A Guide to Ximix Functionality and Implementation

10/02/2014

The Command Applet and the Shuffle Process

The regular Ximix configuration includes a console with an applet for controlling the Ximix network. The applet is primarily for intialising bulletin boards, doing mixes, and then downloading the results. Normal interaction with the `CommandApplet` would be to upload bulletin board data, input a shuffle plan and then trigger a shuffle/download process which will shuffle the selected boards and download the results together with the transcript files that can be used to verify the integrity of the shuffle and the download.

Initialising Bulletin Boards.

The applet is setup to allow a directory to be used as the source of bulletin board data, with one file per bulletin board. On the applet itself this is the “Upload Source Directory” field. Uploading bulletin board data involves specifying the source directory and then selecting “Do Upload”. Once the upload is triggered the boards should be displayed in the table at the bottom of the applet as their upload is completed.

Specifying a Shuffle Plan

The shuffle plan is entered into the “Shuffle Plan” field and consists of a comma separated list of node names. For example a shuffle plan of:

A,A,B,B,C,C,D,D

will specify an 8 part shuffle with a board taking part in the the shuffle been shuffled twice on node A, twice on node B, twice on node C, and twice on node D.

Executing a Shuffle and Download

Having specified a shuffle plan, a shuffle and download is triggered by selecting one or more boards of interest and then clicking on the “Shuffle and Download Selected” button.

The Shuffle and Download Process

The shuffle and download process involves a number of steps.

They are:
1. the shuffle plan is executed on a selected board
2. the transcript of commitments made at each step of the shuffle are downloaded
3. the transcript of selected witness values made at each step of the shuffle are downloaded
4. the commitments matching the witness values are opened and verified
5. the transcripts downloaded are bundled together and loaded up to a threshold number of nodes for the generation of the final decryptions. As each message is received by the client the proofs of correct decryption are logged as received from each node.

Step 1. The shuffle plan is executed.

Shuffles are scheduled and executed by a Ximix client communicating with the board hosting service. In the case of the current implementation the classes involved in these steps are:

`org...client.connection.ClientCommandService$ShuffleOp`

A client-side `Runnable` which guides the shuffling process sending the appropriate messages to the Ximix nodes.

`org...node.mixnet.service.BoardHostingService`

The node class which is the ultimate recipient of messages from the client.

`org...node.mixnet.shuffle.CopyAndMoveTask`

A `Runnable` used by the `BoardHostingService` to copy the board to the initial host for shuffling.

`org...node.mixnet.service.BoardHostingService$ReturnToBoardTask`

A `Runnable` used by the `BoardHostingService` to return the completely shuffled board back to its original board.

`org...node.mixnet.shuffle.TransformShuffleAndMoveTask`

A `Runnable` used the `BoardHostingService` to perform the actual shuffle and generate the appropriate transcripts.

The `TransformShuffleAndMoveTask` is configured partly by the original client message which specifies the transform to use, and partly by the XML configuration for the node which lists the transforms that are available. The commitments and witness values generated with the current configuration are in line with those specified in [1].

Step 2: The transcript of commitments made at each step of the shuffle are downloaded

Shuffle transcripts have two flavours, GENERAL and WITNESS. The download at this step concerns the GENERAL transcript, which is the set of commitments stored at each step of the shuffle. In the case of the current implementation the primary classes involved in this step are:

`org...client.connection.ClientCommandService$DownloadShuffleTranscriptsOp`

A client-side Runnable which controls the download of the shuffle transcripts from the participating Ximix nodes.

`org...node.mixnet.service.BoardHostingService`

The node class which is the ultimate recipient of messages from the client. There are two significant cases in the message processing switch statement DOWNLOAD_SHUFFLE_TRANSCRIPT and DOWNLOAD_SHUFFLE_TRANSCRIPT_STEPS.

The DOWNLOADED_SHUFFLE_TRANSCRIPT_STEPS case establishes whether a particular node has any transcripts for a particular operation and provides a unique query ID which allows the transcript to be downloaded in stages.

The DOWNLOAD_SHUFFLE_TRANSCRIPT case does the actual download of the transcript, keeping track of what has already been downloaded using the query ID. In the case of a general transcript download, the task created in this step will download all the commitments made at a particular stage in the shuffle process.

