Implementation and evaluation of a group encryption scheme

Bachelor Thesis of

Peter Ten

At the Department of Computer Science
Chair for Computer Science II
Software Engineering
Secure Software Systems

Reviewer: Prof. Dr.-Ing. Alexandra Dmitrienko
Advisor: M. Sc. Christoph Hagen
I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 23. December 2019

(Peter Ten)
# Contents

## 1. Introduction

## 2. Fundamentals

- 2.1. Group Definition ........................................ 3
- 2.2. Group Encryption Advantages ............................ 3
- 2.3. Group-oriented Encryption ................................. 4
- 2.4. Proposed Scheme ......................................... 4
- 2.5. Mathematical Basics ..................................... 5
  - 2.5.1. Cyclic Groups ......................................... 5
  - 2.5.2. Discrete Logarithm Problem .......................... 5
  - 2.5.3. Elliptic Curve Cryptosystems ......................... 5
  - 2.5.4. Bilinear Maps ........................................ 7

## 3. Nishat’s Scheme Principle

- 3.1. Security .................................................... 9
  - 3.1.1. Computational Diffie-Hellman (CDH) Assumption .... 9
  - 3.1.2. Decisional Diffie-Hellman (DDH) Assumption ....... 9
  - 3.1.3. V-decisional Diffie-Hellman (V-DDH) Assumption .. 9
  - 3.1.4. Security Notion ...................................... 10
- 3.2. The Functionality Of The Scheme ......................... 10
  - 3.2.1. Group Setup .......................................... 10
  - 3.2.2. Key Generation ....................................... 10
  - 3.2.3. Encryption .......................................... 11
  - 3.2.4. Decryption .......................................... 11
- 3.3. Correctness Of Decryption ............................... 12
- 3.4. Group Changes .......................................... 13
  - 3.4.1. Member Addition ..................................... 13
  - 3.4.2. Member Removal ..................................... 13

## 4. Encountered Problems ........................................ 15

- 4.1. Typing Error In Formula .................................. 15
- 4.2. Public Parameter ......................................... 16

## 5. Implementation Details ..................................... 19

- 5.1. Functionality Of Encryption .............................. 19
- 5.2. Requirements For The API ................................ 19
- 5.3. Limitations ............................................... 20
- 5.4. Tools And Libraries ...................................... 20
5.5. Structure .......................................................... 20
  5.5.1. Data Structure ............................................. 20
  5.5.2. User Management ......................................... 22
5.6. Implementation ............................................... 22
  5.6.1. Methods ..................................................... 23
    5.6.1.1. Group Setup And Key Generation .................. 24
    5.6.1.2. Group Change ....................................... 25
    5.6.1.3. Encryption ......................................... 26
    5.6.1.4. Decryption ......................................... 26

6. Evaluation ....................................................... 27
  6.1. Correctness ................................................ 27
  6.2. Test Environment ......................................... 28
  6.3. Storage Cost ............................................... 28
  6.4. Computation Cost ......................................... 28
    6.4.1. Encryption ............................................ 30
    6.4.2. Decryption ............................................ 30
    6.4.3. Key Generation ........................................ 31
      6.4.3.1. Public Key ...................................... 31
      6.4.3.2. Secret Key ...................................... 32
    6.4.4. Group Changes ........................................ 33
    6.4.5. Group Creation ........................................ 36

7. Conclusion ...................................................... 37

Appendix .......................................................... 39
  A. Lookup Tables ................................................ 39

Bibliography ....................................................... 45
Abstract

Most today’s group-oriented encryption schemes have a so called “one-affects-all-problem”. It means that group changes result in overhead on all members of the group. Therefore, updating the public key has an effect on everyone in the group. It is desirable to avoid that and to have constant public and private key lengths.

A new dynamic scheme was introduced by Koti Nishat and B. R. Purushothama in 2016. This innovative scheme allows every user to keep their private keys. When the group changes, only a new public key and public parameter have to be recomputed. Adding and removing new group members should not cause much overhead and should resolve the one-affects-all-problem. The scheme also preserves forward- and backward-secrecy and has constant key and ciphertext sizes.

This thesis is about explaining the mathematical basics and Nishat’s scheme and introducing an implementation of the scheme. Some mathematical issues occurred, of which one led to a lower performance. These issues will be discussed and corrected. The adapted implementation will be compared in terms of computation time. Encryption, decryption, key generation and group changes will be measured for comparison with the author’s measurements and checked if the scheme is still beneficial.

Deutsche Zusammenfassung:


1. Introduction

Cryptography makes our world much easier. It allows us to send messages to friends securely. It is possible to surf in the Internet without being threatened, that someone is reading along our traffic. Online banking, home office and anonymity are possible because of cryptography. The list of advantages is long and can be filled with many more examples. People began to think early about ways of hiding messages from unintended recipients. It all started back then in the ancient Egyptian time, about 3000 before Christ. At that time altered hieroglyphs were used to encrypt messages. Until modern cryptography with computers as we know them today, encryption was always symmetric. This means that a single key is used for encryption and decryption of messages. Some well known examples of historic ciphers are the “Caesar cipher”, “Vigenère cipher” or the “Enigma”. Today for example AES is used.

Asymmetric encryption, like RSA, allows to separate keys into a public and private key. Messages can be encrypted with the public key and ciphertexts can be decrypted with the private key.

Today’s encryption schemes between two parties are well studied. But humans are living in social groups by nature. People would like to communicate safely with different groups of people e.g. friends, family, colleagues or a study group etc. But there are also highly confidential use cases for groups. For example, a hospital is interested in sharing patient data with different groups of employees. These groups should only be able to read the data intended for them.

Therefore, there is need for cryptographic protocols for groups. These protocols should also be able to handle the addition or removal of members. Most of today’s protocols are static and have a so called “one-affects-all-problem”, because all keys have to be recomputed or other big overhead is caused, when the member constellation changes.

Koti Nishat and B. R. Purushothama tried to solve this problem with an innovative encryption-scheme and published their article “Group-oriented encryption for dynamic groups with constant rekeying cost” in 2016. This scheme allows every user to keep their secret key as long as the group exists. Public and private key and ciphertext sizes are constant.
This thesis is about explaining group-oriented encryption and their mathematical basics in order to understand the scheme and introduces an implementation, that is written as an API, which was evaluated in terms of correctness and performance. During the process of programming, mathematical problems were encountered, of which one led to a lower performance.

Chapter two will discuss the mathematical basics. Chapter three is about Nishat’s scheme and explains the functionality. Chapter four deals with the occurred problems, which will be corrected. Chapter five introduces the implementation and gives an overview over the used tools, structure and functionality. Chapter six evaluates the correctness and performance.
2. Fundamentals

This chapter covers the basics of group encryption and mathematical basics in order to understand Nishat’s scheme and the implementation, that is based on elliptic curves as (mathematical) groups.

2.1. Group Definition

A group is defined as a set of members. One or more members are so called group administrators, who are creating and managing the group. Every member is authorized to decrypt encrypted messages, which are intended for the group.

2.2. Group Encryption Advantages

There are various use cases for groups, when it comes to sharing information with a specific set of users. A simple example is a messenger, that allows to create a group. The members then can exchange messages between each other.

