El Gamal Mix-Nets and Implementation of a Verifier
SA104X Degree Project in Engineering Physics

Erik Larsson (erikl3@kth.se)  Carl Svensson (carlsven@kth.se)
Supervisor: Douglas Wikström
Abstract

A mix-net is a cryptographic protocol based on public key cryptography which enables untraceable communication through a collection of nodes. One important application is electronic voting where it enables the construction of systems which satisfies many voting security requirements, including verifiability of correct execution. Verificatum is an implementation of a mix-net by Douglas Wikström.

This report concerns the implementation of a verifier and evaluation of the implementation manual for the Verificatum mix-net. The purpose of the document is to enable third parties to convince themselves that the mix-net has behaved correctly without revealing any secret information. This implementation is a simple version of the verifier using the document and some test vectors generated by the mix-net. The document contains all information but there are still some possibilities for further clarification in order to make it comprehensible to a larger audience.
1 Introduction

Cryptography concerns the theory of secure communication. For this to have any meaning we must define "secure". This is not easy since the most strict definition is not practically feasible and is thus done on a case to case basis depending on the context.

Originally cryptography was interchangeable with the encryption of messages in order to allow secret communication between two parties over an insecure channel. Early examples include various ciphers such as the Caesar and the Vigenère ciphers. These primitive systems achieve the most central property of a cryptographic system, confidentiality, i.e. only parties in possession of the key can encrypt and decrypt a message [1, p. 3]. This is also an example of a symmetric encryption which means that the same piece of information, the key, is used for encryption and decryption.

One problem that arises when only considering confidentiality is that even though a third party cannot read the message, it can be modified without the receiver noticing. One way to counter this is to introduce some kind of "receipt" computed from the message and sent along with the ciphertext. This receipt computed again by the receiver to verify that the message is unaltered. This property is what we call integrity.

In 1976 Whitfield Diffie and Martin Hellman published the first paper on public key cryptography [1, p. 2]. This was a new sort of cryptographic construction that allowed different keys to be used for encryption and decryption of a message. It was the first asymmetric encryption scheme.

Already at this point we have a very powerful set of tools for secret communication. However, one thing which is not covered by traditional public key cryptography is anonymity. By design, two communicating parties must have some kind of knowledge of each other in order to be able to communicate.

A cryptographic construction, called a mix network or mix-net, facilitates the anonymity requirement by routing messages to a cluster of nodes [5, p. 1]. These nodes takes a number of messages from any number of sources, shuffle them around and deliver them to their destination. This description is very brief, partly because there are different flavours of mix-net intended for different purposes.

1.1 Verificatum

The Verificatum mix-net is designed to take a number of encrypted messages, ciphertexts, which have been encrypted with a public key generated by the mix-net, shuffle them around and eventually decrypt them. This can, for example, be used in an election. Every voter encrypts a vote with the public key and posts it to the mix-net. The mix-net takes all the encrypted votes, shuffles and decrypts them. This way all votes are revealed and countable without revealing information about which vote belongs to which voter - an essential property in elections.
Additionally we would like to be able to verify that the mix-net has done its work correctly and not altered any votes. Therefore the mix-net produces some extra data files during its execution. These data files together with the input ciphertexts and the output plaintexts can be run through a verification process that tells if the process has been correctly performed. The neat thing about all this is that it can securely tell whether this is true or false without revealing any more information about the relation between the ciphertexts and the plaintexts, i.e. voter anonymity is preserved.

1.2 Goals and Scope

The idea is that any third party that wants to check the result should be able to implement a verifier for Verificatum. Wikström has created a document describing this verifier [3]. We have, using this document, implemented a verifier to test that the document is complete with all the information needed for the implementation. Because of time constraints, we have limited ourselves to implement a working prototype with only the most common options available, just sufficient to confirm that the specification contain all the necessary parts to implement a verifier. This prototype could later be expanded to facilitate for all features of the Verificatum mix-net. The implementation of the verifier together with an evaluation of the document is a step in making the document correct and accessible.

