# A Java Framework for Broadcast Encryption Algorithms

Master's Thesis in Information Theory
by
Tobias Hesselius and Tommy Savela

LiTH-ISY-EX-3563-2004
Linköping 2004

# A Java Framework for Broadcast Encryption Algorithms

Master's Thesis in Information Theory
at the Linköping Institute of Technology
by
Tobias Hesselius and Tommy Savela

LiTH-ISY-EX-3563-2004

Supervisor: Kristin Anderson
Examiner: Viiveke Fåk
Linköping 2004-09-10

LINKÖPINGS UNIVERSITET

**Avdelning, Institution**
Division, Department

Institutionen för systemteknik
581 83 LINKÖPING

**Datum**
Date
2004-09-10

**Språk**
Language
Svenska/Swedish
X Engelska/English

**Rapporttyp**
Report category
Licentiatavhandling
X Examensarbete
C-uppsats
D-uppsats
Övrig rapport

**ISBN**

**ISRN** LITH-ISY-EX-3563-2004

**Serietitel och serienummer** **ISSN**
Title of series, numbering

**URL för elektronisk version**
http://www.ep.liu.se/exjobb/isy/2004/3563/

**Titel** Ett ramverk i Java för prestandatest av broadcast-krypteringsalgoritmer
Title

A Java Framework for Broadcast Encryption Algorithms

**Författare** Tobias Hesselius, Tommy Savela
Author

**Sammanfattning**
Abstract
Broadcast encryption is a fairly new area in cryptology. It was first addressed in 1992, and the research in this area has been large ever since. In short, broadcast encryption is used for efficient and secure broadcasting to an authorized group of users. This group can change dynamically, and in some cases only one-way communication between the sender and receivers is available. An example of this is digital TV transmissions via satellite, in which only the paying customers can decrypt and view the broadcast.

The purpose of this thesis is to develop a general Java framework for implementation and performance analysis of broadcast encryption algorithms. In addition to the actual framework a few of the most common broadcast encryption algorithms (Complete Subtree, Subset Difference, and the Logical Key Hierarchy scheme) have been implemented in the system.

This master's thesis project was defined by and carried out at the Information Theory division at the Department of Electrical Engineering (ISY), Linköping Institute of Technology, during the first half of 2004.

**Nyckelord**
Keyword
broadcast encryption, Subset Difference, Complete Subtree, Logical Key Hierarchy, simulation

# Abstract

Broadcast encryption is a fairly new area in cryptology. It was first addressed in 1992, and the research in this area has been large ever since. In short, broadcast encryption is used for efficient and secure broadcasting to an authorized group of users. This group can change dynamically, and in some cases only one-way communication between the sender and receivers is available. An example of this is digital TV transmissions via satellite, in which only the paying customers can decrypt and view the broadcast.

The purpose of this thesis is to develop a general Java framework for implementation and performance analysis of broadcast encryption algorithms. In addition to the actual framework a few of the most common broadcast encryption algorithms (*Complete Subtree*, *Subset Difference*, and the *Logical Key Hierarchy* scheme) have been implemented in the system.

This master's thesis project was defined by and carried out at the Information Theory division at the Department of Electrical Engineering (ISY), Linköping Institute of Technology, during the first half of 2004.

# Acknowledgements

Thanks to our examiner Viiveke Fåk and supervisor Kristin Anderson for giving us this very interesting project, and for (hopefully) approving it. A special thanks to Kristin for having the time and patience to answer all the questions we had about this, for us, new area in cryptology.

We would also like to thank the Department of Electrical Engineering (ISY) for providing an office and lots of free coffee for the duration of this project.

# Contents

# Chapter 1

# Introduction

*This chapter contains the background, purpose and method of this master thesis, and an outline of the report.*

## 1.1 Background

In 1992 [1], Shimson Berkovits introduced the subject of broadcast encryption, and the subject was later formalized by Amos Fiat and Moni Naor in 1993 [2]. The subject of broadcast encryption has received much attention since then, and several algorithms have been presented.

Broadcast encryption solves the problem how to efficiently do a secure broadcast to an authorized set of users. The set of users can change dynamically, for example when a customer in a digital TV network starts or ends a subscription to a channel. The network communication between the broadcaster and the users are often one-way.

Another application of broadcast encryption is for content protection of recordable media such as DVD (see [9]). In this case the content is stored on a medium that may be accessed several years later. This situation does not allow any direct communication between the recorder and the player. The content protection for DVDs today uses a shared-secret scheme called CSS (*Content Scrambling System*). This system was broken in 1999 when a person found the shared secret. If broadcast encryption had been used instead, this attack would not be as severe, since the new DVDs would switch encryption keys and the attack would no longer be effective.

When deciding on an algorithm for a specific scenario where broadcast encryption is needed, there is one important fact to remember: The efficiency of the algorithms will vary depending on the conditions of the scenario. Some of the important factors can be estimated, like maximum number of users. Another factor is if the set of authorized users will be fairly static or change often (the *mobility* of the users). It is therefore desirable to select the algorithm that has the best performance for the specific scenario.

The space and time complexities for the algorithms can often be calculated. But these estimates do not take into account how often users are added or removed or how the users are organized. This might significantly affect the performance of the algorithms.

To easily compare different types of algorithms it would be preferred to perform practical simulations of each algorithm to see how they behave in each specific scenario, and then compare the simulation output. This problem is the incentive for this thesis.

## 1.2 Purpose

The purpose of this thesis is to provide an environment for implementation and analysis of broadcast encryption algorithms. This Java framework should include the basic building blocks needed to easily develop and test new algorithms or improve already existing ones. The specific requirements for the system is presented in section 3.2.

## 1.3 Method

This thesis work was initiated in February 2004. The guidelines for the project were discussed and a preliminary timeline was constructed. It was decided that this thesis would be complete in September 2004 at the latest, although most of the work should be done before the summer.

The main phases for this project are:

1. Gathering of information and theoretical background.
2. Working out the requirements of the project.
3. Designing the system.
4. Implementing and testing the system.
5. Writing the report.

The writing of the report is an ongoing activity throughout the entire duration of the project. It is also estimated that the design and implementation will go through several iterations before reaching the final product. This will be the most time-consuming part of the project.

## 1.4 Outline

**Chapter 1: Introduction** explains the background and purpose of this thesis, along with the planned demarcations and method.

**Chapter 2: Theory** provides the theory behind broadcast encryption and gives a description of some of the most interesting algorithms.

**Chapter 3: System Overview** describes the system specifications and how it is designed, and how to extend the system with new algorithms and simulation types.

**Chapter 4: Results** shows how the final program looks and a few examples of simulation output created by the program. The limitations and advantages of this program is also discussed.

**Chapter 5: Summary** contains a brief evaluation of the results presented in this report and summarizes the application areas and benefits of this system. Ideas about future work is also mentioned.

# Chapter 2

# Theory

*This chapter contains an overview of the broadcast encryption area in general, and more detailed descriptions of the algorithms implemented in this system.*

## 2.1 Introduction

Broadcasting means transmitting information through a medium that is accessible to multiple receivers. A radio transmission for instance uses air as a medium, and everyone with a radio receiver is able to listen to the broadcast. Usually this communication is one-way, meaning that the receivers are not able to send anything back to the broadcast center.

In some applications it is desirable to secure the content of the broadcasted message so that only the authorized users are able to read it. In broadcast encryption theory these users are said to be *privileged*, and the non authorized users are said to be *revoked*. The terms receivers and users are sometimes used interchangeably. The difference is that all the receivers can access the broadcast message, but only the privileged users can access the content of the message.

The broadcast message is usually divided into a header and a body part. The body contains the protected content and the header contains information needed to access the content (key material and user memberships). The header is the most important part when analysing these algorithms.

The most simple broadcast encryption scheme would be to encrypt the message once for each privileged user and then broadcast all encrypted messages. This is obviously a very inefficient scheme in terms of processing time and broadcast message size. The aim of all intelligent schemes is to reduce the processing time at both the broadcast center and at the receivers, to reduce the broadcast message size and to reduce the storage size at the receivers. The broadcast message size is dependent on the number of encrypted messages it contains, but also on the size of the header for the broadcast message. The header often contains critical information such as encrypted keys and information about which receivers can decrypt the message.

A broadcasting algorithm encrypts a message so that multiple users can decrypt it. This can be done in many ways depending on which algorithm is used. To send a message to multiple users means grouping them together. The way in which this is done is critical to the performance of the algorithm. The method of grouping might be dynamic or predefined. In the second case, performance will be affected by how the privileged and revoked users are ordered within the set. It is thus very important that grouping is done in an intelligent manner.

In the dynamic case, one way to do this is to build a *key graph* that is a set of encryption keys ordered in a graph (see section 2.3.1 for more details). The authorized users are added to this graph, and the keys are distributed. When a user is added or removed from the authorized set a rekeying strategy is used, changing the appropriate keys in the graph and transmitting the new keys to the subscribed users. The strategy is constructed in a way so that it guarantees that the newly added user can decrypt the following transmissions, and that any removed user's keys are made unusable. This is called the *Logical Key Hierarchy* scheme and is described later in this chapter.

In many cases, one does not want to force the user to be connected to the broadcast network at all times. In the example of digital TV, the user must still be able to watch the subscribed channels after the receiver has been turned off during the night or unplugged from the network for a period of time. This means that the broadcast center cannot send rekeying information when a user is added or removed, since this information might be lost for some receivers that require this information to function properly.

This is the same problem we get when the receivers are *stateless*, meaning that the receivers cannot update their state (or keys) between sessions. All information needed to decrypt the message (given the information in the current broadcast) has to be stored in the receivers from start. This is again the case for many digital TV networks, where the user usually receives a smartcard containing the decryption keys when starting a channel subscription for the first time.

The storage size requirement for the receiver is a very important factor in a good broadcast encryption scheme. This is because the receiver's secure storage space is often very limited, e.g the memory capacity of a smartcard, or the available memory in a mobile telephone. At the same time, many of the applications for broadcast encryption require the receivers to store a large number of keys. Fortunately there are algorithms to effectively deal with these problems.