A naive approach for group encryption is to simply use classic end-to-end encryption. A group is defined as a specific subset of users. Every user owns an individual public and secret key. A message is encrypted with every individual public key and sent to every user individually, who decrypts the message with the secret key. Group changes do not cause much overhead, since only the defined subset of members must be updated. However, every user needs an own set of public and secret key. Additionally, messages must be encrypted as many times as there are members in the group. With bigger groups, the overhead is getting quite large, because key management gets complicated and high traffic is caused [7].

Therefore, there is need for specific group encryption schemes. There are several desirable requirements:

- A group is associated with a single public key for encryption
- Every user has an individual secret key for decryption
- Creating a group costs linear computation time
• Secret keys do not change over the lifetime of a group
• Changing a group (adding or removing a user) has constant computation time
• Public key and secret key sizes are constant
• The ciphertext length is constant

Not all requirements have to be fulfilled to be considered as an advantage over the naive approach. It strongly depends on the use case and requirements for the groups, but the more properties are fulfilled, the better a scheme is. Depending on requirements in practicability, group size, security level, key management etc., different schemes are thinkable.

2.3. Group-oriented Encryption

A group-oriented encryption scheme is based on public-key cryptography. The group is associated with a public key. Every member has an individual private key. The already mentioned “one-affects-all-problem” describes a high overhead for group changes, therefore, changes of the public key [9]. There are different approaches to realize such a scheme. The mentioned ones in the following are just shortly described, because the focus is on the proposed scheme by Nishat and Purushothama, which is an group-oriented encryption.

Threshold Cryptosystem: A group has a dedicated public key, which allows to encrypt messages. The threshold value defines the minimum amount of users, who have to collaborate to perform a decryption. [3][6].

Identity-based Cryptosystem: A group is defined as \((t, n)\)-group. The variable \(t\) specifies how many users have to perform a decryption together. \(n\) is the amount of members [12].

2.4. Proposed Scheme

Most of the current approaches have the problem, that they cannot deal efficiently with group changes. They are not capable of adding or removing a member, without affecting the remaining members and causing overhead. Nishat’s scheme tries to solve the one-affects-all-problem with addition of a public parameter. This parameter allows every member to keep their private key as long as the group exists. The scheme also provides constant key and ciphertext sizes and preserves forward and backward secrecy. Forward secrecy means that a removed member is not able to read messages in the future. Backward secrecy means that a new member is not able to read messages from the past. The scheme claims that all operations are computable in constant time [9]. However, a mathematical mistake does not allow constant computation time for group changes. This problem will be discussed later in chapter [4]
2.5. Mathematical Basics

This section covers the needed mathematical basics for understanding the encryption scheme and implementation. This section is referring to [10].

2.5.1. Cyclic Groups

First (mathematical) groups are defined.

**Definition 2.5.1.** A group \((G, \circ)\) combines two elements of a given set \(G\) with an operation \(\circ : G \times G \rightarrow G\); \((a, b) \mapsto a \circ b\) with \(a, b \in G\), with following properties:

1. The group operation is *closed*, so no operation \(\circ\) leads to an element, which is not in the set \(G\)
2. The group operation is *associative*, so \(a \circ (b \circ c) = (a \circ b) \circ c, \forall a, b, c \in G\)
3. An *identity element* in \(G\) (denoted as 1) exists, so \(a \circ 1 = a, \forall a \in G\)
4. Every element \(a \in G\) has an *inverse* (denoted as \(a^{-1}\)), so \(a \circ a^{-1} = 1, \forall a, a^{-1} \in G\).
5. A group \(G\) is *abelian* if \(a \circ b = b \circ a, \forall a, b \in G\)

A group is finite, if it has a finite amount of elements. The order of a group \(G\) is denoted as \(|G|\). It is the amount of elements in the set \(G\). The order \(ord(a)\) of an element \(a \in G\) is the smallest positive integer, such that \(a^k = a \circ a \circ \ldots \circ a = 1\).

**Definition 2.5.2.** A cyclic group \((G, \circ)\) is a group, that contains an element \(\alpha\) with \(ord(\alpha) = |G|\). Such an element with *maximum order* is a *generator*. Every element of \(G\) can be calculated with an exponent of \(\alpha\).

A cyclic finite group ensures that the numbers are closed when doing calculations. This allows constant memory size and more efficient calculations.

2.5.2. Discrete Logarithm Problem

The discrete logarithm problem describes a problem, which is computational difficult to solve efficiently. Let \(G\) be a finite cyclic group with order \(n\). Let \(\alpha \in G\) be a generator. Let \(\beta \in G\) be another element. The problem describes an integer \(x\) with \(1 \leq x \leq n\), so that \(\beta = \alpha^x\). Hence, it is hard to find a \(x\) to fulfill the equation, when \(\alpha\) and \(\beta\) is given.

2.5.3. Elliptic Curve Cryptosystems

Elliptic curves are based on the discrete logarithm problem and can satisfy the properties of a finite cyclic group. Elliptic Curve Cryptosystems (ECC) can provide the same level of security as RSA or discrete logarithm systems. However, the size of operands are significantly shorter (160-256 bit vs 1024-3074 bit). The security level of \(n\) bit is the best known attack, which requires \(2^n\) steps.
Elliptic Curves

Elliptic curves are a polynomial equation. The set of points \((x, y)\) are forming the curve.

**Definition 2.5.3.** An elliptic curve over \(\mathbb{Z}_p\), \(p > 3\), is the set of all pairs \((x, y) \in \mathbb{Z}_p\), which fulfill the equation

\[
y^2 \equiv x^3 + a \cdot x + b \mod p,
\]

together with an imaginary point of infinity \(\vartheta\) with \(a, b \in \mathbb{Z}_p\) and

\[
4 \cdot a^3 + 27 \cdot b^2 \neq 0 \mod p.
\]

All group operations are possible through constructing operations on the curve, which fulfill the group requirements. To keep it simple these specific operations will not be discussed in this thesis.

![Example of the elliptic curve](image)
2.5.4. Bilinear Maps

The following section is referring to [1] and [2].

Definition 2.5.4. Let \( p \) be a prime. Let \( G_1 \) and \( G_2 \) be cyclic groups of order \( p \). Let \( g \) be a generator of \( G_1 \). A bilinear map is defined as \( e : G_1 \times G_1 \rightarrow G_2 \) with the following properties:

1. Bilinearity: \( e(g^l, g^k) = e(g, g)^{lk} \forall g \in G_1, l, k \in \mathbb{Z}_p^* \)
2. Non-degeneracy: \( e(g, g) \neq 1 \)
3. Symmetry: \( e(g^l, g^k) = e(g^k, g^l) = e(g, g)^{lk} \forall g \in G_1, l, k \in \mathbb{Z}_p^* \)
4. Computability: For computing the bilinear map, an efficient algorithm exists

Bilinear maps are also called *pairings*. The defined pairing is symmetric, because of \( G_1 \times G_1 \rightarrow G_2 \). An asymmetric pairing would be \( G_1 \times G_2 \rightarrow G_t \) with \( G_t \) being a cyclic group with the same order as \( G_1 \) and \( G_2 \).
3. Nishat’s Scheme Principle

The scheme was invented to solve the one-affects-all-problem. Therefore, allowing a group to be dynamic in an efficient way. The size of secret keys, public key and ciphertexts remain constant. To achieve this the keys are split into sub-keys. The group administrator also computes a public parameter, which is used by every user to compute a secret number, that is part of the public key. In chapter 4, a mistake in the calculation of the public parameter and a typing error in one of the decryption formulas will be discussed and corrected. The following chapter is referring to [9].