2 Background

Our work concerns the Verificatum mix-net, VMN, which is an implementation of a cryptographic system called a mix network or mix-net. Before describing VMN and mix-nets in general, we need to learn about the basic building blocks of these compounds. First a common public key cryptographic scheme, called the El Gamal cryptosystem is defined. This scheme constitutes a very central part of VMN. Later on, other cryptographic building blocks are defined. The chapter is concluded by a description of a mix-net.

2.1 El Gamal Cryptography

The El Gamal cryptosystem is based on group theory [1, p. 297], to which a short introduction is provided in Appendix A. In full generality, a public key cryptosystem, or an assymetric cryptosystem, provides two parties the means to communicate privately without the need of sharing a common secret beforehand as is the case with symmetric cryptography. The cryptosystem consists of a public key, a secret key and two functions - one encryption function which uses the public key and one decryption function which uses the secret key [1, p. 283].

In the El Gamal cryptosystem the communicating parties agrees on a suitable cyclic group and a generator element. The secret key is an integer, while the public key
is a group element - the exponentation of the generator by the secret key integer [1, p. 297]. Encryption of a message (note that the messages need to be encoded into elements of the group) is done by multiplying the message with a group element that has a special relation to the public key. Now, only the possessor of the secret key can decrypt, since without the key it is unfeasible to invert this multiplication. This is a slight simplification, but it should capture the main idea of the cryptosystem.

Security of asymmetric cryptosystem depend on the assumption that there exist computationally difficult problems [1, p. 306]. This is also the case for the El Gamal cryptosystem of which the security depends on the difficulty of deducing information about the discrete logarithm in a cyclic group [1, p. 294].

As we shall see, the El Gamal cryptosystem possesses some special properties that will be useful in creating a mix-net.

2.1.1 Definition

The El Gamal cryptosystem is defined over a cyclic group \( G_q = \langle g \rangle \) of prime order \( q \), generated by \( g \in G_q \). A private key \( sk = x \in \mathbb{Z}_q \) is chosen randomly and is used to compute the public key \( pk = (g, y) \in G_q \times G_q \) where \( y = g^x \).

Encryption of a plaintext \( m \in G_q \), denoted \( Enc_{pk}(m, s) \), is done by choosing a random \( s \in \mathbb{Z}_q \) and computing

\[
Enc_{pk}(m, s) = (u, v) \in G_q \times G_q
\]

where \( u = g^s \) and \( v = y^s m \). Decryption of a ciphertext \( (u, v) \in G_q \times G_q \), denoted \( Dec_{sk}(u, v) \), is achieved by using the private key \( x \) to compute

\[
Dec_{sk}(u, v) = u^{-x} v = (g^s)^{-x} y^s m = (g^x)^{-s} y^s m = y^{-s} y^s m = m
\]

One common choice of group to use in the El Gamal cryptosystem is a multiplicative subgroup \( G_q \subset \mathbb{Z}_p^* \), where \( p = kq + 1 \), for some \( k \). Another common choice is to use certain elliptic curves, in which case one may obtain the same security with a smaller group and hence a more space efficient implementation [1, p. 297].

2.1.2 Security

In order to use a cryptosystem in good conscience we need to address the issue of security. One commonly used security definition is semantic security. A cryptosystem is said to be semantically secure if any efficient (probabilistic, polynomial time) algorithm cannot with non-negligible probability distinguish between the encryption of two different plaintexts [1, p. 306]. This means that if an attacker is given the encryption of one of two possible plaintexts he or she will not be able to tell which plaintext was encrypted better than just guessing. The semantic security of the El Gamal cryptosystem relies on the Decisional Diffie-Hellman assumption [4], which is explained below.
Let \( b = g^a \in G_q \) where \( a \in \mathbb{Z}_q \), then \( a \) is said to be the discrete logarithm of \( b \) in the group \( G_q \). There is currently no known efficient classical algorithm that given a general group \( G_q \), \( g \) and \( b \) as above, is able to calculate \( a \) in a reasonable amount of time [1, p. 103].

The Decisional Diffie-Hellman assumption concerns a problem related to the discrete logarithm. In any group \( G_q \) the assumption means that if \( a, b, c \in \mathbb{Z}_q \) are chosen randomly, every efficient algorithm, on input \( g^a, g^b \) and \( y \in \{ g^{ab}, g^c \} \), is unable to tell if \( y = g^{ab} \) or \( y = g^c \) [4].