## 2.2 Stateless Subset-Cover Algorithms

The algorithms that will be presented here are *stateless*, which means that the initialization step for the receivers only needs to be performed once. After that, the receivers will be able to decrypt any message, as long as they are privileged, without having to update any of their stored information.

Two algorithms will be explained in this chapter: *Complete Subtree* and *Subset Difference*. These algorithms are *flexible* with respect to the number of revoked users, $r$, which means that the storage size at the receiver is not a function of $r$. This is an important characteristic of the algorithms because it allows dynamic changes in user access rights without having to update the receivers.

It has also been proven that the *Subset Difference* algorithm offers a substantial improvement over other methods (see section 2.4) in terms of efficiency. This improvement is due to the fact that the key assignment is computational rather than information-theoretic. For a more detailed discussion and proof see [3].

These algorithms give a pre-defined grouping of the users. Each group is called a *subset*, $S$, and a user can be a member of several subsets. To distinguish these subsets each subset is assigned an index. In the *Complete Subtree* algorithm this index is simply an integer, $i$, refering to a node in a binary tree. In the *Subset Difference* algorithm this index is a pair of integers, $(i, j)$, refering to two nodes.

The *cover*, $C$, is defined as the set of subsets that precisely contain all the privileged users, $U$, and none of the revoked users, $R$. The terms user and receiver will be used interchangeably in the following sections. The subsets in a cover are always disjoint, which means that a user belongs to at most one subset in the cover. This is not essential for the algorithms to work, but rather a consequence of how the subsets are organized, as will be seen later.

The cover and set of privileged users can be expressed as shown below. The notation is for the *Complete Subtree* algorithm but can easily be rewritten to comply with the *Subset Difference* algorithm.

$$C = \{S_{i_1}, S_{i_1}, \ldots, S_{i_m}\}$$

$$U = \bigcup_{j=1}^{m} S_{i_j}$$

Each subset is associated with a secret key, $L$. This key is only known to those users that belong to that subset. The initialization step encompasses sending the secret keys to the receiver and these keys make up the secret information stored at the receiver. Actually the secret information does not have to consist of the keys directly, rather it must contain the information needed to be able to deduce all the keys, which might be much more efficient (as in the *Subset Difference* algorithm).

### Encryption Functions

All of these algorithms share the need for two basic encryption functions. The first function is denoted $F_K(M)$ and is used to encrypt the message, $M$, which is the content of the broadcast message. This message may be of substantial length, which suggests that $F$ should be very fast. A good choice is to let $F$ be a stream cipher.

The second function is denoted $E_L(K)$ and is used to encrypt a secret key, $K$. These keys are short and of constant length, which means that a slower encryption function might be used for $E$. Furthermore, usually only one decryption is needed at the receiver. Typically, a block cipher is used for $E$.

### Steiner Tree

To find the cover for a set of privileged users, these algorithms use the *Steiner tree* [10] for the set of revoked users. This is simply the tree stripped from all privileged users. It is created by marking the edges between the revoked users and the root node. In figure 2.1 the revoked users are marked by dark circles, the privileged users are marked by grey squares and the Steiner tree is drawn with a dashed line.

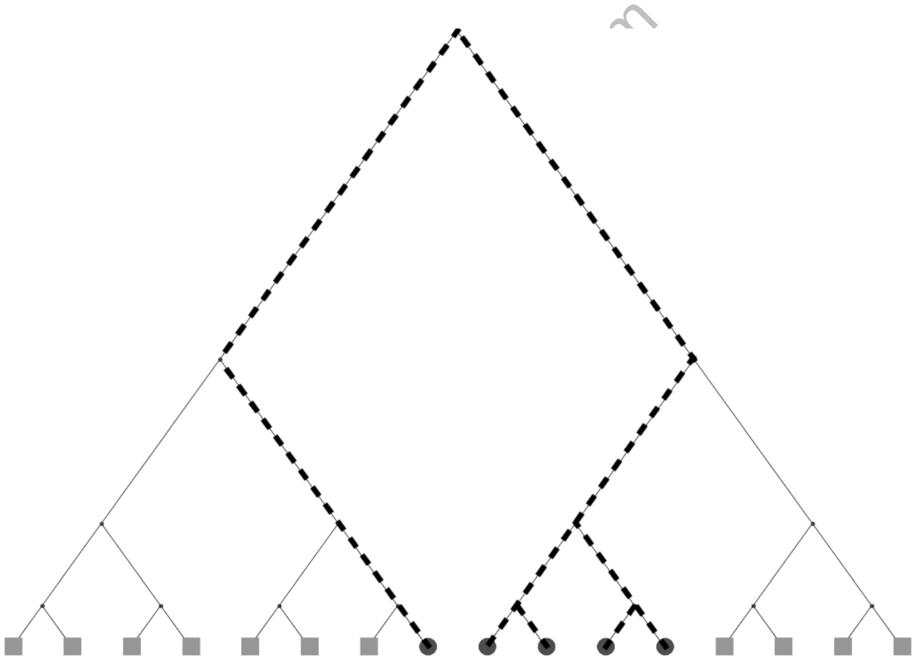


Figure 2.1: Steiner tree for revoked users.

### 2.2.1 Complete Subtree

In this algorithm the subsets can be graphically represented as sets of nodes in a binary tree. Each user is a leaf in the tree. A subset $S_i$ with node index $i$ is then defined as containing all users that are descendants of this node. In figure 2.2 five users are revoked, which results in a cover that contains four subsets: $S_3$, $S_9$, $S_{21}$ and $S_6$. The privileged trees are marked by a light gray color.

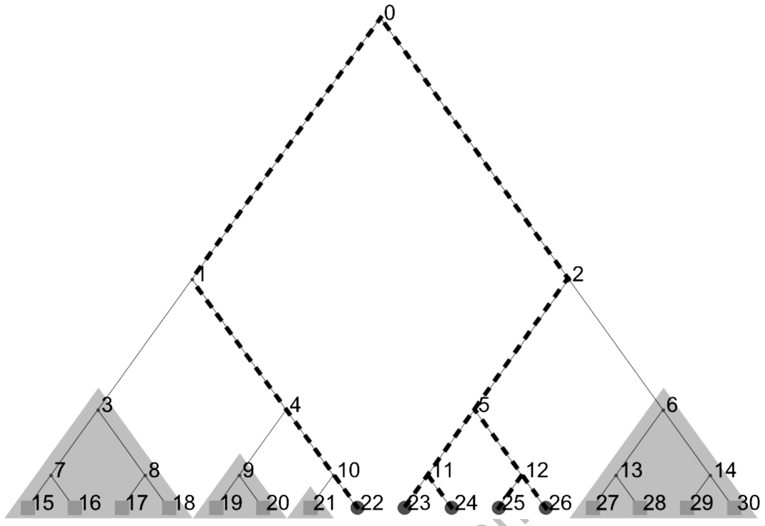


Figure 2.2: Steiner tree and subset-cover for the *Complete Subtree* algorithm. The users at node 22-26 are revoked.

The tree is always complete (all the leaves are at the same depth, and all internal nodes have degree 2), so the number of leaves $N$ in the tree is always a power of two. This is the upper bound for the number of users in the system. Of course the system might only utilize a portion of this. The total number of subsets is $2N-1$ and each subset $S_i$ is specified by an index $i \in \{0, ..., 2N-2\}$ (see figure 2.2).

#### Initialization

To begin with, the *broadcast center* must generate random keys for each subset in the tree. Each user is then supplied with its secret information, through a private back-channel. For example, this secret information might be sent to the user on a smartcard or coded into the receiver when it is manufactured.

A receiver must be able to deduce all the secret keys for the subsets it belongs to, from the secret information it has been given from the broadcast center. In the *Complete Subtree* scheme the secret information is simply the keys for all the subsets the user belongs to, in other words, all the subsets from the leaf to the root node. The size of the secret information is therefore $O(log(N) + 1)$.

The Steiner tree for the set of revoked can easily be used to generate the cover: simply add all the subsets that are at distance one from the Steiner tree graph. These subsets do not belong to the Steiner tree and therefore all users of those subsets must be privileged.

### Encryption

The purpose of encrypting the message is that only the privileged users should be able to read it. These users are all enveloped by the subset-cover. This means that they are a member of one subset in the cover and therefore have the key to that subset. The broadcast center uses these keys to encrypt the message in the following way:

1. Choose a random session key $K$ and encrypt the message using an encryption function $F_K(M)$.

2. For each subset in the cover, encrypt the session key using an encryption function $E_{L_i}(K)$, where $L_i$ is the secret key associated with that subset.

3. Add the encrypted keys along with their indexes to the broadcast message header and the encrypted message to the broadcast message body.

Only the privileged users will be able to decrypt the session key since none of the revoked users have a subset key in the cover.

### Decryption

Decryption is a simple matter since the receiver has stored all the keys that it needs to decrypt. The only problem is to search if the user belongs to any subset specified in the broadcast message header. When a subset is found the corresponding key can be retrieved from the secret information in constant time. The decryption can be divided into these steps:

1. Search the header for a subset $S_i$ that the user belongs to.

2. Retrive the subset key $L_i$ from the secret information.

3. Decrypt the session key $K$ using $E_{L_i}^{-1}(E_{L_i}(K)) = K$.

4. Decrypt the message using $F_K^{-1}(F_K(M)) = M$.

One way to check if the user belongs to a subset is to trace the path from the user to the root, and check if any of the subsets along the path are in the header. Another more efficient way is to represent the index by a binary bitset , where a left node is represented by a 0 and a right node is represented by a 1. For example the subset with index 9 would be represented by the bitset 010. To check if a user belongs to a subset, simply check if the subset bitset is a suffix in the user bitset.

### 2.2.2 Subset Difference

This algorithm has many similarities with the *Complete Subtree* algorithm. It may also be represented as a binary tree with the users as the leaves, although as will be shown, this algorithm is more efficient in describing the subset cover. This is mainly because a user may belong to substantially more subsets than in the previous algorithm.

In the *Subset Difference* algorithm the subsets are defined by two nodes in the tree. The subset $S_{i,j}$ is defined as containing all users that are descendants of node $i$, but not descendants of node $j$. This can be written as $S_{i,j} = S_i \setminus S_j$.