3.1. Security

3.1.1. Computational Diffie-Hellman (CDH) Assumption

For given \(g, g^x\) and \(g^y\) with \(x, y \in \mathbb{Z}_p^*\), the CDH assumption states that computing \(g^{xy} \in \mathbb{G}_1\) is a computationally hard problem in \(\mathbb{G}_1\). There does not exist a probabilistic polynomial time algorithm \(A\) with non-negligible advantage \(\epsilon\), such that \(\Pr[A(g, g^x, g^y) = g^{xy}] \geq \epsilon\).

3.1.2. Decisional Diffie-Hellman (DDH) Assumption

Given the distributions \((g, g^x, g^y, g^{xy})\) and \((g, g^x, g^y, Q)\), where \(x, y, Q \in \mathbb{Z}_p^*\) are chosen randomly. The DDH assumption states that two distributions are computationally indistinguishable, so there does not exist a probabilistic polynomial time algorithm \(A\) with non-negligible advantage \(\epsilon\), such that \(|\Pr[A(g, g^x, g^y, g^{xy}) = 0] - \Pr[A(g, g^x, g^y, Q) = 0]| \geq \epsilon\).

3.1.3. V-decisional Diffie–Hellman (V-DDH) Assumption

Given the distributions \((g, g^x, g^{xy}, g^{xz}, g^{yz})\) and \((g, g^x, g^{xy}, g^{xz}, Q)\), where \(x, y, z \in \mathbb{Z}_p^*\) and \(Q \in \mathbb{G}_1\) are chosen randomly. The V-DDH assumption states that the two distributions are computationally indistinguishable, so there does not exist a probabilistic polynomial time algorithm \(A\) with non-negligible advantage \(\epsilon\), such that \(|\Pr[A(g, g^x, g^{xy}, g^{xz}, g^{yz}) = 0] - \Pr[A(g, g^x, g^{xy}, g^{xz}, Q) = 0]| \geq \epsilon\).
3.1.4. Security Notion

The security is defined by a game between \( \mathcal{A} \) (adversary) and \( \mathcal{C} \) (challenger). The game is defined as follows:

1. **Setup**: The system is initialized by \( \mathcal{C} \) and \( \mathcal{A} \) gets the resulting public parameters for the group \( G \). \( \mathcal{C} \) does not disclose the secret key.

2. **Challenge**: Two messages \( M_0, M_1 \) (of equal length) are randomly chosen by \( \mathcal{A} \) and given to \( \mathcal{C} \). A random bit \( b \in \{0, 1\} \) is chosen by \( \mathcal{C} \) and encryption is performed to the message \( M_b \), which produces the challenge ciphertext \( C^*_G \).

3. **Guess**: \( \mathcal{A} \) guesses \( b' \in \{0, 1\} \). If \( b = b' \), \( \mathcal{A} \) wins the game.

The encryption scheme is secure if \( \mathcal{A} \)'s advantage in winning the game, given by
\[
\epsilon = \left| \Pr(b = b') - \frac{1}{2} \right|
\]
is negligible.

The proposed scheme provides security against chosen plaintext attacks under the \textit{V-DDH} assumption.

3.2. The Functionality Of The Scheme

A group consists of a group administrator/creator (GC) and \( n \) users \( u_1, \ldots, u_n \). Every group is associated with a public key. The users have their own individual private key for decrypting messages, that are intended for the group. Let \( A \) be the name of a group. The scheme consists of four major algorithms: group setup, key generation, encryption and decryption.

3.2.1. Group Setup

Input to this algorithm is a security parameter \( \lambda \), which defines the security level. At first two cyclic groups, \( G_1 \) and \( G_2 \), of order \( p \), where \( p \) is prime, are generated. Let \( g \in G_1 \) be a generator of \( G_1 \). An efficiently computable bilinear map is defined as \( e : G_1 \times G_1 \to G_2 \). The administrator randomly chooses \( \alpha, \beta \in \mathbb{Z}_p^* \) and \( h \in G_1 \).

The output of the algorithm is the master secret key \( MSK = (\alpha, \beta) \) and \( \Gamma = (h, g, g^\alpha, g^\beta) \) as the system parameters.

3.2.2. Key Generation

After the group is initialized the administrator can compute the user’s private keys
\[
SK_i = (s_{i_1}, s_{i_2}, s_{i_3}, s_{i_4}, s_{i_5}, s_{i_6}, s_{i_7}),
\]
public key and public parameter. Input to this algorithm is \( \Gamma \), the group ID and the associated users. Output are the keys and public parameter \( \gamma \). The administrator randomly chooses \( r_i \in \mathbb{Z}_p^* \), such that \( r_i \) is prime, for every user \( u_i \). The seven sub-keys are calculated as follows:

1. \( s_{i_1} = h^{r_i} \)
2. \( s_{i_2} = g^{r_i} \)
3. \( s_{i_3} = s_{i_1}^{\beta^{-1}} = h^{r_i\beta^{-1}} \)
3.2. The Functionality Of The Scheme

4. \( s_{i4} = g^{\alpha r_i \beta^{-1}} \)

5. \( s_{i5} = h^{\beta^{-1}} \)

6. \( s_{i6} = g \cdot h^{-i} \)

7. \( s_{i7} = r_i \)

The administrator computes the public key \( PK_A = (PK_{A1}, PK_{A2}) \) and public parameter for group \( A \) as follows:

1. Chooses random \( k \in \mathbb{Z}_p^* \), such that \( k < r_i \ \forall i \in 1, ..., n \)
2. Chooses random \( a \in \mathbb{Z}_p^* \)
3. Computes the public parameter \( \gamma = (a \times r_1 \times ... \times r_n) - k \)
4. Computes \( PK_{A1} = g^{ak} \) and \( PK_{A2} = g^{3k} \)

### 3.2.3. Encryption

The input for the encryption algorithm is the message \( M \in \{0, 1\}^\lambda \) and the group’s public key. The output is the ciphertext intended for the users. A message for group \( A \) is encrypted by a sender as follows:

1. Chooses random \( t \in \mathbb{Z}_p^* \)
2. Calculates \( c_1 = e(g, PK_{A1})^t \cdot M \)
3. Calculates \( c_2 = (h \cdot PK_{A1})^t \)
4. Calculates \( c_3 = (PK_{A2})^t \)

The ciphertext \( C_A = (c_1, c_2, c_3) \) is sent to the group.

### 3.2.4. Decryption

The input for the decryption algorithm is the ciphertext \( C_A \), the secret key \( SK_i \) of user \( u_i \) and the public parameter \( \gamma \) of group \( A \). The output is the message \( M \).