Step 3: The transcript of selected witness values made at each step of the shuffle are downloaded

The same classes are involved in this step as are involved in step 2 with one important addition. As only a portion of the witness values should be downloaded a generator is created which will only produce a portion of the index values of messages on the board of interest. The current default for this is to use the `SeededChallenger` in the `org.cryptoworkshop.ximix.common.util.challenge` package. This uses a random seed provided by the client and the step number to provide odds/even style so that the same index numbers are never generated on two successive steps, but are still randomly distributed across the board. If ShuffleTranscriptOptions.withPairingEnabled() is set to true when the witness transcript is downloaded the seeded challenger is combined with the PairedChallenger class. In this case 2 consecutive shuffles on the same node should reproduce what is described in Figure 1, Section 1.1 in [1] and section 4.4 in [1].

Step 4: The commitments matching the witness values are opened and verified

Having collected files of commitments and a sampling of witness values, the commitments need to be opened and verified.

This verification process is currently performed using the `ECShuffledTranscriptVerifier` class in the `org.cryptoworkshop.ximix.client.verify` package. This class processes the commitments and witnesses values, using the witnesses to open the commitments and confirm the index number mappings. The commitments also include the randomness used for re-encrypting the ballot, and this is used to also check the cipher texts for consistency. This procedure is broadly outlines in section 5 of [1] as a static approach to validating the results of a shuffle.

Step 5: the transcripts resulting from the shuffle are uploaded to the mixnet for verification by each node holding a share of the network private key.

Assuming the nodes are able to verify the transcript files the messages in the final transcript are then decrypted and downloaded into the client with the appropriate proofs of correct decryption from each node.

In the case of EC El Gamal encryptions there are a few classes involved in this process. They are:

`org...client.connection.ClientCommandService$DownloadShuffleResultOp`

This class is responsible for assembling the partial decrypts as they come in from each node in the threshold process and rebuilding the complete messages and doing an initial verification of correct decryption for each message.

`org...common.crypto.threshold.LagrangeWeightCalculator`

This class is used to calculate the appropriate weights for the partial decrypts so that a full message can be reassembled.

`org...node.crypto.service.NodeShuffledBoardDecryptionService`

This class is the ultimate recipient of the transcript files from the client-side and contains the code for calculating a partial decrypt on messages and associated proof using the proof generator.

`org...node.crypto.service.ProofGenerator`

This class responsible for putting together a non-interactive Zero-Knowledge Proof of correct decryption. The technique used follows the algorithm in [9] with challenge generation been done in line with the Fiat-Shamir heuristic and in consideration of the issues raised in [8].

`org...client.verify.ECDecryptionChallengeVerifier`

Once the download of decrypted messages is complete a final pass is done over the log file produced by the message download process in order to verify it. This is done by the `ECDecryptionChallengeVerifier` which is in the `org.cryptoworkshop.ximix.client.verify` package. The verification calculations in the verifier class are also done in the fashion described [9].

An Example of Board Data Flow

What follows is an example of what would happen to a bulletin board of messages, starting with a file of encrypted ballots, for our purposes called REGION-001. The messages providing the underlying protocol for this process live in the org.cryptoworkshop.common.asn1.message package.

1. CommandApplet used to upload REGION-001

In this case the CommandApplet would be used to open a directory which would contain the REGION-000 ballot file and possibly others. The CommandApplet would use CommandService.createBoard() to create a board on a particular node (the primary), with a backup board on another node (the secondary). We will assume the primary node is node3 (C), and we will assume the secondary is node is node4 (D).

UploadService.uploadMessages() would then be used to upload the ballot data to the primary node.

2. CommandApplet used to specify a shuffle plan of “A, A, B, C, D, D”, the REGION-001 board is selected in the command applet and “Shuffle And Download Selected” is clicked.

In a four node network, initially the disk area associated with the nodes would have looked like this:

```
conf/node1/boards/
conf/node2/boards/
conf/node3/boards/REGION-001
conf/node3/boards/REGION-001.p
conf/node4/boards/REGION-001.backup
conf/node4/boards/REGION-001.backup.p
```

With the the files in node3/boards representing the MapDB files storing the primary board, and the files in node4/boards representing the backup board. The “.p” files are utility files used by MapDB.

