to
                      form the elements written to the output files. The number
                      of input files is derived from the "-widths" or "-noin"
                      option.

                      The combination of the input files is viewed as a single
                      two dimensional array, i.e., the content of the ith input
                      file is viewed as the ith source (row). The jth component
                      (column) in the ith source is denoted by (i,j). The
                      sources may have different widths.

                      The letter "x" is used to indicate taking the direct
                      product of components (concatenation), e.g., the
                      following denotes the concatenation of the 2nd component
                      of the 1st input file and the 3rd component from the 4th
                      input file: (0,1)x(3,2). To simplify notation multiple
                      sources/indices may be denoted as "s-e", where s is the
                      starting index (inclusive) and e is the ending index
                      (exclusive).

                      The descriptions of the contents of the output files are
                      separated by colons, e.g., the format "(0,0-2):(0-1,4)"
                      states that there are two output files with contents
                      formed as explained above. The number of input files and
                      output files must match the total number of files given
                      as command-line arguments.
        -h          - Print usage information.
    -inter <string> - Colon separated list of descriptions of intervals, i.e.,
                      expressions of the form "s-e", where s >= 0 is the
```

```
                  inclusive starting index and e > s is the exclusive
                  ending index.
   -noin <value>  - Number of input public keys.
  -pkeys          - Re-arrange public keys.
  -plain          - Re-arrange plaintexts.
-shallow          - Re-arrange input arrays according to the given format.
                  One of the options "-ciphs" or "-plain" is required.
    -sub          - Split input array into one or more subarrays. One of the
                  options "-ciphs" or "-plain" is required.
-version          - Print the package version.
     -wd <string> - Working directory used for file based arrays. This
                  defaults to a uniquely named subdirectory of /tmp/com.
                  verificatum.
  -width <value>  - Width that specifies a number of ciphertexts/plaintexts
                  considered as a single block in the input.
 -widths <string> - Comma-separated list of widths, e.g., "2:1:5". Each width
                  specifies a number of ciphertexts/plaintexts considered
                  as a single block in one of the input arrays. The number
                  of input files is determined from the number of entries
                  in the list of widths.
```