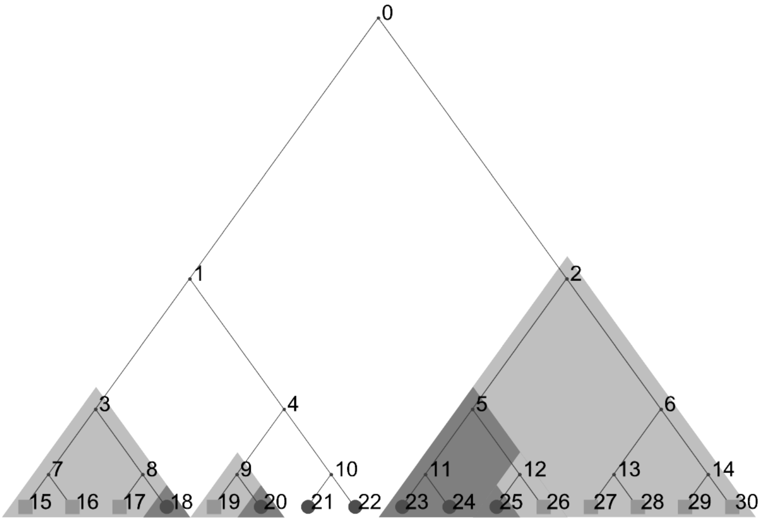


Figure 2.3: Subset-cover for the *Subset Difference* algorithm. The users at node 18 and 20-25 are revoked.

In figure 2.3 seven users are revoked, which results in a cover that contains four subsets: $S_{3,18}$, $S_{9,20}$, $S_{2,5}$ and $S_{12,25}$. The subsets can be seen as shaded triangles in the tree. The light gray color marks a privileged tree and the dark gray color marks a revoked tree.

### Labels

The number of possible keys in this algorithm is substantially larger than in the previous algorithm, $O(N)$ instead of $O(log(N))$. In general this amount of keys is impossible to store directly at the receiver. Instead a structure called *label* is introduced. Each node is associated with a label, $I$, and all possible subset keys can then be derived from these few labels.

A label is a random set of bits that are generated during the initialization step of the algorithm (just like the keys in the *Complete Subtree* algorithm are generated).

To derive the subset keys a pseudo-random sequence generator, $G : \{0,1\}^n \rightarrow \{0,1\}^{3n}$, is used. This generator is a strong one-way function that triples the input length. It is crucial for the security of the algorithm that this function is not invertible. The output of this function is divided into three parts: left, right and middle. The left and right parts are called *intermediate labels*, denoted $I_{i,j}$, and are used when initializing the receivers. For simplicity, the generator is sometimes seen as a combination of three separate functions: $G_L$, $G_R$ and $G_M$, producing the left, right and middle part of the output.

When generating a subset key the generator is applied to a label in a recursive manner. In figure 2.4 the procedure of generating the key for subset $S_{1,9}$ and $S_{2,12}$ is illustrated (with two separate notations). The label for node 2, $I_2$, is used to start with. It is passed through the generator $G$ to produce the output $W_2$. Since node 12 is a descendant of the left child of node 2, the left part of $W_2$ is passed through the generator to give the output $W_{2,5}$. In the next iteration, node 12 is a descendant of the right child of node 5, so the right part of $W_{2,5}$ is used to generate the output $W_{2,12}$. To get the key $L_{2,12}$ the middle part of $W_{2,12}$ is extracted.

This reduces the amount of information to store in the receivers since one label may be used to derive all subset keys that originate from that label.

### Initialization

Creating the cover is slightly more complex than in the *Complete Subtree* algorithm. It requires a recursive algorithm that operates on the Steiner tree. The subsets can be traced in the tree by starting at a node that has out-degree one, and ending at a node that has an even out-degree (zero or two). This creates *chains* in the tree that, for a subset $S_{i,j}$ in the cover, start at node $i$ and end at node $j$.

In figure 2.5 these chains circle the four subsets in the cover. For example, the chain starting at node 2 and ending at node 5 creates the subset $S_{2,5}$. The Steiner tree at node 2 has an out-degree of one and the node at 5 has an even out-degree.

The broadcast center must also randomly select labels for all the nodes in the tree. Then it must supply the receivers with the labels they need to derive all the

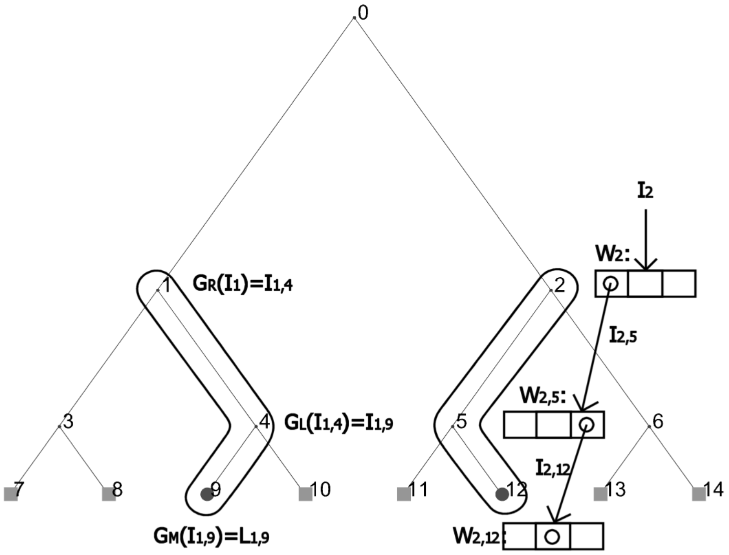


Figure 2.4: Key and label generation for the *Subset Difference* algorithm.

subset keys for the subsets they belong to. This must be done in a way so that the security of the algorithm is not compromised, in other words, the receiver must not be able to derive keys for subsets that it does not belong to. For example, the user at node 11 in figure 2.4 must not receive label $I_2$ since it could then derive the subset key $L_{2,11}$.

In figure 2.6 all the intermediate labels that the user at node 11 need are marked by dashed lines and the nodes just outside the path to the root are circled. These labels make up the secret information for the user at node 11. Since $G$ is irreversible the user cannot obtain the original node labels from these intermediate labels. In the general case, if a user $u$ belongs to a subset $S_i$, then that user must be able to derive all subset keys for subsets of the form $S_{i,j}$, where $u \notin S_j$. By giving the user the intermediate labels of the form $I_{i,j}$ the user can derive all the keys it could possibly need.

### Encryption

The encryption scheme is exactly like in the *Complete Subtree* algorithm, except that the subsets and keys are indexed by the pair $(i, j)$. The encryption of a message $M$ can be divided into these steps:

1. Choose a random session key $K$ and encrypt the message using an encryption function $F_K(M)$.

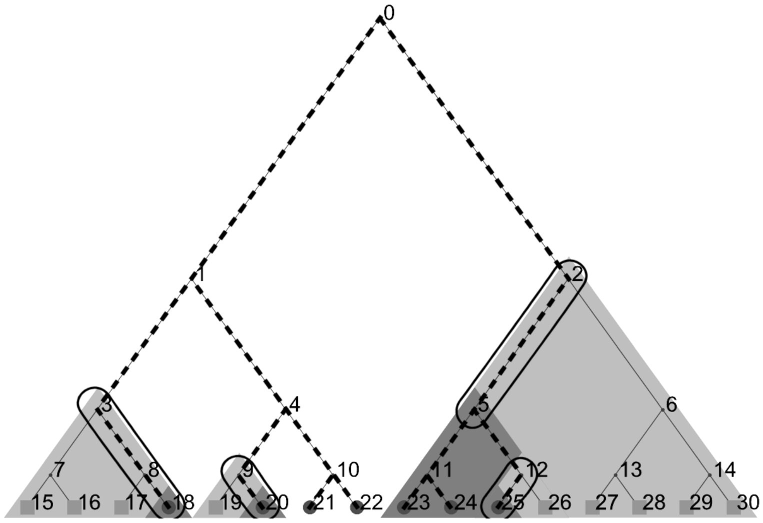


Figure 2.5: Steiner tree and subset-cover for the *Subset Difference* algorithm.

2. For each subset $S_{i,j}$ in the cover, encrypt the session key using an encryption function $E_{L_{i,j}}(K)$, where $L_{i,j}$ is the secret key associated with $S_{i,j}$.

3. Add the encrypted keys along with their indexes to the broadcast message header and the encrypted message to the broadcast message body.

### Decryption

The decryption step is exactly like in the *Complete Subtree* algorithm, except that finding the subset key is a little more complicated. Once the user has found which subset it belongs to the key for that subset must be derived from its secret information. These means finding the intermediate label and deriving the key from this label, as explained in section 2.2.2. The decryption can be divided into these steps:

1. Search the header for a subset $S_{i,j}$ that the user belongs to.

2. Find the intermediate label in the secret information from which to derive the subset key.

3. Derive the subset key $L_{i,j}$ from intermediate label.

4. Decrypt the session key $K$ using $E^{-1}_{L_{i,j}}(E_{L_{i,j}}(K)) = K$.

5. Decrypt the message using $F^{-1}_K(F_K(M)) = M$.

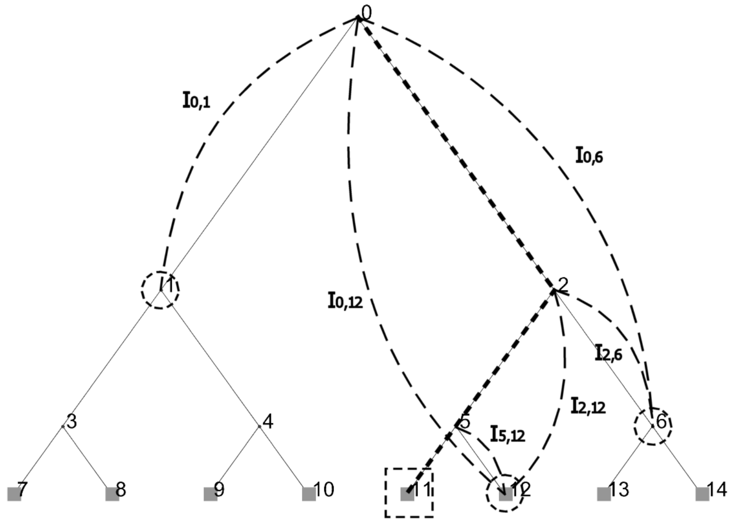


Figure 2.6: Secret labels for user at node 11 in the *Subset Difference* algorithm.