Once the shuffle plan has executed the directory structure will look more like this:

```
conf/node1/boards/1386583074860.REGION-001.1
conf/node1/boards/1386583074860.REGION-001.1.p
conf/node1/boards/1386583074860.REGION-001.2
conf/node1/boards/1386583074860.REGION-001.2.p
conf/node2/boards/1386583074860.REGION-001.3
conf/node2/boards/1386583074860.REGION-001.3.p
conf/node3/boards/1386583074860.REGION-001.0
conf/node3/boards/1386583074860.REGION-001.0.p
conf/node3/boards/1386583074860.REGION-001.4
conf/node3/boards/1386583074860.REGION-001.4.p
conf/node3/boards/1386583074860.REGION-001.7
conf/node3/boards/1386583074860.REGION-001.7.p
conf/node3/boards/REGION-001
conf/node3/boards/REGION-001.p
conf/node4/boards/1386583074860.REGION-001.5
conf/node4/boards/1386583074860.REGION-001.5.p
conf/node4/boards/1386583074860.REGION-001.6
conf/node4/boards/1386583074860.REGION-001.6.p
```

```
conf/node4/boards/REGION-001.backup
conf/node4/boards/REGION-001.backup.p
```

In this case what has happened is the network assigned the shuffle operation the operation number 1386583074860. One purpose of the operation number is to provide the internal data in the Operation object returned in the client API call CommandService.doShuffleAndMove() which is needed to download the commitment and witness values making up the shuffle transcripts at the end of the operation. The other purpose is to provide a unique stem for the transit boards associated with the shuffle. The last number in the transit board name represents the step number in the operation, so a transit board with the name:

```
1386583074860.REGION-001.5
```

Represents a transit board for operation number 1386583074860, step number 5.

As the board is hosted on node3 the board REGION-001 is first locked and then a copy of it is made to initialise the shuffle process – this is the step 0 board on node3. As the shuffle plan has node1 (A) as first port of call in the shuffle the transit board for the first processing step, step 1, is on A, and a shuffled version of the step 0 board is sent to A. As the second processing step in the shuffle plan is on node1 as well, the second transit board (step 2) is also on A. As the shuffle proceeds new transit boards are created all the way along to node4 where we have the last step in the shuffle step 6. After step 6 is complete the board is then shuffled back to its home node (step 7) and finally the step 7 board is copied onto the primary board (REGION-001) and transmitted to the back up node as part of the process.

The task that guides this process appears in the ClientCommandService which implements the CommandService interface. The name of the task which is an extension of the Operation class, is ShuffleOp.

At the end of the shuffle the CommandApplet downloads and verifies the transcripts for the shuffle and downloads the board contents, issuing periodic challenges as it goes. The method called for downloading transcripts is the CommandService.downloadShuffleTranscripts() and two types of transcripts are downloaded one, the transcript of commitments is referred to as the GENERAL transcript, the other which includes witness values is the WITNESS transcript. The entire general transcript is downloaded but only a section of the WITNESS transcript is downloaded. In the default case, the manner in which the WITNESS transcript is downloaded is partly restricted by the node configuration and also set by the user using a random seed value which is only revealed when the WITNESS transcript is requested.

The output files in the worked example we are looking at here would be as follows:

```
REGION-001.0.gtr  REGION-001.2.wtr  REGION-001.5.gtr
REGION-001.0.wtr  REGION-001.3.gtr  REGION-001.5.wtr
REGION-001.1.gtr  REGION-001.3.wtr  REGION-001.6.gtr
REGION-001.1.wtr  REGION-001.4.gtr  REGION-001.6.wtr
REGION-001.2.gtr  REGION-001.4.wtr  REGION-001.7.gtr
```

The GENERAL transcripts are represented by .gtr files, the WITNESS transcripts are represented by .wtr files, the proofs of decryption are in the .plg files and the .out file is the decrypted ballots. It is important to note that every step that involves shuffling creates a .wtr file and a .gtr file.

The GENERAL and WITNESS transcripts are then bundled together appropriately and feed into the

ECShuffledTranscriptVerifier for verification.

Finally, assuming everything has passed so far, the board data is decrypted using the CommandService.downloadShuffleResult() methods which takes the transcript files as input, producing a stream of decrypts and proofs of decryption if the transcripts uploaded verify correctly. The task for doing this is also defined in the ClientCommandService class – its name is DownloadShuffleResultOp. The log file produced by the decryption process is can be cross-checked by the ECDecryptionChallengeVerifier class.

The final step produces two additional file:

REGION-001.out   REGION-001.plg

which contains the output of decryption and the proofs of correct decryption respectively.

Elliptic Curve Key Generation

Generation of EC threshold keys is done following the process outlined in Figure 2. of [3] entitled "Protocol New-DKG". The scheme builds on the [4] and [5] to produce a threshold key generation protocol with the capacity to do some validation on the values of shares of the key as they are generated.