A user \( u_i \) of group \( A \) is decrypting the ciphertext as follows:

1. Calculates \( \kappa = r_i - \gamma \ \text{mod} \ r_i = k \)
2. Calculates \( \zeta_1 = s_{i1} \cdot (s_{i2})^\kappa \)
3. Calculates \( \zeta_2 = \frac{s_{i3} \cdot (s_{i4})^\kappa}{(s_{i5})^{\kappa^{-1}}} \)
4. Calculates \( \tau = \frac{e(s_{i6}, c_2) \cdot e(\zeta_2, c_3)}{e(\zeta_1, c_2)} \)
5. Calculates \( M = \frac{c_1}{\tau} \)
3.3. Correctness Of Decryption

Let \( u_i \) be a user of a group and \( C \) a ciphertext. \( u_i \) is performing a decryption with its secret key \( SK_i \). \( \kappa \) is calculated as follows:

\[
\begin{align*}
\kappa &= r_i - \gamma \mod r_i \\
&= r_i - ((a \times r_1 \times \ldots \times r_n) - k) \mod r_i \\
&= r_i - ((a \times r_1 \times \ldots \times r_n) \mod r_i - (k \mod r_i)) \\
&= r_i - (0 - k) \mod r_i \\
&= r_i - (r_i - k) \\
&= k
\end{align*}
\]

The reason why \( k \) has to be smaller than every \( r_i \ \forall i \in \{1, \ldots, n\} \) is because the result of \( (k \mod r_i) \) would not necessarily be \( k \).

Rearranging with secret sub-keys:

\[
\begin{align*}
\zeta_1 &= s_{i_1} \cdot (s_{i_2})^\kappa \\
&= h^{r_i} \cdot (g^{r_i})^k \\
&= h^{r_i} \cdot g^{kr_i}
\end{align*}
\]

\[
\begin{align*}
\zeta_2 &= s_{i_3} \cdot (s_{i_4})^\kappa \\
&= \frac{(s_{i_5})^{\kappa-1}}{(s_{i_5})^{\kappa-1}} \\
&= \frac{h^{r_i \beta^{-1}} \cdot (g^{\alpha r_i \beta^{-1}})^k}{(h^{\beta^{-1}})^{k-1}} \\
&= h^{(r_i - k^{-1}) \beta^{-1}} \cdot g^{\alpha kr_i \beta^{-1}}
\end{align*}
\]

Using \( \zeta_1 \) and \( \zeta_2 \) and the ciphertext components \( c_2 \) and \( c_3 \) to compute \( \tau \):

\[
\begin{align*}
\tau &= \frac{e(s_{i_6}, c_2) \cdot e(\zeta_2, c_3)}{e(\zeta_1, c_2)} \\
&= e(g, PK_{A_1})^t
\end{align*}
\]

The user receives the message by using \( c_1 \) and \( \tau \):

\[
\begin{align*}
M &= \frac{c_1}{\tau} \\
&= \frac{e(g, PK_{A_1})^t \cdot M}{e(g, PK_{A_1})^t} \\
&= M
\end{align*}
\]
3.4. Group Changes

The proposed advantage of the scheme takes effect when the group changes in case of addition or removal of a user. Every member can keep their secret key. They only need to update the public key and public parameter for encryption and decryption.

3.4.1. Member Addition

The group administrator is able to add a user to the group. The new user’s private key is generated like shown in section 3.2.2. The public key and public parameter have to be recomputed to preserve backward secrecy. Let the new user, who is not part of the group \( A \) yet, be \( u_{n+1} \). We denote the current members as the set \( \{ u_1, ..., u_n \} \). The current public key is \( PK_A = (g^{a_k}, g^{\beta_k}) \). The administrator adds the new user to group \( A \) as follows:

1. Chooses a random prime \( r_{n+1} \in \mathbb{Z}_p^* \)
2. Calculates the secret key of \( u_{n+1} \)
3. Chooses a new \( a' \in \mathbb{Z}_p^* \)
4. Chooses a new \( k' \in \mathbb{Z}_p^* \), where \( k' < r_i \forall i \in \{1, ..., n+1\} \)
5. Calculates a new public parameter \( \gamma' = (\gamma + k) \times a^{-1} \times a' \times r_{n+1} - k' \)
6. Calculates the public key components \( PK'_{A_1} = g^{a_k} \) and \( PK'_{A_2} = g^{\beta_k} \)
7. Sets the public key \( PK_A = (PK'_{A_1}, PK'_{A_2}) \) and \( \gamma = \gamma' \)

All users can still decrypt in the usual way, like described in section 3.2.4.

3.4.2. Member Removal

The removal of a group member is very similar to the addition. The group administrator is performing the same steps like shown in the previous section 3.4.1 except that a group member \( u_x \) is removed and the calculation of the public parameter is different. \( \gamma' \) is calculated instead as follows:

\[
\gamma' = (\gamma + k) \times a^{-1} \times a' \times r_x^{-1} - k'.
\]

Forward secrecy is also preserved, because of the new public key and public parameter.

All users can still decrypt in the usual way like described in section 3.2.4.
4. Encountered Problems

There are some problems and misconceptions in the article. Sometimes the format and denotations are a bit unusual (e.g. multiplication points looked like a comma or random elements were denoted as $\in R$ without any comment), but those are just small details that get clear when continuing with reading. This chapter discusses an important typing error and a crucial mathematical mistake (unfortunately, there was no reaction from the authors to emails regarding the problems).

4.1. Typing Error In Formula

The reason why this typing error is important to mention is because it completely changes an equation during the decryption. The scheme, respectively decryption, does not work correctly without correcting this mistake. The first appearance of the wrong equation in the article is in section 4.4 Decryption, when $\zeta_2$ is computed. In section 4.4.1 the authors proved the correctness of $\zeta_2$ with equation (3).

The original formula is:

$$\zeta_2 = \frac{s_i \cdot (s_i^i)_{\kappa-1}}{(s_i^i)_{\kappa-1}}$$

The corrected formula is:

$$\zeta_2 = \frac{s_i \cdot (s_i^i)^\kappa}{(s_i^i)^{\kappa-1}}$$
For the correction the formula is reversed and the private sub-keys $s_i^3 = h^{r_i \beta^{-1}}$, $s_i^4 = g^{\alpha r_i \beta^{-1}}$ and $s_i^5 = h^{\beta^{-1}}$ are used:

$$\zeta_2 = h^{(r_i - k^{-1}) \beta^{-1}} \cdot g^{\alpha \cdot k \cdot r_i \cdot \beta^{-1}}$$

$$= h^{((r_i - k^{-1}) \beta^{-1})} \cdot (s_i^4)^k$$

$$= (si_3)^k \cdot (s_i^4)^k$$

The typing error was repeated in all calculations of $\zeta_2$.