### 2.2.3 Security

The security of these two algorithms is said to be broken if an adversary that is not a privileged user is able to decrypt a broadcast message. The adversary may collect the secret information of all revoked users in his attempt to break the scheme. He may also be able to influence the choice of messages encrypted (chosen plaintext). Even with this kind of attack it is improbable that an adversary can distinguish an encryption of a chosen plaintext from an encryption of a random string (assuming a good choice of encryption functions $F$ and $E$).

For the security of these algorithms to hold they must fulfill a property called *key-indistinguishability*. This means that a subset key $L$ cannot be distinguish from a random key, by all the revoked users. This is trivially true for the *Complete Subtree* algorithm, since each subset key is chosen at random. For a detailed discussion on why this implies the security of the *Subset Difference* algorithm, see [3].

During the initialization step the secret information must be delivered to the users through a secure back-channel. If an adversary obtains this information then the security of the algorithm is temporarily compromised until the affected users have been revoked.

## 2.3 Stateful Algorithms

As opposed to the *stateless* algorithms, a *stateful* algorithm requires that the receivers have to be able to update the stored keys, usually when users are added or removed from the privileged user set. This usually also means that an privileged receiver has to be connected to the broadcast network at all times, in order not to lose any key update messages that might be sent. This is the case with the *Logical Key Hierarchy* scheme that is explained below.

### 2.3.1 Logical Key Hierarchy

The *Logical Key Hierarchy* (LKH) scheme was first presented in 1997 by a group led by Chung Kei Wong [5]. The basic idea of LKH is to build a graph that contains a set of encryption keys (this graph is called a *key graph*), and add the privileged users to it. When adding or removing a user, the keys in the graph are updated in a way that guarantees that a newly added user cannot use the obtained keys to decrypt previous broadcasts (called *backward access control*), and that a removed user's keys can no longer be used for decrypting future broadcasts (called *forward access control*). Each time the key graph is reconstructed the newly changed keys are distributed to a subset of the users. This also means that as opposed to the Complete Subtree and Subset Difference algorithms, the LKH scheme does not work with stateless receivers.

Below is a more detailed description of how the key graph is constructed and how adding and removing a user is done.

#### Structure of the Key Graph

The key graph is a directed acyclic graph with two types of nodes: the *u-nodes* representing users and the *k-nodes* representing keys. Each k-node is assigned a unique random key. At first the graph does not contain any nodes except for the root node. More nodes are added dynamically when a user wants to join the graph, as explained below.

For simplicity we assume that the graph is constructed as a tree with *degree d*. The tree degree is the maximum number of incoming edges of a node in the tree. An example of this can be seen in figure 2.7.

#### Adding and Removing Users

The broadcast center that handles the graph is called the *server*. When a user sends a join request to the server, the server and user first authenticate each other using a protocol such as SSL. If the user is authenticated and accepted to join

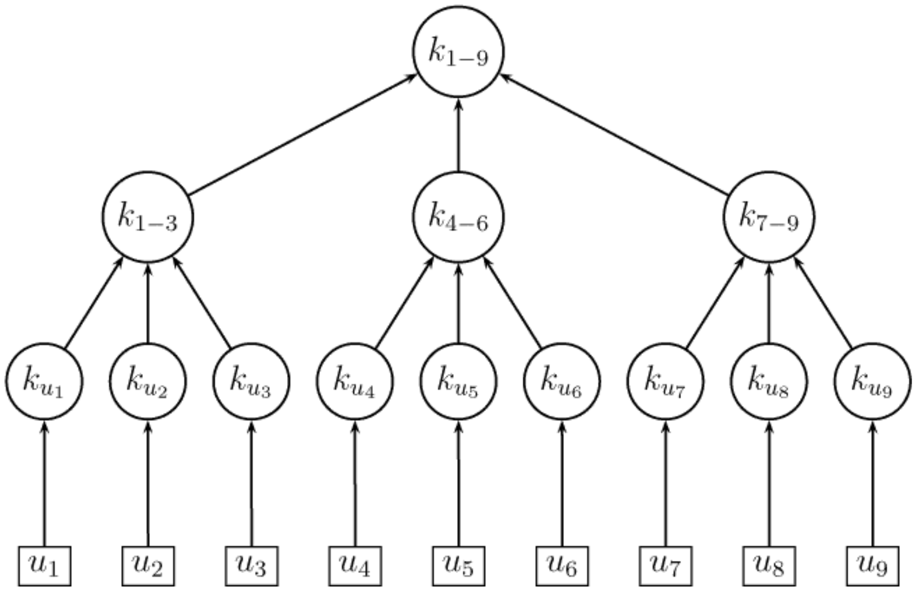


Figure 2.7: A key tree of degree 3 with 9 users, thus the tree is *full*. $u_i$ represents user $i$, and $k_{ui}$ is the user's *individual key*.

the group, a u-node and the corresponding k-node that contains the *individual key* (that will only be known to the server and this user, see figure 2.7) are created. This individual key is then securely transmitted to the user, for example by broadcasting it encrypted with the user's public key.

The next step is to find the *joining point* (the node to which the individual key is added as a child) of the new user. If there is a k-node that has room for more children (the node degree is less than the tree degree), this node is selected as the joining point. If not, a new subtree is created and added to the key tree as shown in figure 2.8. After the joining point has been decided, the new user is added to that node as a child.

The new user should then be given all keys in the path from (and including) the root to the joining point. Since each user will need to store all keys from the root to the individual key, the required storage space is $O(h)$, where $h$ is the number of nodes from the root to the individual key (the *height* of the tree). However, using the existing keys from the key tree will allow the new user to decrypt previous broadcasts. To avoid this, all k-nodes on the path from the joining point to the root have to be assigned new keys. After generating new keys the broadcast center needs to distribute all modified keys to the users that are descendants of the modified k-nodes. This will also include the newly added user. By doing this, backward access control is guaranteed.

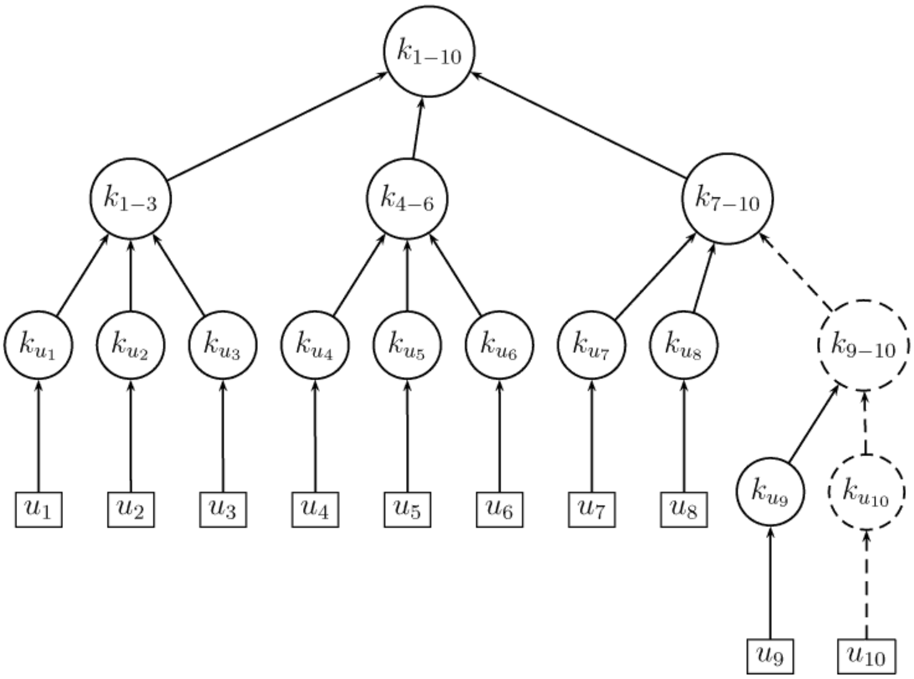


Figure 2.8: The previously full key tree extended with one more user. The new nodes and edges are dashed.

To accomplish this the server sends rekeying messages to the users, containing one or more encrypted keys. The chosen *rekeying strategy* decides how to do this. Some examples of rekeying strategies are *user-oriented rekeying*, *key-oriented rekeying* and *group-oriented rekeying*. These strategies differ in how to construct and send the rekey messages, primarily in what key to use when encrypting the modified keys. In short, the differences are as follows:

**User-oriented rekeying** constructs a rekey message for each user, that contains exactly the new keys needed by that user. The keys are encrypted using a key held by the user, and then distributed to that user.

**Key-oriented rekeying** encrypts each key individually. All updated keys are encrypted with each of its children and distributed to the *user set* of that child. The *user set* of a key is the set of users that share that key.

**Group-oriented rekeying** constructs a single rekey message containing all updated keys encrypted with the child nodes' keys. This message is then distributed to the entire group.

See [5] for more details on these strategies.

Removing a user is done in a similar way. First the corresponding u- and k-nodes are removed from the graph. To prevent the removed user from decrypting future broadcasts, new keys are calculated for all k-nodes in the path from the leaving point to the root. These new keys are then distributed to the remaining users in the same way as described above. This guarantees forward access control.

### Individual vs Batch Rekeying

The above algorithm for adding and removing users from the key tree is called *individual rekeying*. The name comes from the fact that rekey messages are sent after each individual join/leave. However, individual rekeying has two problems. The first problem is that this algorithm is very inefficient when doing a large amount of join/leaves, since rekey messages are sent after each request. This is often not needed, especially if the message broadcasts happen rarely compared to the frequency of join/leave requests. The other problem is an out-of-sync problem between keys and data (see [7] for a discussion about this problem).

A solution for the first problem would be to collect all join/leave requests that arrive over a period of time (the *rekey interval*), and process all of them at the same time. Rekey messages are then created from the resulting key tree, after all requests have been processed. This reduces the number of rekey sessions from $J+L$ (the number of joining and leaving users) to 1. The *batch rekeying* algorithm presented in [7] does this by defining a set of rules for how to add and remove users from the tree, and by using node markings to decide what node keys needs to be updated. This algorithm is described below.