The semantic security of the El Gamal cryptosystem relies on the Decisional Diffie-Hellman assumption in finite cyclic groups \( G_q \). This means that the El Gamal cryptosystem is secure as long as the assumption is true [4].

### 2.1.3 Properties

If \( G_q \) is a group, then so is \( G_q \times G_q \) with the group operation defined as \( (a, b)(c, d) = (ac, bd) \) for any \( (a, b), (c, d) \in G_q \times G_q \). From this and the definition of the El Gamal cryptosystem, one can deduce that it is homomorphic. This means that for any two messages \( m_1, m_2 \in G_q \) and random numbers \( s_1, s_2 \in \mathbb{Z}_q \)

\[
\text{Enc}_{pk}(m_1, s_1)\text{Enc}_{pk}(m_2, s_2) = (g^{s_1}, y^{s_1} m_1)(g^{s_2}, y^{s_2} m_2) =
\]

\[
= (g^{s_1+s_2}, y^{s_1+s_2} m_1 m_2) = \text{Enc}_{pk}(m_1 m_2, s_1 + s_2)
\]

that is, the encryption of the product of two messages equals the product of the encryptions of the messages [1, p. 289]. In particular, by choosing \( m_1 = m \) and \( m_2 = 1 \) one obtains

\[
\text{Enc}_{pk}(m, s_1)\text{Enc}_{pk}(1, s_2) = \text{Enc}_{pk}(m, s_1 + s_2)
\]

This homomorphic property of the El Gamal Cryptosystem may be used to reencrypt an already encrypted message. If \( s_1 \in \mathbb{Z}_q \) and \( s_2 \in \mathbb{Z}_q \) are chosen with uniform randomness, then \( s_1 + s_2 \in \mathbb{Z}_q \) will be uniformly random as well. So the distribution of ciphertexts encrypted once will be indistinguishable from the distribution of ciphertexts that have been reencrypted [3, p. 1].

For future convenience we define

\[
\text{ReEnc}_{pk}(c, s) = c \cdot \text{Enc}_{pk}(1, s)
\]

As will become clear, this reencryption function will be used to enable hidden shuffling in mix-nets.
2.2 Cryptographic Primitives

Many cryptographic systems, including VMN, make use of some basic functions, normally called cryptographic primitives. These primitives are functions or objects which possess properties interesting in cryptographic contexts. VMN uses hash functions, pseudo random generators and random oracles, all of which are described in this chapter.

2.2.1 Hash functions

In general, a hash function is an easily computable function that takes an input from an arbitrarily large input space and map it to an element, a hash, in a finite sized hash space [1, p. 321]. As a consequence of this, there will exist several inputs which map to the same hash. Hash functions are used in many different areas of computer science and there are many different kinds tailored to have the properties desired for its particular application. A cryptographic hash function is a hash function with two important properties. First, the hashes are uniformly distributed in the hash space. Simply put, this means that if you try to guess which hash a given input will produce you will never have a significantly better chance than just random guessing. Furthermore, for a cryptographic hash function all of the following are infeasible.

- Finding an input which produces a given hash
- Given an input and its hash, finding another input with the same hash
- Finding two inputs which produce the same hash

In mix-nets any cryptographic hash function could be used. A common choice is the SHA-2 family of hash functions, namely SHA-256, SHA-384 and SHA-512 [3, p. 6]. Their main difference is the number of bits they output, i.e. the size of the hash space which is 256, 384 and 512 respectively.

2.2.2 Pseudo Random Generators

A Pseudo Random Generator, PRG, is a function which takes an initial seed and expands it into a longer sequence of seemingly random data, pseudo random data [1, p. 170]. The output is random in the sense that it should be indistinguishable, for any efficient algorithm, from truly random output but the same seed will always produce the same output.

VMN uses a PRG based on a cryptographic hash function that hashes the seed together with a counter which increases for every iteration [3].
2.2.3 Random Oracles

In theory, a Random Oracle, RO, is a black box which takes an input and returns a truly random output from its output space [2]. It will always respond with the same output for the same input. This is very similar to a hash function but has some subtle differences. An RO is a purely theoretical construct which defines no actual function, only a relationship between inputs and outputs.