### 4.2. Public Parameter

The public parameter $\gamma$ is the crucial element of Nishat’s scheme. As explained in chapter 3, the parameter allows every group member to calculate the secret number $k$. The article describes the calculation of the public parameter with group operations. All elements used in the calculation of $\gamma$ are elements of the cyclic group $\mathbb{Z}_p^*$. When adding or removing a user, the inverses $a^{-1}$ and $r_x^{-1}$ are used. These do not exist, because it would violate the closure property of $\mathbb{Z}_p^*$. The counterexample below makes clear that the inverse of $a$ has to be $\frac{1}{a}$, which is not an integer.

The article shows the correctness of the calculation of $\kappa$ by a user in section 4.4.1. These calculations can be used in following counterexample, that shows that group operations are not valid.

Let $\mathbb{Z}_p^*$ be $\mathbb{Z}_{13}^*$. There are two members in the group and the administrator chooses the following elements $a = 3$, $r_1 = 5$, $r_2 = 7$ and $k = 2$, like shown in the scheme.

Now $\gamma$ is calculated with group operations:

$$\gamma = (a \times r_1 \times r_2) - k$$

$$= (((3 \cdot 5 \cdot 7) \mod 13) - 2) \mod 13$$

$$= ((105 \mod 13) - 2) \mod 13$$

$$= (1 - 2) \mod 13$$

$$= 12$$

Now we try to calculate $\kappa$ with $r_1$ and group operations:

$$\kappa = (5 - 12 \mod 5) \mod 13$$

$$= 5 \mod 13 - 2 \mod 13$$

$$= (5 - 2) \mod 13$$

$$= 3 \neq k = 2$$

This small example proves that the information $k$ gets lost, when calculating with group operations. We can still use these equations, when all calculations, involving the public parameter, are done instead in a field as “normal” multiplications. Thus, the inverse of $a$ is $\frac{1}{a}$.
Therefore, after group changes no complete new public parameter has to be calculated but instead can be updated like shown in sections 3.4.1 and 3.4.2. This still leads to the big disadvantage that the size of the parameter is dependent on the group size. The larger the group gets, the larger the parameter gets. For security reasons the $7^{th}$ key part $s_{i7}$ has to be a large number. Hence, the parameter gets extremely large with a big group size.

The, by the authors, proposed constant computation time, would only be possible with a closed public parameter. Group operations would make sure, that the size remains constant and, therefore, updating the public parameter would be possible in constant time.
5. Implementation Details

The implementation is intended for evaluating correctness and performance of the scheme. It is written in the programming language “C (C99 conformity)” and structured as an API.
A full documentation is available and included in the API files.

5.1. Functionality Of Encryption

The messages that are sent to the group are 128-bit AES keys. Therefore, the actual message is encrypted with AES and only the encrypted key is sent to the members. In order to decrypt the actual message, group members decrypt the encrypted AES key and use it to obtain the message.

5.2. Requirements For The API

The implementation is able to perform all algorithms described in the scheme, therefore group setup, key generation, encryption, decryption, member addition and member removal.
It is possible to create multiple groups.

The API was tested on Windows 10 64-bit, but should be platform independent.
5.3. Limitations

The current implementation is only for testing purposes to measure performance. It does not provide:

- Secure computations against e.g. side-channel attacks
- Boundary or format checks. It is assumed that all inputs are correct
- No cryptographic pseudo random generator
- Ability to store information securely
- Ability to send information respectively run a group on different devices

In order to actually use the API for cryptographic use cases, secure computation have to be ensured. This also involves a secure pseudo random generator, which may be chosen by the user. When actually creating a group, the group’s public values, private keys and ciphertexts, have to be distributed among the participating users. This can be achieved with e.g. “Google protocol buffers”, which allows to serialize a struct for sending it over a network.

5.4. Tools And Libraries

For implementing the scheme, “Microsoft Visual Studio Community 2019” was used. Included libraries are “MPIR” (Multiple Precision Integers and Rationals) and “PBC” (Pairing-Based Cryptography) \[8\][11]. There are only a few libraries, which implemented pairing-based cryptography. One of them is PBC, that offers a good range of algorithms and is based on elliptic curves. It is also well documented. PBC is depended on the mathematical library “GMP” (The GNU Multiple Precision Arithmetic Library), which is a performant library \[4\]. MPIR is forked from GMP and offers the possibility to include MPIR in a Visual Studio project.

5.5. Structure

The structure of the API is very similar to the structure in the article. The algorithms are directly translated into C. Structs were used to separate the different group elements like administrator, users, public values and ciphertexts.

The users are saved in a linked list and the administrator manages the list. Further information regarding the user management is described in section 5.5.2.

The API is usable with help of methods, that start with “dge_” ([d]ynamic [g]roup-oriented [e]ncryption).

5.5.1. Data Structure

The group administrator holds all values for managing the group and calculating the keys and public parameter. These values are secret.

\textit{element\_t} is a point on an elliptic curve or an element from \(\mathbb{Z}_p^*\). \textit{mpz\_t} is an integer.
### 5.5. Structure

**typedef struct groupAdmin {**
```c
    mpz_t order; // for internal calculations
    element_t alpha; // part of master secret key
    element_t beta; // part of master secret key
    element_t betaInv; // for internal calculations
    element_t gToAlpha; // part of system parameter
    element_t gToBeta; // part of system parameter
    mpz_t smallest_r; // smallest r for calculating k
    mpz_t k; // current k
    mpz_t a; // current a
} groupAdmin_t;
```

The group entity holds all public values needed for encryption and decryption:

**typedef struct groupEntity {**
```c
    element_t pk1; // sub-key of public key
    element_t pk2; // sub-key of public key
    mpz_t publicParam; // public parameter
    pairing_t pairing; // the used pairing and curve
    element_t h; // needed for encryption
    element_t g; // needed for encryption
} groupEntity_t;
```

The user holds the secret key and an unique id:

**typedef struct user {**
```c
    element_t s1; // sub-key of private key
    element_t s2; // sub-key of private key
    element_t s3; // sub-key of private key
    element_t s4; // sub-key of private key
    element_t s5; // sub-key of private key
    element_t s6; // sub-key of private key
    element_t s7; // sub-key of private key
    int id; // unique id
} user_t;
```

The ciphertext is saved in a struct:

**typedef struct cipher {**
```c
    element_t c1; // cipher component
    element_t c2; // cipher component
    element_t c3; // cipher component
} cipher_t;
```
In order to make the API easy to use there is a group struct. This struct encapsulates every other struct except for ciphertexts. When using the API only this and the cipher struct are needed.

```c
typedef struct group {
    groupAdmin_t* admin; // administrator struct
    user_t* adminUser;   // administrator user struct
    node_t* adminHead;   // user list head
    idNode_t* idHead;    // id list head
    groupEntity_t* groupEntity; // group entity struct
} group_t;
```

### 5.5.2. User Management

The users are managed in a linked list and every user has a single id. The id is used to remove a user from a group. The administrator cannot be removed and is the head node, which saves all information about the list.

When removing a user, the id is saved in another linked list. Every time a user is added to the group, the id list gets checked. In case there are values stored in the id list, the new user gets the id of the first node. Deleted ids are used again in this way.