- $J = L$: All leavers are replaced by the joiners. All nodes from the replacement locations to the root are marked UPDATE.
- $J < L$: Replace the J shallowest leavers with the J joiners. Mark all nodes from the replacement locations to the root UPDATE. Mark the remaining leavers DELETE. If a node's children are all marked DELETE, mark it DELETE as well. Mark all the nodes lying on the path from a node marked DELETE to the root UPDATE.
- $J > L = 0$: Find a shallowest leaf node $v$ and remove it from the tree. Create a new tree $T$ that has all joiners and $v$ as leaf nodes. Attach this tree to the old location of $v$. Mark all $T$'s internal nodes NEW and mark all the nodes from the root to the parent of $v$'s old location UPDATE.
- $J > L > 0$: Replace all leavers with joiners. Find the shallowest leaf node $v$ of the replaced nodes and remove it from the tree. Construct a new tree $T$ that has the remaining joiners and $v$ as leaf nodes. Attach the tree to the

old location of $v$. Mark all $T$'s internal nodes NEW and mark all the nodes from the root to the parent of $v$'s old location UPDATE.

After doing this, all nodes marked DELETE are removed from the tree. Rekeying messages are then created according to the selected rekeying strategy, using the nodes marked NEW or UPDATE. For examples and more discussions about this algorithm, see [7].

Some relaxed versions of batch rekeying exists, for example the *simple-batch* algorithm in [8]. This algorithm offers nearly the same performance as the batch rekeying, while having a less complex implementation.

### Encryption/Decryption

When broadcasting a message to all privileged users, the message can simply be encrypted with the root key. Since the authorized users (and only those) already have the current root key from the rekeying messages, they can decrypt the message without doing any additional computation. Thus the complexity for both encrypting and decrypting a broadcast is $O(1)$. In the same way, it is also possible to encrypt a message to only a group of users that share a group key by encrypting with this shared key.

### Security

One problem with using rekey messages is that a user could masquerade as the server and send unauthorized rekeying messages. To prevent this, a message digest such as MD5 can be calculated for each rekeying message and each digest signed with the server's private key. The signed digest is then transmitted along with the rekeying message. However, this would require as many digital signature operations as there are messages. Since digital signature operations are computationally expensive [6], it would be preferred to reduce the amount of these operations in some way. Wong, Gouda and Lam presented in [5] and [6] techniques to reduce the number of digital signatures to one per set of rekeying messages, greatly improving performance, especially for user- and key-oriented rekeying.

## 2.4 Complexity Analysis

In table 2.1 the efficiency of a few of the most common encryption schemes are listed. The most important factors when evaluating a broadcast encryption scheme are broadcast message size, storage space at the receivers (sometimes also at broadcast center) and processing time.

The first trivial scheme simply encrypts one time for each privileged user. The user only needs to store one secret key. The second trivial scheme assumes the user has a key for every possible combination of users. The *k-resilient* scheme was first introduced by Fiat and Naor in 1993 and is secure from an attack from a coalition of k users (see [2]). The *Layered Subset Difference* scheme is an extension of the *Subset Difference* scheme and can handle very large user sets (see [4]).

| **Scheme** | **Message Size** | **Storage Size** | **Processing Time** |
|---|---|---|---|
| Trivial 1 | $O(N-r)$ | $O(1)$ | - |
| Trivial 2 | $O(1)$ | $O(2^N)$ | - |
| k-Resilient[2] | $k^2log^2(k)log(N)$ | $klog(k)log(N)$ | - |
| Complete Subtree | $O(rlog(\frac{N}{r}))$ | $O(log(N))$ | $O(log(log(N)))$ |
| Subset Difference | $O(r)$ | $O(\frac{1}{2}log^2(N))$ | $O(log(N))$ |
| LSD[4] | $O(r/\epsilon)$ | $O(log^{1+\epsilon}(N))$ | $O(log(N))$ |
| LKH (Key Tree) | $O(1)$ | $O(h)$ | $O(1)$ |
| NP[13] | $O(t)$ | $O(1)$ | $O(t)$ |
| CS+[14] | $O(rlog_a(\frac{N}{r})+r)$ | $O(1)$ | $O(2^alog_a(N))$ |
| SD+[14] | $O(r)$ | $O(1)$ | $O(N)$ |

Table 2.1: Complexities of different broadcast encryption schemes. $N$ is total number of users, $r$ is number of revoked users, $k$ is size of adversary coalition, $\epsilon$ is small positive value, $d$ is degree of key tree, $t$ is threshold of collusion, $a$ is arity of tree in Asano scheme

As seen in table 2.1, the Logical Key Hierarchy scheme has lower complexity than *CS* and *SD*. However, these complexities are only for the actual message broadcast and does not include the LKH rekeying messages. The average cost per join/leave operation for LKH when using individual rekeying in a key tree of degree $d$ is $O(d/(d-1))$ (see [5] for the theory behind this). If the set of authorized users is fairly static (the users have low *mobility*), LKH should give better performance. In a more dynamic case the number of rekeying messages will increase, thus increasing the overall complexity of LKH. The task to compare the effectiveness of the algorithms is much more complex in this case, see [11] and [8] for practical performance comparisons between stateless and stateful algorithms.

# Chapter 3

# System Overview

*This chapter describes the development from specification to implementation and gives an overview of the design and graphical user interface of the system. The chapter ends with a detailed description of how to extend the system.*

## 3.1 Introduction

The purpose of this project is to develop a framework in which different broadcast encryption algorithms can be implemented and tested. The system currently implements all the algorithms discussed in the previous chapter. It is an essential criterion for the system that it can be extended and handle a wide array of algorithms.

One of the requirements for this project was that the system should be implemented in Java. The entire project is implemented in a Java package named *beaf*, which stands for *Broadcast Encryption Algorithm Framework*.

The idea of this framework is to provide a class hierarchy that can be used as a basis to develop new algorithms. In fact, to add a new algorithm to the system the developer only needs to implement two classes. The framework also provides an application environment with a graphical user interface. Through this interface the developer is presented with the basic functionality needed to interact with the algorithms.

## 3.2 Requirements

In the beginning, the purpose of this project was to create a framework for implementing algorithms and perhaps interacting with the algorithms. This did not seem adequate since it did not give much detailed information about the algorithms. The most interesting aspects of a broadcast encryption algorithm are its security, efficiency, resource requirements, and limitations. The system should therefore provide feedback on these properties.

All the algorithms share the concept of multiple users that may or may not be privileged. Handling these users and defining which ones are privileged has also been made an integral part of the system.

The following is a list of the requirements that the system was designed for:

- It should be possible to simulate a broadcasting environment with a broadcasting center and receivers.
- It should support a graphical representation of the algorithms.
- It should be possible to define which users are privileged and which ones are revoked.
- It should be possible to dynamically modify the parameters for the algorithms.
- It should be possible to gather statistical information on the algorithms.
- It should be possible to display the statistical information in a graph.
- It should be possible to run simulations on the algorithms with predefined input.
- It should contain implementations of the Complete Subtree, Subset Difference, and Logical Key Hierarchy algorithms.

## 3.3 Design

In order to support a broad array of algorithms the system must be very general. At the top of the hierarchy is the base class for all broadcast encryption algorithms, the *BroadcastEncryptionAlgorithm* class. This class is subclassed to specialize the behavior of the algorithm. As is shown in figure 3.1, the class *BinaryTreeAlgorithm* is a subclass to *BroadcastEncryptionAlgorithm*, and holds some common functionality for the classes *CompleteSubtreeAlgorithm* and *SubsetDifferenceAlgorithm*.

The server and client side versions of the algorithms are implemented in different classes to better separate the encryption and decryption functionality. The client side functionality of the algorithms are all implemented in subclasses of *BroadcastDecryptionAlgorithm*, see figure 3.2.

The framework uses polymorphism to easily handle the different algorithms. This is a necessary design decision since all the algorithms must be run in the same

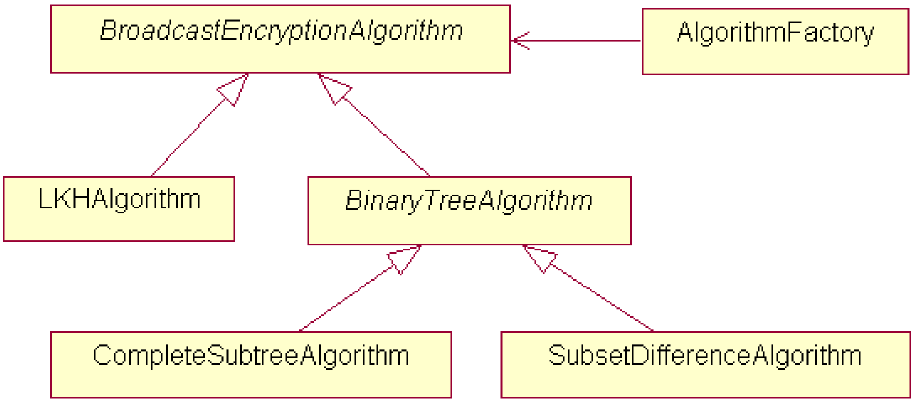


Figure 3.1: The structure of the algorithm classes, server side.

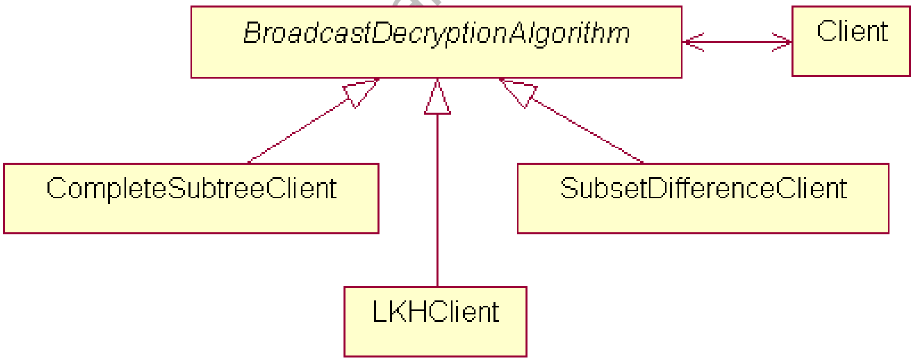


Figure 3.2: The structure of the algorithm classes, client side.

environment. This is what makes the system flexible: Adding a new algorithm does not change the existing framework.