Threshold key generation of EC keys involves the following classes:

`org...ximix.client.connection.KeyGenerationCommandService`

This is a client-side class which is an implementation of the `KeyGenerationService` interface. It is responsible for, generating the H value required for New-DKG, kicking off the key generation process and passing the assembled public key back to the requester.

`org...node.crypto.service.NodeKeyGenerationService`

This is the main node class which receives the key generation requests. Its function is really one of a router where it chooses the appropriate key pair generator to call for the algorithm being requested.

`org...node.crypto.key.ECKeyPairGenerator`

This is the business end, it creates a `ECNewDKGGenerator` and kicks off the generation process in the other nodes that are involved. The class then accumulates shares for its host node in the node's `ECKeyManager`.

`org.cryptoworkshop.ximix.node.crypto.key.ECNewDKGGenerator`

This class uses the `ECKeyManager` to generate the local EC key pair and then passes the results onto the `ECNewDKGSplitter` bundling the shares and commitments generated by that into a series of messages for local processing and distribution to other nodes.

`org.cryptoworkshop.ximix.common.crypto.threshold.ECNewDKGSecretSplitter`

This class sits on top of the ShamirSecretSplitter class and calculates the additional commitment values required for validation of the shares as they are distributed according to the method described in Fig. 2. of [3].

`org.cryptoworkshop.ximix.common.crypto.threshold.ShamirSecretSplitter`

This class contains an implementation of the algorithm outlined in [5] for splitting a big integer secret into a set of shares.

`org.cryptoworkshop.ximix.node.crypto.key.ECKeyManager`

The `ECKeyManager` class is where a particular node's shares of a key end up. The class makes use of a countdown latch to ensure that the network public key associated with the distributed private key is not returned until it is fully assembled.

The most interesting method in this class is the `buildSharedKey()` method which is based on the validation section of Fig. 2. of [3]. In it the commitment coefficients associated with each share are validated as the key shares come in and if the shares pass the test they used to build up the local private key share and the network public key.

BLS Signature Generation

BLS signatures were originally described in [6]. The current implementation in Ximix is based on the jPBC library [7]. Unfortunately due to a bug in jPBC it is not currently possible to properly implement “Protocol new-DKG”.

The following classes are involved in BLS signature generation:

`org...client.connection.ClientSigningService`

The ClientSigningService is a client-side router which works out which actual signature generation protocol is required. In the case of BLS signatures it invokes a BLSSigningService.

`org...client.connection.signing.BLSSigningService`

This class manages the collection of the partial signatures to build a complete one to return to the original invoker of the service.

`org...node.crypto.service.NodeSigningService`

The NodeSigningService is the ultimate recipient of the client generated messages associated with signature generation. As with the key generator its job is to route the message to the handler dealing with the messages algorithm. In this case BLSSignerEngine.

`org...node.crypto.signature.BLSSignerEngine`

The BLSSignerEngine generates the actual partial signature using the BLSPrivateKeyOperator class. The result of the PRIVATE_KEY_SIGN case in its switch statement is a share value based on the jPBC Element class.

References

[1] "Making Mix Nets Robust For Electronic Voting By Randomized Partial Checking" Markus Jakobsson, Ari Juels, Ronald L. Rivest, 11th USENIX Security Symposium, 2002.
[2] "A Secure and Optimally Efficient Multi-Authority Election Scheme" R. Cramer, R. Gennaro, B. Schoenmakers, CGS Journal, October, 1997.
[3] “Secure Distributed Key Generation for Discrete-Log Based Cryptosystems” R. Gennaro, S Jarecki, H. Krawczyk, T. Rabin, Journal of Cryptology, 2007.
[4] "A Threhold Cryptosystem without a Trusted Party (Extended Abstract)" T. P. Perdersen, Springer-Verlag, 1998.
[5] “How to share a secret” A. Shamir, Communications of the ACM, November, 1979.
[6] "Short signatures from the Weil pairing" D. Boneh, B. Lynn, and H. Shacham, AsiaCrypt 2001.
[7] The Java Pairing Based Cryptography Library (jPBC) http://sourceforge.net/projects/jpbc/, Downloaded November 2013.
[8] "How not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios" D. Bernhard, O. Pereira, B, Warinschi, volume 7658 of Lecture Notes in Computer Science, pages 626-643. 2012.
[9] "Efficient Cryptographic Protocol Design Based on Distributed El Gamal Encryption" F. Brandt, Proceeding ICISC'05 Proceedings of the 8th international conference on Information Security and Cryptology, Pages 32-47 Springer-Verlag. 2006.