The administrator node (head) saves the last added node to save time, when appending a new node to the list.

```c
typedef struct node {
    user_t* data;       // the user struct
    struct node* next;  // next node, NULL when last
    int listSize;       // current size
    struct node* lastNode; // current last node
} node_t;
```

```c
typedef struct idNode {
    int id;           // removed id
    struct idNode* next; // next node
    int listSize;     // current size
} idNode_t;
```

### 5.6. Implementation

Figure 5.1 shows an overview of the API. The dge methods are the main methods, which provide full functionality in order to use the implementation. The group struct saves everything needed to create and manage a group. The administrator manages the group with help of the dge methods. Encryption and decryption functionality is also provided by the dge methods.
5.6. Implementation

5.6.1. Methods

The methods shown in the following are written in pseudo code and are referring to methods in the source code.

Figure 5.2 shows an illustration of the group creation and removal of a user in a real case scenario.
1. The administrator creates a group and distributes each user’s private key over a secure channel and saves the own private key. Then the administrator computes the public key and public parameter for the group.
2. The administrator removes a user and updates the public key and public parameter with new values.

Figure 5.3 shows an illustration of the addition of a user to an existing group in a real case scenario.
1. An existing group like the created one from figure 5.2
2. The administrator adds a user to the group, computes a new private key and sends it over a secure channel to the newly added user. The public key and public parameter are updated with new values.
5. Implementation Details

5.6.1.1. Group Setup And Key Generation

During setup all structs are initialized with valid values. Therefore, memory is allocated for every non-primitive variable. The pairing and elliptic curves are set. The “A-type param” (a.param) pairing from the PBC library was used [11]. This curve has the form $y^2 = x^3 + x$. After that all private keys for every user, public key and public parameter are computed.

The algorithm below is referring to createGroup() and firstPublicKeyParam().

---

Figure 5.2.: Illustration of creating a group and removing a member

Figure 5.3.: Illustration of adding a user to an existing group
### 5.6. Implementation

**Algorithm**  group setup and key generation  
**Input:** headNode, groupAdmin, groupEntity  
**Output:** void  

1. Initialize curve and structs  
2. Choose random \( g \in \mathbb{G}_1 \) and update groupEntity  
3. Choose random \( h \in \mathbb{G}_1 \) and update groupEntity  
4. Choose random \( \alpha \in \mathbb{Z}_p^* \) and update groupAdmin  
5. Choose random \( \beta \in \mathbb{Z}_p^* \) and update groupAdmin  
6. Calculate \( g^\alpha \) and update groupAdmin  
7. Calculate \( g^\beta \) and update groupAdmin  
8. for all users in user list do  
9. Calculate secret key like in the scheme and update user  
10. if current \( s7 < \text{smallest}_ri \) then  
   Set new \( \text{smallest}_ri = s7 \) and update groupAdmin  
11. firstPublicKeyParam(headNode, groupAdmin, groupEntity)  

// Calculate the public key and public parameter like in the scheme and update groupEntity

**5.6.1.2. Group Change**

When changing a group, the public key and public parameter have to be recomputed. The user list must also be updated. In case of addition, the new user’s private key is computed.

The following algorithm is referring to addUser() and newPublicKeyParam().

**Algorithm**  user add  
**Input:** headNode, groupAdmin, groupEntity, newUser, idHead  
**Output:** void  

1. Initialize newUser struct  
2. add_userList(headNode, newUser, idHead)  
   // Add the new user to the list. Check if there is a removed id in the id list and reuse it if available  
3. Calculate secret key like in the scheme and update newUser  
4. if new \( s7 < \text{smallest}_ri \) then  
   Set new \( \text{smallest}_ri = s7 \) and update groupAdmin  
5. newPublicKeyParam(headNode, groupAdmin, groupEntity)  

// Calculate new public key and parameter like in the scheme and update groupEntity

The following algorithm is referring to removeUser() and newPublicKeyParam().
5. Implementation Details

Algorithm user removal

Input: headNode, groupAdmin, groupEntity, userId, idHead

Output: int success: 1 = success | 0 = not found

1 remove_userList(headNode, userId, idHead)

   // Searches for the user and in case it exists -> remove
2 if smallest ri = s7 then
3   Find new smallest s7 in updated user list and update groupAdmin
4 newPublicKeyParam(headNode, groupAdmin, groupEntity)

   // Calculate new public key and public parameter like in the scheme and update groupEntity

5.6.1.3. Encryption

The public information needed for encryption is available in the groupEntity struct. The sender encrypts a 128-bit AES key, which is used to encrypt the actual message.

The following algorithm is corresponding to encrypt().

Algorithm encryption

Input: c1, c2, c3, AES-key, groupEntity

Output: ciphertext = (c1, c2, c3)

1 Choose random \( t \in \mathbb{Z}_p^* \)
2 Calculate c1 like in the scheme
3 Calculate c2 like in the scheme
4 Calculate c3 like in the scheme

5.6.1.4. Decryption

The user needs the current public parameter for decryption. The obtained AES key can be used to decrypt the actual message.

The following algorithm is corresponding to decrypt().

Input: ciphertext, user, groupEntity

Output: AES-key

1 Calculate \( \kappa \) like in the scheme
2 Calculate \( \zeta_1 \) like in the scheme
3 Calculate \( \zeta_2 \) like in the scheme
4 Calculate \( \tau \) like in the scheme
5 Decrypt AES-key like in the scheme
6. Evaluation

Nishat’s scheme was measured by the authors in terms of performance. They measured the computation time of encryption, decryption, key generation and rekeying cost after adding and removing a member in relation of the group size. In this chapter theses measurements will be compared to the time the API needs for the same operations.

6.1. Correctness

The correctness of the implementation was tested with unit tests. Those are also included in the API files. The test cases are:

- A group is created and checked if the user list and administrator are correctly initialized
- It is checked if deleting the administrator is rejected
- It is checked if a group with three initial users is correctly initialized
- It is checked if a user can be added correctly to an existing group
- It is checked if a user can be removed correctly from an existing group
- It is checked if removing and adding several users is done correctly. Furthermore, it is checked if new users got the removed ids
- It is checked if group members can correctly compute $k$. Furthermore, forward and backward secrecy is checked with added and removed users
- It is checked if group members can correctly decrypt messages. Furthermore, forward and backward secrecy is checked with added and removed users
6.2. Test Environment

The author’s measurements were running on a Dual Core Intel Pentium processor with 4 GB RAM and 3 GHz x 2 on Ubuntu 14.04 LTS as operating system. The API measurements were running on a Quad Core Intel Haswell i7 4790k processor with 16 GB RAM and 4 GHz x 4 with 4.4 GHz Turbo Boost on Windows 10 64-bit as operating system. The authors used the same library [11] but have not specified what (mathematical) group they have used for their tests. The API is using a 160-bit elliptic curve.

6.3. Storage Cost

Let $n$ be the amount of group members.

1. **Storage at administrator**: Nishat’s scheme proposes storage cost of $O(n)$. This is because the administrator saves $\alpha, \beta, \Gamma = (h, g, g^\alpha, g^\beta), k, a$ and every $r_i$ associated with every user $u_i$ with $i \in \{1, 2, ..., n\}$. The storage cost of the implementation is consistent with the scheme. Instead of saving only $r_i$ of every user, the administrator manages a user list with all user nodes, which needs space of $O(n)$.