To provide algorithm-specific user interfaces, the framework uses a document-view model between the algorithm and it's user interface. This is done by allowing the algorithms to subclass the *AlgorithmPanel* class and implement the specific user interface functionality to it, see figure 3.3. The *AlgorithmFactory* handles the coupling of the algorithm and it's panel, to allow for easy construction of both objects.

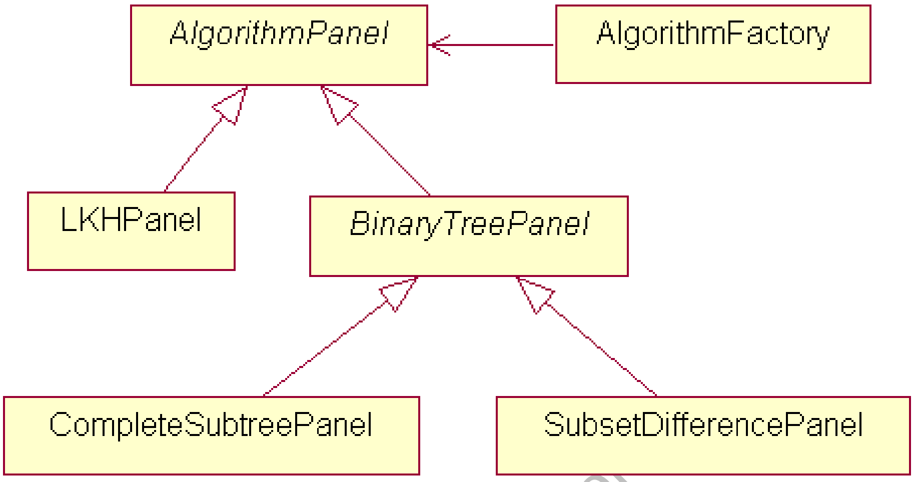


Figure 3.3: The structure of the algorithm user interface classes.

The broadcasting scenario implies a server/client architecture. The main application is simulating the broadcast center, or server side of the connection. The client application may also be run to simulate a receiver, although this is not necessary for the main application to function.

### 3.3.1 Notable Classes

In this section a few of the most important classes are presented and discussed. A more detailed description of the existing classes are provided in appendix D.

#### BroadcastEncryptionAlgorithm

This is the base class for all algorithms. It declares the functions that must be implemented in its descendants. For instance, algorithms must implement the *initialize* method, which is called to initialize the algorithm and send essential information to the clients, before sending any broadcasts. The algorithms must also implement *broadcast*, which sends a broadcast message to all clients.

#### BroadcastDecryptionAlgorithm

This base class is used for implementing the client side version of an algorithm, and is used by the *Client* class for decrypting the messages sent by the server.

#### AlgorithmPanel

This is the base class for representing the algorithms in a graphical interface. By subclassing this class, an algorithm can show algorithm specific information and allow user interaction with the algorithm.

#### AlgorithmFactory

This is a factory class for generating an instance of an algorithm, together with the corresponding AlgorithmPanel.

#### Server and Client

The *Server* and *Client* classes use TCP/IP sockets to establish direct communication. All connected clients are registered at the server, so that when an algorithm does a broadcast, it is automatically sent to all connected clients.

#### Statistics

The *Statistics* class is used for storing generated performance data (statistics) from algorithms and simulations. One example of data that is being generated and reported by the CS and SD algorithms is the number of subsets for a specific user configuration. These statistics can later be used for complexity analysis.

The application contains one instance of this class, and the data reported to this object is presented in the statistics panel and chart, see appendix B.

#### Simulation

The *Simulation* class has several methods for performing different types of simulations on the algorithms. This includes a *batch file* mode that is capable of performing a series of simulations with given parameters. See appendix C for a description of the batch files.

### 3.3.2 Cryptography

This application uses the standard Java packages *java.security* and *javax.crypto* for encryptions and other cryptographic operations. In addition to the standard Java

API documentation, see [12] for a good overview of these packages and examples on how to use them.

These standard packages lack some encryption algorithms, for example AES, but the existing algorithms (DES and 3DES among others) were considered enough for this system. If other algorithms are required in the future, some external package supporting those algorithms will have to be provided.

## 3.4 System Extension

This section describes how some of the probable extensions of the system should be done.

### 3.4.1 Adding an Algorithm

The system can easily be extended with new algorithms. To do this, the following steps should be performed:

1. Subclass *BroadcastEncryptionAlgorithm* and implement the server side functionality of the algorithm.

2. Subclass *AlgorithmPanel* and implement the algorithm specific user interface (this is an optional step).

3. Modify *AlgorithmFactory* to be able to create instances of the above classes.

4. Subclass *BroadcastDecryptionAlgorithm* and implement the client side functionality of the algorithm (if decryption capability is wanted).

5. Modify *Client* to be able to handle the new decryption algorithm.

Below is a more detailed description of how to do the above steps. All created classes should be put in a specific package for that algorithm, like *beaf.lkh*.

#### Subclass BroadcastEncryptionAlgorithm

This is the major part of the implementation of an algorithm. Subclass *BroadcastEncryptionAlgorithm* and implement the server-side functionality of the algorithm. The provided interface includes retrieval and setting of algorithm parameters, updating the algorithm with new user configurations, and methods for server-client communication. See the Javadoc for details on what methods to implement, and functional descriptions of those methods.

### Subclass AlgorithmPanel

If any type of algorithm specific in- or output is wanted in the user interface, subclass *AlgorithmPanel* and implement the interface to it. The corresponding algorithm object will be provided during construction to allow the panel to communicate with the algorithm directly.

### Modify AlgorithmFactory

Modify the *AlgorithmFactory* class to be able to create objects of the above algorithm and panel classes. The methods to modify are *getInstance* and *getAvailableAlgorithms*.

### Subclass BroadcastDecryptionAlgorithm

Subclass the *BroadcastDecryptionAlgorithm* class and implement the client-side behaviour of the algorithm. See the Javadoc for details on what methods have to be implemented. In short, the algorithm will receive control and broadcast messages sent by the server-side algorithm, and it is up to this class to process the algorithm specific messages (server-client control messages are automatically filtered and not visible to this class).

### Modify Client

Modify the algorithm allocation code in the *process* method of the *Client* class to be able to allocate the newly created BroadcastDecryptionAlgorithm object.

## 3.4.2 Adding a Simulation

Additional simulation types can be added to the system. To do this, implement new methods (and corresponding working thread) in the *Simulation* class. Please see the existing *random* method and its working thread (*RandomThread*) for details.

# 3.5 External Dependencies

The system uses two non-standard libraries to improve the usability and reduce the development time. These packages are:

**JFreeChart** [16], a library for chart drawing. This is used by the *StatisticsChartFrame* class to draw the actual chart. It can also generate PNG-image output of the charts.

**EPSGraphics** [17], a library for generating EPS output from a subclassed *Graphics2D*-object. This is used by the *StatisticsChartFrame* class in conjunction with the JFreeChart classes to generate EPS output of the charts.

Both of these libraries are published under the GNU General Public Licence.

# Chapter 4

# Results

*This chapter shows how the final program looks and a few examples of simulation output created by the program. Also the major limitations and advantages of this program is discussed.*

## 4.1 System

The program (see figure 4.1) looks like a typical windows application. From the menu it is possible to choose which algorithm to display, the size of the algorithm and other parameters. A simulation can also be started from the menu on the current algorithm or from a file. This will generate statistics data which can be viewed in a special chart window.

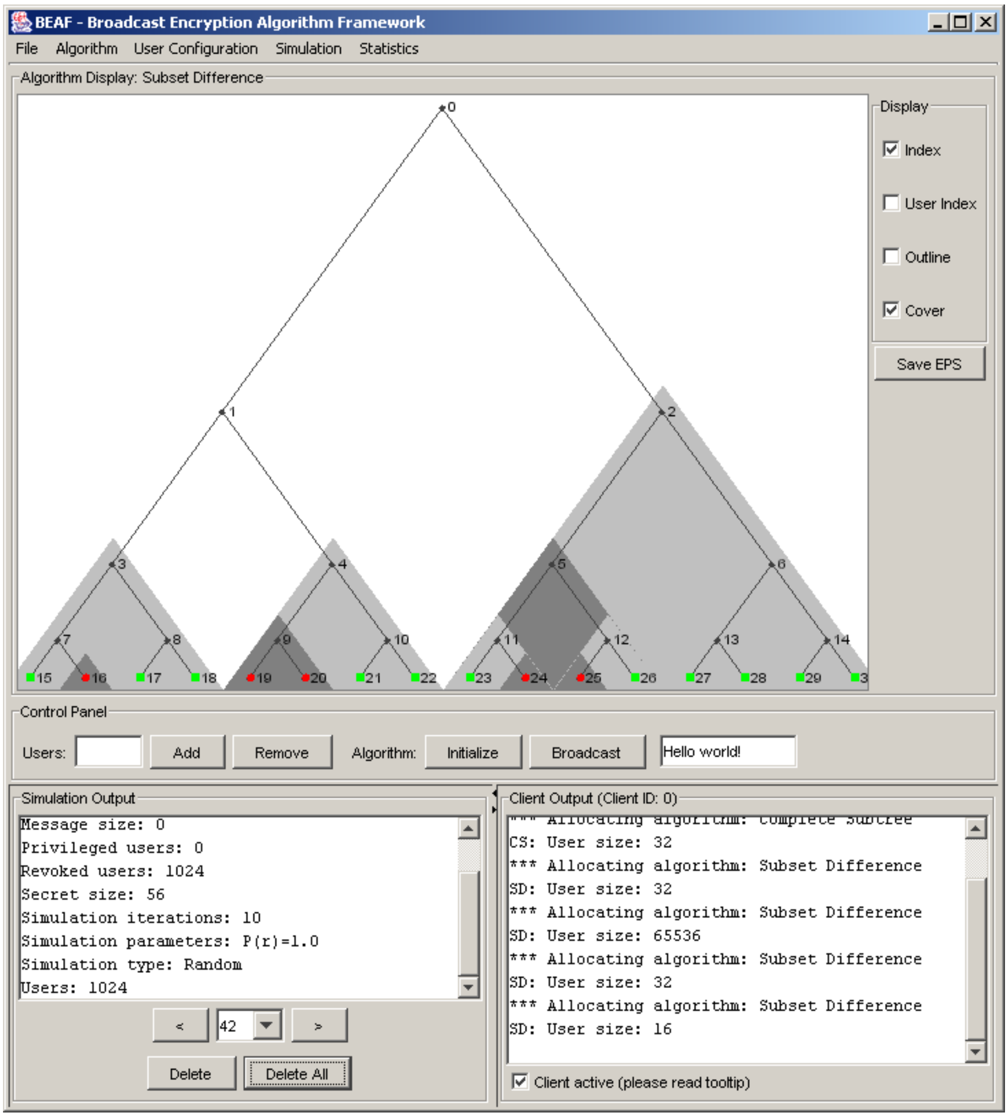
In the main GUI of the application the algorithm is displayed and several buttons to interact with the algorithm are available depending on the algorithm chosen. The general usage of a broadcasting algorithm is to broadcast data to its clients and this can be done by clicking the broadcast button. A message will then be shown in the client window telling the user if the message was received and decrypted correctly.