In practice though, within VMN a construct called an RO is used. This construct is based on a cryptographic hash function and a seed. The purpose of the seed is to permute the relationship between the inputs and outputs of the hash function and thus creating an easily randomizable RO without coming up with a whole new hash function [3].

2.3 Mix Networks

There are different kinds of mix networks. Here, only reencryption mix-nets will be treated - first on a general level followed by a description of a certain type of reencryption mix-net based on the El Gamal cryptosystem. In any mix-net used in the context of electronic elections, the need for verification of correct execution is essential. How this can be obtained is explored in the end of the section.

2.3.1 Overview

One purpose of mix networks is to provide untraceability to its users. A mix-net may, for example, take as input a list of encrypted messages of different origins. These messages pass through the mix-net and is output decrypted and in a randomized order [5]. This property may be used to enable anonymous voting systems [6].

Different types of mix-nets exist. There are decryption mix-nets and reencryption mix-nets. A reencryption mix-net consists of a number of servers, henceforth the mix servers, which sequentially process the messages by reencrypting the list of messages and outputting them in a randomized order [5]. After passing through all servers, the list of ciphertexts is decrypted and the result is a list of the messages in random order. It should after decryption be infeasible to deduce the position of each element in the initial list. We define this as untraceability [6]. This provides the voters with anonymity.

One use of mix-nets is in the context of electronic voting systems. An electronic election can be performed by the use of a reencryption mix-net as follows [7]

1. The mix servers prepare the mix-net by generating public and secret keys.
2. Each voter encrypts his vote with the public key and appends it to a public list of encrypted votes.
3. In sequential order each mix server takes as input the list of encrypted votes, reencrypts and outputs them in a randomized order, replacing the previous list of encrypted votes.

4. After all mix servers have processed the list, each vote is jointly decrypted and posted on a bulletin board making the outcome of the election universally available.

Notice that the reencryption step is necessary before the actual mixing as if it were omitted, the mixing would merely scramble the list of original cryptotexts, providing no untraceability at all.

2.3.2 El Gamal Mix-Nets

A common choice for reencryption mix-nets is to use some version of the El Gamal cryptosystem [7], since its homomorphic property allows reencryption. A mix-net based on the El Gamal cryptosystem consists of \( k \) mix-net servers mixing the votes of \( n \) voters. Suppose the underlying group is \( G_q \) of prime order \( q \) and with generator \( g \in G_q \). The mix-net works as follows [8]

1. An El Gamal public key \( pk = (g, y) \) is generated.

2. Each voter \( j \) encrypts his vote \( m_j \) to create \( c_{j,0} = Enc_{pk}(m_j) = (g^{r_j}, my^{r_j}) \) for some random \( r_j \in \mathbb{Z}_q \). A list of all encrypted votes \( c_0 = (c_{1,0}, \ldots, c_{n,0}) \) is created.

3. For each mix server \( i \in \{1, \ldots, k\} \), given the input \( c_{i-1} \), a random permutation \( \pi_i \) is chosen and a list

\[
c_i = (\text{ReEnc}_{pk}(c_{\pi_i(1), i-1}), \ldots, \text{ReEnc}_{pk}(c_{\pi_i(n), i-1}))
\]

is returned.

4. The final list \( c_k \) is decrypted using the secret key \( sk = x \in \mathbb{Z}_q \) to produce the output list

\[
(m_{\pi(1)}, \ldots, m_{\pi(n)}) = (\text{Dec}_{sk}(c_{k,1}), \ldots, \text{Dec}(c_{k,n}))
\]

for where \( \pi \) is a permutation \( \pi = \pi_k \circ \ldots \circ \pi_1 \).

The result of the election may now be computed while the origins of the individual votes are unknown. Remark that all encryptions \( Enc_{pk} \) are performed with some randomness \( r \in \mathbb{Z}_q \).

2.3.3 Verification

There are some problems related to electronic voting using mix-nets. One issue is that the mix servers may or may not execute their part of the mix-net properly. For example dishonest servers could completely change the outcome of the election by replacing the
true votes with their own [7]. A first solution may be to make sure that every server is reliable. This is difficult, however, since there is always a risk that a server is compromised. Another and more feasible solution is to allow verification by external parties.