2. **Storage at user**: Nishat’s scheme proposes constant cost of $O(1)$. This is consistent with the implementation, which saves the user key and id for each user in a node.

3. **Public key**: Nishat’s scheme proposes constant cost of $O(1)$. This is consistent with the implementation.

4. **Ciphertext**: Nishat’s scheme proposes constant cost of $O(1)$. This is consistent with the implementation.

5. **Public parameter**: Contrary to Nishat’s assumption, the size of the group influences the size of the public parameter. The storage cost is in $O(n)$ and for each user approximately 20 bytes of memory have to be allocated, because every $r_i$ is about 160-bit long. Figures 6.1 and 6.2 below show the allocated size in relation to the group size.

6.4. Computation Cost

The API performance was measured with the build-in Visual Studio performance tools. A group with a given amount of users was created. Then the functions being tested were executed. This was repeated 20 times. Visual Studio calculated the average time needed for every function used in the program. For values up to 300 users the tests were repeated six times. From 400 up to 1000 users the tests were repeated three times, because the computation times are getting fairly long. The standard deviation was calculated with the average times. For better comparison with the article, group sizes of 10,000, 50,000 and 100,000 users were also measured. Those were only calculated once, because the computation times are long with large groups. The exact measure values to all tests can be found in the appendix in section A.
The figures from Nishat’s article are modified. Originally they compared their scheme to another encryption schemes. Those schemes were not covered in this thesis. For reasons of clarity those schemes were removed from the figures.

![Figure 6.1.](image1)

Figure 6.1.: Space consumption with API of public parameter in relation to group size

![Figure 6.2.](image2)

Figure 6.2.: Space consumption with API of public parameter in relation to group size (large scale)
6.4.1. Encryption

![Graph showing time to encrypt one message](modified)

Nishat’s measurement claims that the computation time to encrypt a message is constant ($O(1)$). The group size does not affect the computation time of encryption in the API. The sizes of all operands are constant, so the calculation times are always nearly the same.

A message was encrypted 1000 times with the API and the average was calculated. This test was repeated five times. The standard deviation was calculated with those five average values.

The average time needed for one encryption is **8.38 milliseconds** with standard deviation of **0.017 milliseconds**.

6.4.2. Decryption

![Graph showing time to decrypt one ciphertext](modified)

Nishat’s measurement claims that the computation time to decrypt a message is constant ($O(1)$). Just like in case of encryption, the group size does not affect decryption.
6.4. Computation Cost

The measurement was carried out like the encryption test, but instead with ciphertexts and the decryption function. The average time needed for one decryption is **15.63 milliseconds** with standard deviation of **0.02 milliseconds**.

6.4.3. Key Generation

The authors measured the computation time for generating the public and private key. The calculation times for those operations are constant ($O(1)$) and are not dependent on the group size. The sizes of the operands are constant, so the time needed is always nearly the same.

6.4.3.1. Public Key

![Figure 6.5.: Time for computing public key (Nishat’s measurement)](image)

![Figure 6.6.: Time needed with API for public key generation (large scale)](image)

Figure 6.5.: Time for computing public key (Nishat’s measurement) [9]

Figure 6.6.: Time needed with API for public key generation (large scale)

Figure 6.6 above shows that the result is consistent with Nishat’s results.
6.4.3.2. Secret Key

Figure 6.7.: Time needed for computing private key per user (Nishat’s measurement) \[9\] (modified)

Figure 6.8.: Time needed with API for key generation per user
6.4. Computation Cost

Figure 6.9.: Time needed with API for private key generation per user (large scale)

Figure 6.9. above shows that the result is consistent with Nishat’s results.

6.4.4. Group Changes

Group changes are dependent on the calculation of public key and mostly the public parameter. The authors state, that this would be possible in constant time. The calculation of the public parameter is the big difference between API and Nishat’s scheme, because of the mistake described in chapter 4.

Figure 6.10.: Time needed for rekeying to add and remove a user (Nishat’s measurement) [9] (modified)

Adding or removing a user causes a bit more overhead in the API compared to only rekeying cost, because the user list has to be managed. However, this overhead is extremely small compared to the calculation of the public parameter (for removing a user, small ids were used to make sure the overhead is minimal).
34 6. Evaluation

Figure 6.11.: Time needed with API to add a new user

Figure 6.12.: Time needed with API to add a new user (large scale)

Figure 6.12 and 6.14 show that, contrary to Nishat’s measurements, rekeying cost after adding or removing a user is not constant. The larger the public parameter gets, the longer the multiplications take. A long computation time for group changes is a big disadvantage. For smaller groups up to 1000 members, the time needed resembles approximately a constant behaviour.
Figure 6.13.: Time needed with API to remove a member

Figure 6.14.: Time need to remove a member (large scale)
6.4.5. Group Creation

The time needed for creating a group with the API was also measured. All needed memory gets allocated, every user gets a private key and public key and public parameter are computed. All structs get initialized with values.

Figure 6.15.: Time needed for API to create a group

Figure 6.16.: Time needed for API to create group (large scale)

It is not clear how the curve would continue in figure 6.16. The time complexity of multiplications is not linear [5]. Due to very long computation times it is hard to test even bigger group sizes. It is also unknown how the API will behave on slower CPUs or other architectures.
7. Conclusion

We have seen how Nishat’s scheme works, what problems it has, implemented it and evaluated performance and correctness. Further, we have compared the implementation with the author’s measurements. The proposed constant computation time for group changes cannot be achieved due to the change of the public parameter calculation. The size of the parameter is dependent on the group size with the corrected equation. However, it seems that the scheme could be a valid option for group sizes up to 1000 members. Above 1000 members the time needed for changes is getting too high and it is questionable, whether the scheme can be still considered as truly dynamic. It is desirable to have constant overhead for group changes. The results show, that Nishat’s scheme is not providing this property. The ”one-affects-all-problem” is not resolved like proposed. Although, group members can keep their secret keys, they have to update the linear growing public parameter after every group change.

The scheme preserves forward and backward secrecy, like proposed. In summary, the results are consistent with Nishat’s scheme except for the public parameter.