The algorithms that are implemented allows the user to interact with them. For instance, it is possible to add/remove users to/from the algorithm and see how the algorithm updates it's structures.

See appendix B for a more detailed description of the user interface and program functionality.

## 4.2 Simulation

A few examples of output generated by the program are shown in figures 4.2, 4.3, 4.4, 4.5 and 4.6. They are generated by executing a broadcast scenario in which



Hello world!

```
Message size: 0
Privileged users: 0
Revoked users: 1024
Secret size: 56
Simulation iterations: 10
Simulation parameters: P(r)=1.0
Simulation type: Random
Users: 1024
```

```
*** Allocating algorithm: Complete Subtree
CS: User size: 32
*** Allocating algorithm: Subset Difference
SD: User size: 32
*** Allocating algorithm: Subset Difference
SD: User size: 65536
*** Allocating algorithm: Subset Difference
SD: User size: 32
*** Allocating algorithm: Subset Difference
SD: User size: 16
```

☑ Client active (please read tooltip)

Figure 4.1: The main GUI of the system.

the broadcast center sends messages to one receiver. During this process statistics about the algorithm is gathered.

In figure 4.2 the cover sizes for the *Complete Subtree* algorithm and the *Subset Difference* algorithm are compared. In these algorithms this is the same as the message header size. The message size should be $O(rlog(\frac{N}{r}))$ for the *Complete Subtree* algorithm and $O(r)$ for the *Subset Difference* algorithm (see table 2.1). Since $\frac{N}{r} = 10$ in this example the only difference between the message sizes is a constant factor.

By averaging the cover sizes (for user sizes from 128 to 16384) in figure 4.2 for the *Subset Difference* algorithm an expected message size of about $1.06r$ is obtained. This is small compared to the average $1.38r$ suggested in [3]. However this average is quite dependant on the scenario and the user sizes tested.

In figure 4.3 the secret sizes (or number of keys/labels at the receivers) are plotted. The *Subset Difference* algorithm has a greater secret size which is also seen in table 2.1.

The *Subset Difference* algorithm always performs better than the *Complete Subtree* algorithm in terms of cover size, as can be seen in figure 4.4. In this example the user size is set to 1024 and the number of revoked users is increased for each simulation.

Figures 4.5 and 4.6 show some results of the LKH algorithm. Figure 4.5 shows the average number of rekey messages sent per update, for a span of user sizes. As expected (see [5]) the group-oriented rekeying constantly produces one rekey message per update, while the user- and key-oriented strategies generate a larger number of messages.

Figure 4.6 displays the number of encryptions made per update. Since the user-oriented strategy encrypts the updated keys with the individual keys, the number of encryptions are significantly larger compared to the key- and group-oriented strategies. Since the key- and group-oriented strategies basically function in the same way, except for the construction of the rekey message, they have an equal number of encryptions per update. This is again correct according to [5].

## 4.3 Limitations

The system has some limitations as explained below:

**Algorithm Size.** All the algorithms have upper limits to the number of users they can contain, depending on how much memory is available. On the testing machine the limits for CS were $2^{18}$ users, for SD $2^{17}$ users and for LKH approximately $2^{14}$ users.

**Performance.** Since the system is written in Java, performance may not be as

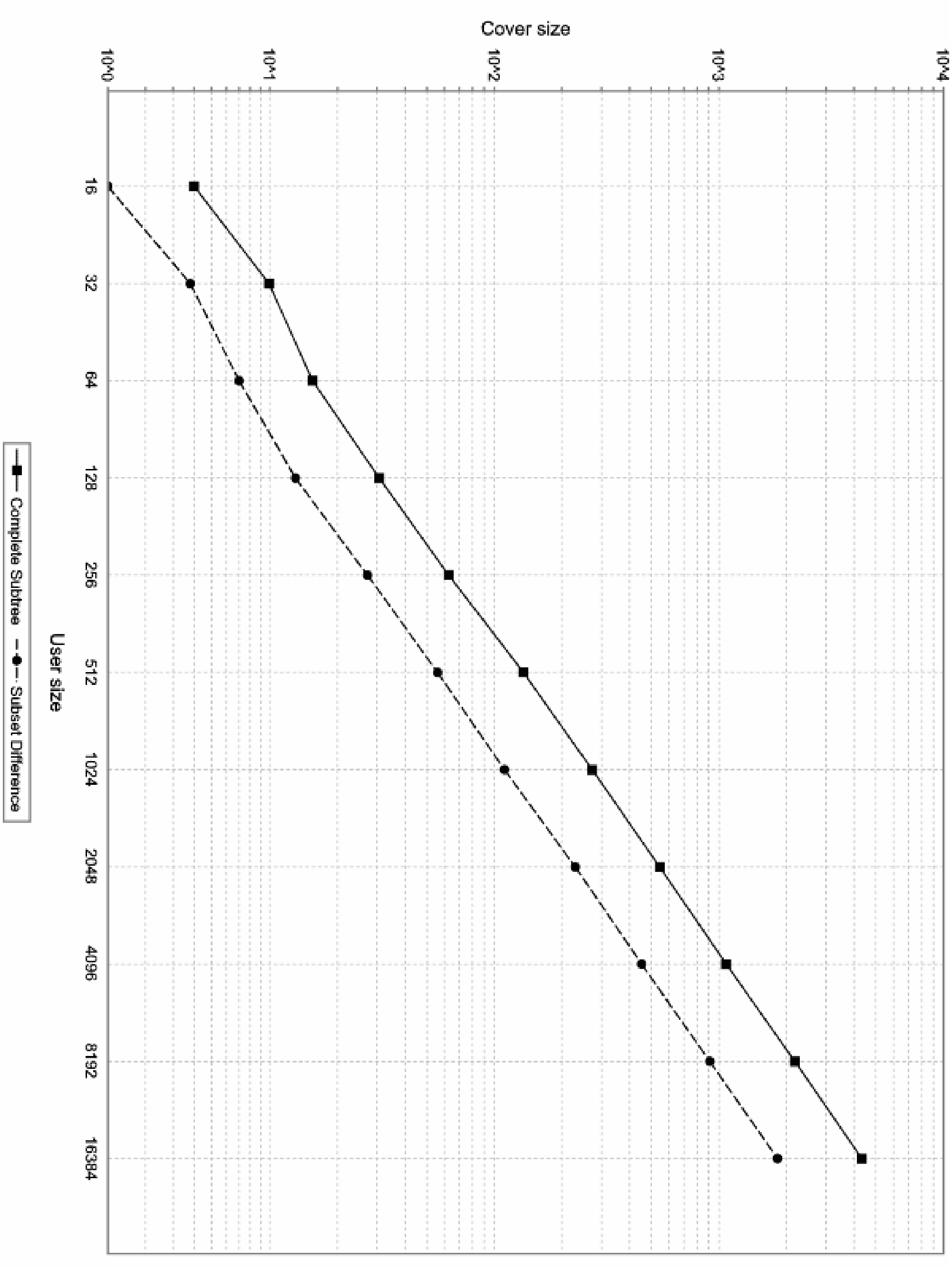


Figure 4.2: Comparison between *Complete Subtree* and *Subset Difference* and the generated cover size for different user sizes, where 10% of the users are revoked (using a random spread).