The verification of execution relies on a concept called zero-knowledge proof. A zero-knowledge proof is a cryptographic protocol that can be used by one party, the prover, to prove to another party that some statement is true. The greatest benefit of using this protocol is that the prover can convince the other party of the truthfulness of the statement without actually revealing any additional information [10].

It is possible to create a zero-knowledge proof based on the discrete logarithm problem [10]. In this interactive scheme the prover is able to prove that he or she possesses some secret discrete logarithm of an element in a group and this will be done without actually revealing the secret. Only a transcript, containing communication during the scheme, is generated. In fact, the scheme does not reveal any additional information because any party is able to produce a correct looking transcript on their own, which means that reusing (showing it someone else) an old transcript does not prove knowledge of the secret.

When it comes to mix-nets, in order to ensure correct execution, one can use zero-knowledge proofs to prove that every party shuffles their input ciphertexts and reencrypts these ciphertexts. The shuffling part may be proven to be correct by a zero-knowledge proof that proves that two lists of ciphertexts are permutations of each other [9].

It is also important to consider the possibility that one corrupt mix server may reveal information that should not be revealed, for example revealing the origin of some vote [7]. This is a problem in the El Gamal mix-net described above, since if all parties were using the same public key \( pk \), all parties would also possessed the same secret key.

In order to achieve untraceability, the mix-net servers need to generate different public keys and still remain able to jointly decrypt the output list of ciphertexts. The El Gamal cryptosystem provides protocols for distributed key generation and decryption by several parties [3]. The details of these protocols will be omitted, but the basic principle is the following.

A beforehand specified threshold number of mix-net servers independently generate their own secret keys. The \( l \)th mix-net server generates \( sk_l = x_l \in \mathbb{Z}_q \), from which a partial public key \( pk_l = (g, y_l) = (g, g^{x_l}) \) can be derived. A joint public key is then generated as \( pk = \prod_i y_i \) and the corresponding secret key \( sk = \prod_i x_i \). After each server has processed the ciphertext list, the list is jointly decrypted using a certain procedure. Omitted in this description is the fact that the secret keys are actually shared among all mix-net servers in such a way that no subset of the servers smaller than the threshold number is able to reveal information about the voting before completion of execution. This protocol with a suitably chosen threshold values makes it possible to correctly execute the mix-net even when the mix-net contains dishonest parties [3].
3 Implementation of a Verifier

3.1 Verificatum Mix Network

The Verificatum mix-net is a reencryption mix-net based on the El Gamal cryptosystem [3, p. 1]. During execution it generates a verifiable proof of correctness.

VMN has different types of execution, all of which require different verification. There is a shuffle session, a decryption session and a mixing session, producing different types of proofs. However, the proofs for the mixing session consists of a shuffle and decryption proof. The details on how to verify these different proofs is described in the document [3] by Wikström. We have evaluated this document in order to provide feedback to its author.

3.2 Specification

The document [3] describing a verifier for correct execution of VMN contains detailed instructions for the implementation. A number of subtasks are described, some of which may be implemented independently. The subtasks include implementation of general representation of data, an arithmetic library needed to perform group operations, the cryptographic primitives and the structure of files created during an execution of the mixnet. Furthermore, all algorithms executed during the verification are described in detail in order to allow an independent verifier.

3.3 General Design Choices

The first implementation choice was what programming language to use. We wanted to use well established technologies and have good performance of the verifier. Considering these criteria and our programming skills, the choice was made between C, C++ and Java. Since other groups are working on implementations in Java we decided to use C or C++ and since we wanted to use features such as operator overloading and some OOP, we settled on C++.

The program consists of a few different parts. To represent the various mathematical objects in the calculations, we created a collection of classes called a byte tree. These store group elements and a few other types of data as specified in the documentation.