Further research could be an implementation on embedded systems to evaluate performance on slower processors. It can be checked, whether group sizes up to 1000 members can still be achieved. The evaluation in this thesis was realized with a fairly fast and modern processor. The scheme can be also compared to another group-oriented schemes and to the naive approach.
Appendix

A. Lookup Tables

<table>
<thead>
<tr>
<th>performance test public key</th>
<th>amount user</th>
<th>avg time in ms</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>each test 20 times and then 6 times</td>
<td>10</td>
<td>6,13</td>
<td>0,03</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>6,03</td>
<td>0,01</td>
</tr>
<tr>
<td>each test 20 times and then 3 times</td>
<td>500</td>
<td>6</td>
<td>0,02</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>5,93</td>
<td>0,05</td>
</tr>
<tr>
<td>each test one time</td>
<td>10000</td>
<td>5,92</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>5,9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>5,91</td>
<td></td>
</tr>
<tr>
<td>performance test group generation</td>
<td>amount user</td>
<td>avg time in s</td>
<td>standard deviation</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>each test 20 times and then 6 times</td>
<td>1</td>
<td>0.03731</td>
<td>0.00018</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.16025</td>
<td>0.00125</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.28266</td>
<td>0.0016</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>0.40527</td>
<td>0.00248</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.5269</td>
<td>0.0019</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>0.64639</td>
<td>0.00141</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>0.76718</td>
<td>0.00052</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>0.89093</td>
<td>0.0026</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>1.00965</td>
<td>0.00187</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>1.13314</td>
<td>0.0012</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1.25913</td>
<td>0.00614</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>1.52166</td>
<td>0.00151</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>1.74952</td>
<td>0.00741</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>1.99303</td>
<td>0.0063</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2.23369</td>
<td>0.0022</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.47781</td>
<td>0.0054</td>
</tr>
<tr>
<td></td>
<td>220</td>
<td>2.7328</td>
<td>0.00566</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>2.9805</td>
<td>0.00574</td>
</tr>
<tr>
<td></td>
<td>260</td>
<td>3.22836</td>
<td>0.0067</td>
</tr>
<tr>
<td></td>
<td>280</td>
<td>3.47455</td>
<td>0.00659</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>3.72132</td>
<td>0.00513</td>
</tr>
<tr>
<td>each test 20 times and then 3 times</td>
<td>400</td>
<td>4.94393</td>
<td>0.00194</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>6.17247</td>
<td>0.0132</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>7.37704</td>
<td>0.01529</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>8.59057</td>
<td>0.00658</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>9.84483</td>
<td>0.03012</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>11.07468</td>
<td>0.02577</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>12.28701</td>
<td>0.03045</td>
</tr>
<tr>
<td>each test one time</td>
<td>10000</td>
<td>125.30735</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>651.01333</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>1375.82685</td>
<td></td>
</tr>
<tr>
<td>performance test member add</td>
<td>amount</td>
<td>user</td>
<td>avg time in ms</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------</td>
<td>------</td>
<td>----------------</td>
</tr>
<tr>
<td>each test 20 times and then 6 times</td>
<td>1</td>
<td>19,18</td>
<td>0,1023</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>19,14</td>
<td>0,2552</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>19,15</td>
<td>0,0898</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>19,04</td>
<td>0,1758</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>18,99</td>
<td>0,1108</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>18,89</td>
<td>0,0893</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>18,8</td>
<td>0,1791</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>18,7</td>
<td>1,2832</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>18,77</td>
<td>0,025</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>19,17</td>
<td>1,1908</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>18,72</td>
<td>1,4492</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>18,97</td>
<td>0,1144</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>19,31</td>
<td>0,6642</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>19,02</td>
<td>0,1043</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>18,94</td>
<td>0,088</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>19,04</td>
<td>0,098</td>
</tr>
<tr>
<td></td>
<td>220</td>
<td>18,92</td>
<td>0,984</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>19,04</td>
<td>0,047</td>
</tr>
<tr>
<td></td>
<td>260</td>
<td>18,87</td>
<td>0,0655</td>
</tr>
<tr>
<td></td>
<td>280</td>
<td>18,97</td>
<td>0,0613</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>19,04</td>
<td>0,1</td>
</tr>
<tr>
<td>each test 20 times and then 3 times</td>
<td>400</td>
<td>19,14</td>
<td>0,085</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>19,28</td>
<td>0,025</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>19,47</td>
<td>0,31</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>19,41</td>
<td>0,04</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>19,38</td>
<td>0,0704</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>19,48</td>
<td>0,03</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>19,66</td>
<td>0,12</td>
</tr>
<tr>
<td>each test one time</td>
<td>10000</td>
<td>26,44</td>
<td>0,000</td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>61,1</td>
<td>0,000</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>483,4</td>
<td>0,000</td>
</tr>
<tr>
<td>performance test member remove</td>
<td>amount user</td>
<td>avg time in ms</td>
<td>standard deviation</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------</td>
<td>----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>each test 20 times and then 6 times</td>
<td>1</td>
<td>6,81</td>
<td>0,726</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7,05</td>
<td>0,3631</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>6,72</td>
<td>0,34</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>6,78</td>
<td>0,11</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>6,64</td>
<td>0,0268</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>6,66</td>
<td>0,0367</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>7,07</td>
<td>0,3584</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>7,43</td>
<td>0,45</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>6,64</td>
<td>0,66</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>7,33</td>
<td>0,559</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>7,25</td>
<td>0,6336</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>6,76</td>
<td>0,07</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>6,89</td>
<td>0,1907</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>6,75</td>
<td>0,0415</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>6,78</td>
<td>0,0356</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>6,85</td>
<td>0,016</td>
</tr>
<tr>
<td></td>
<td>220</td>
<td>6,84</td>
<td>0,1125</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>6,87</td>
<td>0,0403</td>
</tr>
<tr>
<td></td>
<td>260</td>
<td>6,84</td>
<td>0,041</td>
</tr>
<tr>
<td></td>
<td>280</td>
<td>6,91</td>
<td>0,029</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>6,91</td>
<td>0,017</td>
</tr>
<tr>
<td>each test 20 times and then 3 times</td>
<td>400</td>
<td>7,13</td>
<td>0,055</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>7,23</td>
<td>0,005</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>7,31</td>
<td>0,05</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>7,49</td>
<td>0,085</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>7,58</td>
<td>0,016</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>7,72</td>
<td>0,015</td>
</tr>
<tr>
<td>each test one time</td>
<td>1000</td>
<td>8</td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>22,71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>405,18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>586,39</td>
<td></td>
</tr>
<tr>
<td>storage test public parameter</td>
<td>amount user</td>
<td>average space in kbytes</td>
<td>standard deviation</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------</td>
<td>--------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>each test 20 times and then 6 times</td>
<td>1</td>
<td>0.04</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.436</td>
<td>0.0012</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0.83</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>1.224</td>
<td>0.00464</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>1.617</td>
<td>0.00255</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.012</td>
<td>0.00464</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>2.404</td>
<td>0.0030984</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>2.798</td>
<td>0.0026533</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>3.192</td>
<td>0.0025933</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>3.586</td>
<td>0.0026533</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>3.980</td>
<td>0.0021541</td>
</tr>
<tr>
<td></td>
<td>220</td>
<td>4.374</td>
<td>0.0019596</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>4.767</td>
<td>0.0028</td>
</tr>
<tr>
<td></td>
<td>260</td>
<td>5.162</td>
<td>0.0026533</td>
</tr>
<tr>
<td></td>
<td>280</td>
<td>5.553</td>
<td>0.0026533</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>5.949</td>
<td>0.0024</td>
</tr>
<tr>
<td>each test 20 times and then 3 times</td>
<td>400</td>
<td>7.919</td>
<td>0.0031241</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>9.888</td>
<td>0.0043813</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>11.857</td>
<td>0.0048</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>13.826</td>
<td>0.0232</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>15.790</td>
<td>0.0064992</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>17.766</td>
<td>0.01344</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>19.736</td>
<td>0.0054991</td>
</tr>
<tr>
<td>each test one time</td>
<td>10000</td>
<td>196,988</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>984,824</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>1969,556</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