We wanted to keep the verifier as close to the specification in layout as possible. The byte trees were modelled with a few classes representing nodes and leaves. This enabled us to hide away complex operations behind simple interfaces which resulted in a more readable and maintainable code throughout the rest of the verifier.
3.4 Tools

We developed the program on two different platforms. The code was kept in a git repository and hosted on GitHub. One of the development platforms used OS X 10.8 with Emacs and g++ 4.7. The other used Windows 8 with Visual Studio 2012. We generated reference documents for the program with Doxygen.

3.5 Third Party Libraries

The verifier makes use of a few third party libraries. The actual arithmetic in the byte tree nodes are done with the GMP arithmetic library. This enables us to handle arbitrary large numbers which are used in cryptography. Furthermore, RapidXML is used for parsing the protInfo.xml file at the very beginning of the verifier. The cryptographic hash functions are based on OpenSSL and unit testing of our code was performed by using the Google Test library.

3.5.1 Arithmetic Library

For the actual arithmetics done in the byte tree classes we use the GNU Multi-Precision Library (GMP). On Windows this is replaced by the library Multiple Precision Integers and Rationals (MPIR), a drop in replacement which is easier to compile on Windows systems. GMP was chosen because it is a well known, stable and free library for bignum operations.

3.5.2 XML Parser

RapidXML is used to parse the protInfo.xml file in the beginning of the algorithm. Since this is a very simple operation only performed once, a lightweight and simple library was sought for. RapidXML fulfills these criteria. It consists of a few header files and has an easy to use interface.

3.5.3 Cryptographic Primitives

All of the PRGs and ROs in the verifier are implemented in the verifier but at the core they all depend on cryptographic hash functions such as the SHA-2 family. We take these functions from the OpenSSL library. OpenSSL is a well known, stable and free library for various cryptographic related functions.
3.5.4 Testing

To test the byte tree classes and our cryptographic functions unit testing was done. We used Google Test as our unit test framework because it was simple and easy to use. The unit tests were invaluable in tracking down subtle bugs in the byte tree implementation, especially regarding memory management.

3.6 Math Library - The Byte Tree

To facilitate for the various calculations that needs to be done with various mathematical objects, we created the byte tree classes. The classes, that use the GMP library internally, wrap these operations in a class hierarchy which enables us to create arrays and trees and perform calculations with these compound structures. There are also functions to import strings and files and convert them into these classes. The classes can also be serialized into arrays and strings which are used for testing and debugging purposes.

3.7 Pseudorandom Generators and Random Oracles

The specification also provides details on the PRG:s and RO:s used in Verificatum. These primitives are implemented as classes which are instantiated to perform their respective operations as described earlier. These classes use hash functions from the OpenSSL library internally to perform the hashing part.

3.8 Verifier

To keep track of some data which is persistent throughout the execution, we created a structure to hold this data. This structure was created at the beginning of the execution and then passed around to the different algorithms.

Apart from the byte tree classes and the cryptographic primitives, we needed a few helper functions. The purpose of these functions was to check that some different byte tree adhered to a certain structure.

Each named algorithm in the specification was implemented as its own function. The functions became long but since little code was repeated and the possibilities for isolated testing was minimal, we chose to not split up the algorithms into several functions.

3.9 Testing and Debugging

For the byte tree, PRG and RO classes, a small collection of unit tests were created. The tests for the PRG and RO classes were taken from the test vectors chapter in the specification while we created the test cases for the byte tree classes. The tests were implemented with the Google Test framework.
When all the parts of the program had been implemented we received test data from Wikström to verify that the program worked. The data contained two small instances with all the input files required by a normal run of the verifier. Additionally the test data contained the correct state of key variables throughout important steps of the execution. The debugging was done in Visual Studio. By stepping through the program and comparing states against the test data we removed bugs in the program.

4 Results

4.1 Resulting Code

Our resulting implementation of the mix-net consists of about 5500 lines of C++ code divided into three parts, Arithmetic, Crypto and Verifier. The first two parts are the byte tree classes and the cryptographic primitives respectively. The Verifier part contains the actual verification algorithms.

The code is available on GitHub[11] but should not be considered a stable release.

4.2 Comments on the Documentation

The overall impression of the documentation was that it contained all information needed to implement the verifier. There were, however, errors affecting the execution of the verifier. Some of the problems arising from these errors were overcome after discussion with Wikström. See Appendix B for a list of errors found and specific comments about the report.

In the absolute beginning of the document, the reader is thrown into details about the zero-knowledge proofs used in VMN. A more gentle approach would be to introduce the VMN without assuming too much acquaintance with it or provide the means for the reader to do so on his or her own. It would also be appreciated if the document contained a description of Pedersen commitments.

The background, including a description of a mix-net based on the El Gamal cryptosystem, is clear and concise. After the background, the document contains a list of manageable subtasks in order to facilitate implementation. This list was greatly appreciated as it gave someone unacquainted with VMN ideas of suitable starting points.

Chapters 4 through 6 contain necessary and easily accessible information. However, chapter 4 on byte trees and chapter 6 on representation of arithmetic objects are closely related and could advantageously be presented together while bringing up the cryptographic primitives of chapter 5 afterwards. The part on deriving group elements from random strings depends on chapter 5 and consequently needs to be presented thereafter. See Appendix C for a clarification on these comments.
Chapter 6.6 on Marshalling Groups contains specific details regarding the Verificatum software with strong connections to Java. This chapter could be rephrased to only include information actually needed for implementation of a verifier.

Lastly, the chapter on verification of Fiat-Shamir proofs relies heavily on the derivation of group elements from random strings. By moving chapter 7 to right before the chapter on cryptographic primitives, usage of the document will probably demand less page turns.

5 Conclusion

The use of mix-nets in electronic voting brings with it many advantages but also several risks. It is important that voters feel confident and safe about voting, because otherwise the voting system has lost its purpose. The creation of verifiable mix-nets speaks in favour for electronic voting.

The evaluation and improvement of the VMN verifier document is important in order to allow third parties to independently verify correct execution of the mix-net. By implementing a verifier, several errors could be corrected and suggestions regarding the document’s structure could be made.

The implementation and code creation of the verifer would have been simplified by defining more layers of abstractions in the code structure. Specifically, some classes representing various mathematical objects, such as group elements, would have made the code easier to maintain. We believe that a more solid understanding of the structure of VMN before we started programming would have helped in creating a better structure of the verifier. The choices of third party libraries were good and they were all easy to use in our project. This also makes the amount of code which need to be written smaller. The lack of test data greatly increased the difficulty of implementing the verifier.

The specification document for the verifier does include all the information required to write the verifier. However, the structure could be improved for greater readability. There is also some unnecessary information in the document. Lastly, the document could benefit from an improved description of the VMN so that one easier can acquire a better understanding of what approach to take when implementing the verifier.
References

Handbook of Applied Cryptography. ISBN: 0-8493-8523-7

Random Oracles are Practical: A Paradigm for Designing Efficient Protocols

How to Implement a Stand-alone Verifier for the Verificatum Mix-Net


Universal Re-encryption for Mixnets
Topics in Cryptology – CT-RSA 2004

[6] A Survey on Mix Networks and Their Secure Applications
Krishna Sampigethaya and Radha Poovendran

[7] Lecture 17: Introduction to Electronic Voting
Lecture notes by Ben Adida
http://courses.csail.mit.edu/6.897/spring04/L17.pdf

[8] Lecture 18: Mix-net Voting Systems
Lecture notes by Yael Tauman Kalai
http://courses.csail.mit.edu/6.897/spring04/L18.pdf

[9] Proofs of Restricted Shuffles
BJörn Terelius and Douglas Wikström
http://www.nada.kth.se/ dog/research/TW10Conf.pdf

[10] Σ-Protocols Continued & Introduction to Zero Knowledge
Lecture notes by Joël Alwen
http://cs.nyu.edu/courses/spring07/G22.3220-001/lec3.pdf

Erik Larsson and Carl Svensson https://github.com/ZetaTwo/sa104x-kexjobb
6 Appendix A - Group Theory

A group \((G, \cdot)\) is a set \(G\) and a binary operation \(\cdot : G \rightarrow G\), called multiplication, that satisfies the four group axioms:

1. The product of any two elements of \(G\) is in \(G\)
   \[
   \forall a, b \in G, \ a \cdot b \in G
   \]

2. Multiplication is associative
   \[
   \forall a, b, c \in G, \ (a \cdot b) \cdot c = a \cdot (b \cdot c)
   \]

3. There exists a unique identity element in \(G\)
   \[
   \exists! e \in G \text{ such that } \forall a \in G, \ a \cdot e = e \cdot a = a
   \]

4. Every element in \(G\) has an inverse
   \[
   \forall a \in G, \ \exists b \in G \text{ such that } a \cdot b = b \cdot a = e
   \]

When the operation of the group \((G, \cdot)\) is understood from the context, one often abbreviates the notation and calls \((G, \cdot)\) simply \(G\).

Oftentimes one wants to multiply an element to itself a number of times. This is called exponentation. Let \(g \in G\) and \(x \in \mathbb{Z}\), then \(g\) multiplied to itself \(x\) times

\[
g \cdot \ldots \cdot g = g^x
\]

is called \(g\) raised to the power of \(x\). The usual laws of exponentation holds in a group.

The number of elements in a group \(G\) is called the order of the group. If the order of \(G\) is finite, \(G\) is said to be finite. If \(x \in \mathbb{Z}\) is the smallest integer such that \(g^x = e\), then \(x\) is called the order of the element \(g \in G\). Notice that if \(y, y' < x, y \neq y'\), then \(g^y \neq g^{y'}\) as otherwise

\[
g^y = g^{y'} \Rightarrow e = g^{y} \cdot g^{-y} = g^{y'-y}
\]

where \(0 < y' - y < x\) and hence \(x\) is not the order of \(g\), which is a contradiction.

If the order of \(G\) is \(q < \infty\) and there exists an element \(g \in G\) such that the order of \(g\) is \(q\), then \(G\) is said to be cyclic with \(g\) as a generator. The motivation behind this naming, is that any element \(h \in G\) is equal to \(g^x\) for some \(x \leq q\), so \(g\) generates the whole group.
### 7 Appendix B - Errors and Comments

<table>
<thead>
<tr>
<th>Error/Comment</th>
<th>Correction</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of algorithm is Protocol 19</td>
<td>Change to <em>Algorithm 19</em></td>
<td>Algorithm 19, p.15</td>
</tr>
<tr>
<td>Random vector $h$ not in argument list</td>
<td>Add $h$ to argument list</td>
<td>Argument 19, p.15</td>
</tr>
<tr>
<td>$\tau_i^{dec}$ incorrect symbol</td>
<td>Replace with $\sigma_i^{dec}$</td>
<td>Files in main directory 22, p.21</td>
</tr>
<tr>
<td>Private keys $x_l$ not in argument list</td>
<td>Add $x_l$ to argument list</td>
<td>Algorithm 26, p.25</td>
</tr>
<tr>
<td>$PDec_{x_l}(L)$</td>
<td>$PDec_{x_l}(L_l)$</td>
<td>Algorithm 26, p.25</td>
</tr>
<tr>
<td>Creation of prefix $\rho$ incorrect</td>
<td>TBA</td>
<td>Algorithm 27, p.26</td>
</tr>
<tr>
<td>Reading of $\mu$ not described</td>
<td>Introduce $\mu$ properly</td>
<td>Algorithm 24, p.23</td>
</tr>
<tr>
<td>$N'$ not defined</td>
<td>Define $N'$ in chapter 6.7</td>
<td>Chapter 6.7, p.14</td>
</tr>
</tbody>
</table>

Table 1: Errors and comments on documentation describing a Verificatum verifier.
8 Appendix C - Suggestion for Revised Document Structure

1. Introduction
2. Background
3. How to Write a Verifier
4. Byte Trees
5. Representations of Arithmetic Objects
   - Basic Objects
   - Prime Order Fields and Product Rings
   - Multiplicative Groups Modulo Primes
   - Standard Elliptic Curves over Prime Order Fields
   - Arrays of Group Elements and Product Groups
6. Protocol Info Files
7. Cryptographic Primitives
8. Groups
   - Marshalling Groups
   - Deriving Group Elements from Random Strings
9. Verifying Fiat-Shamir Proofs
10. Verification
11. Standard Command Line Interface of Verifier
12. Additional Verifications Needed in Applications
13. Appendices