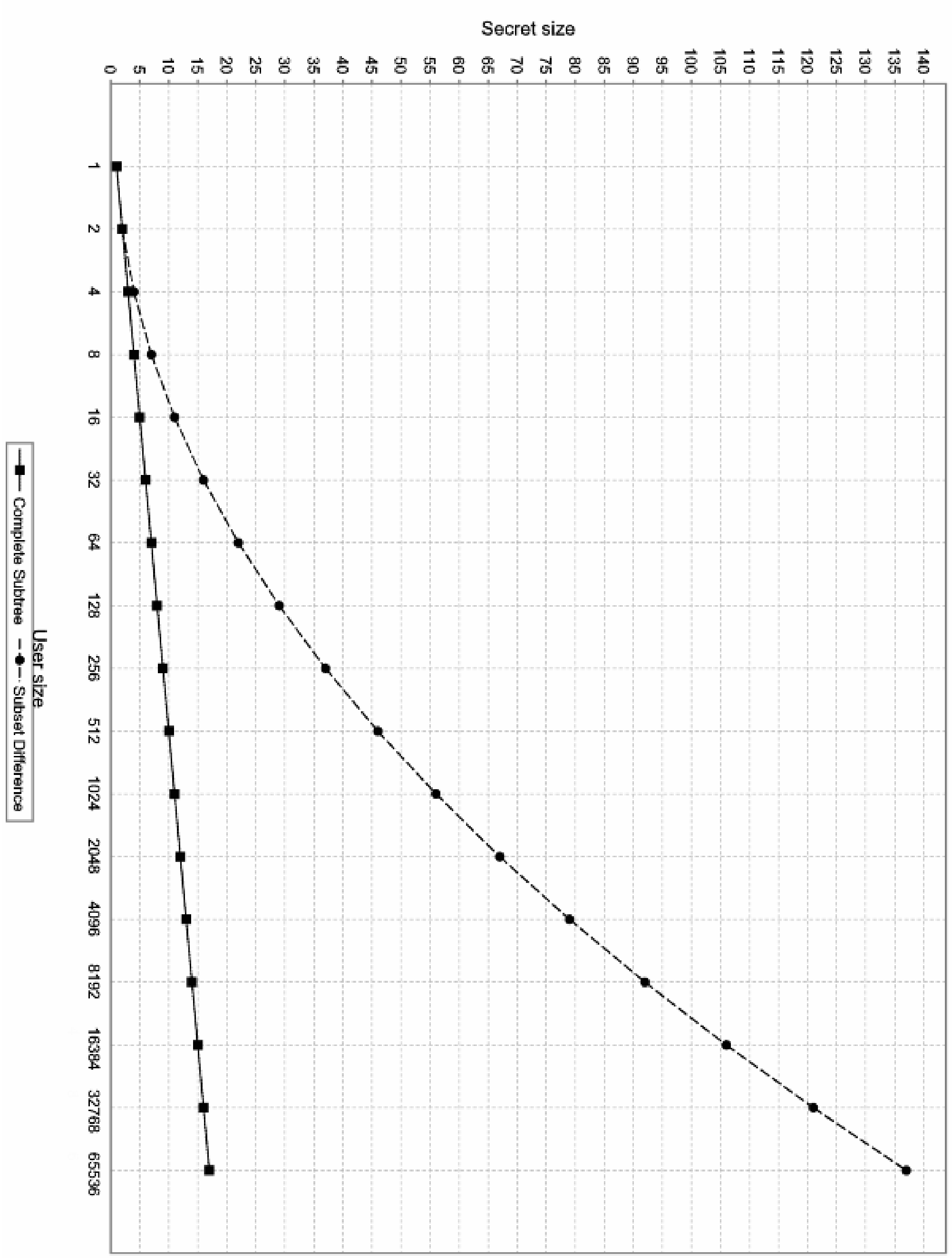


Figure 4.3: Comparison between *Complete Subtree* and *Subset Difference* and the secret size (storage size at receiver) for different user sizes. (This size is constant for each user size.)

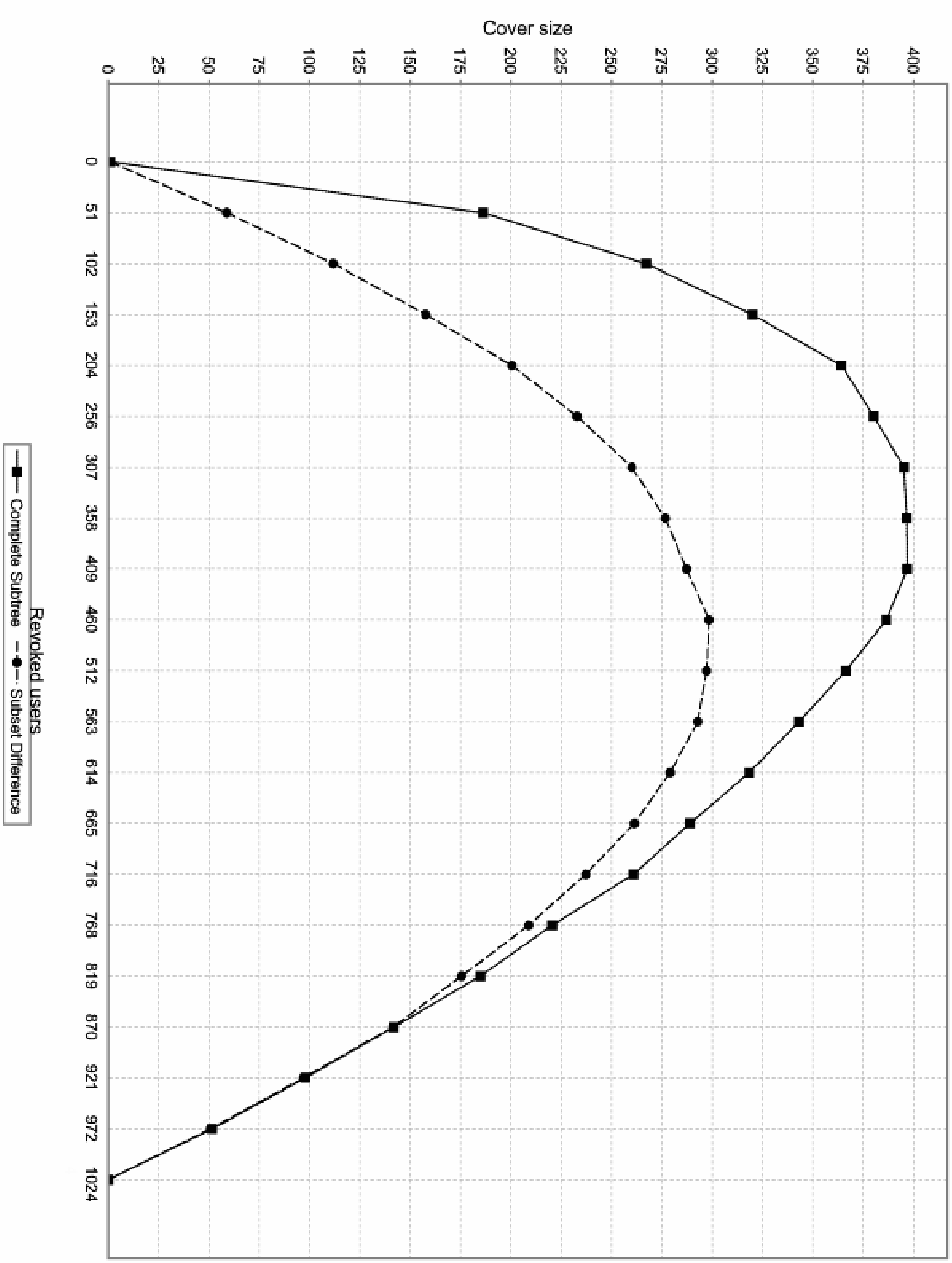


Figure 4.4: Comparison between *Complete Subtree* and *Subset Difference* and the generated cover size for different number of revoked users. The maximum number of revoked users is 1024.

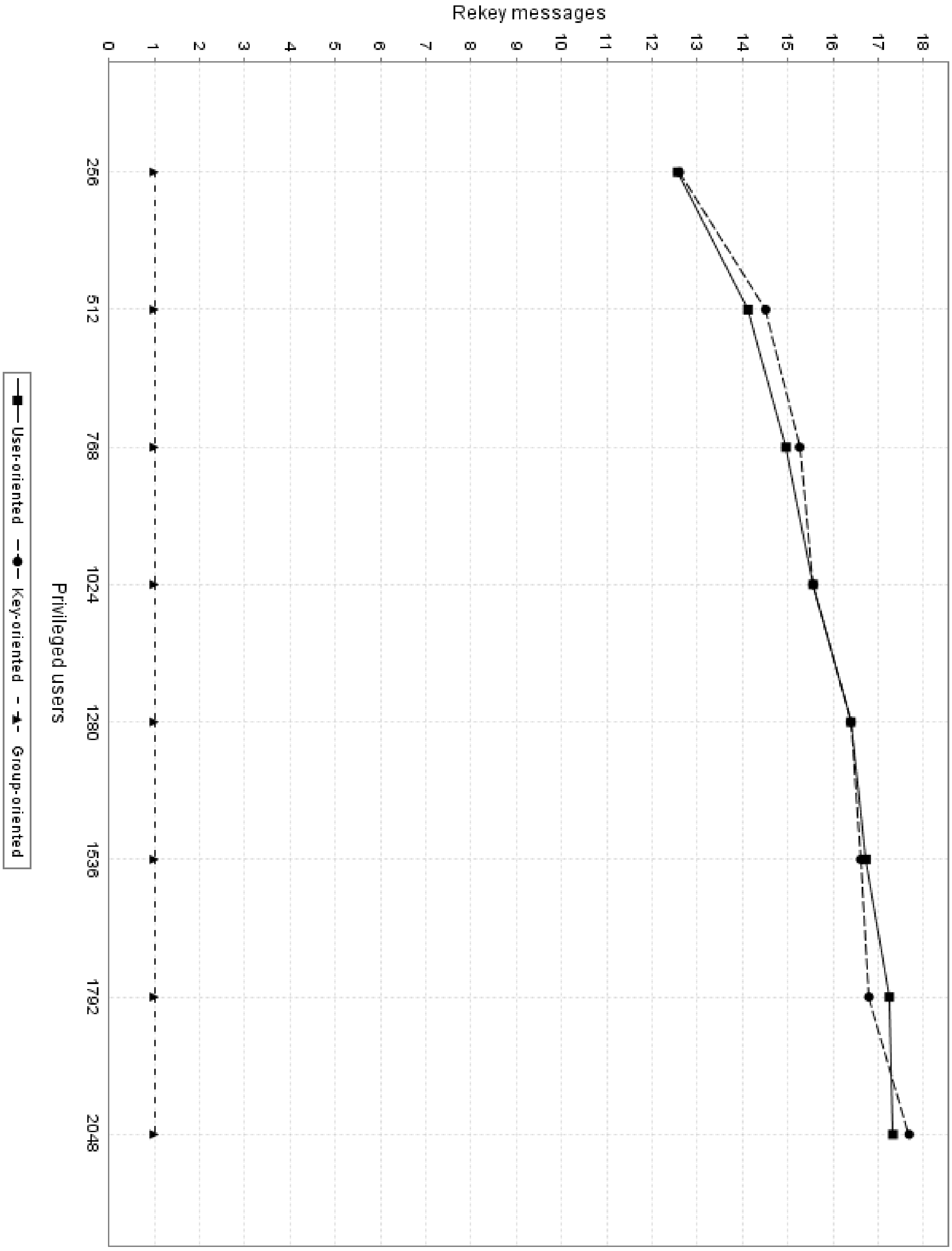


Figure 4.5: Comparison of the number of rekey messages per join/leave for different LKH rekey strategies. Key tree degree is 4.

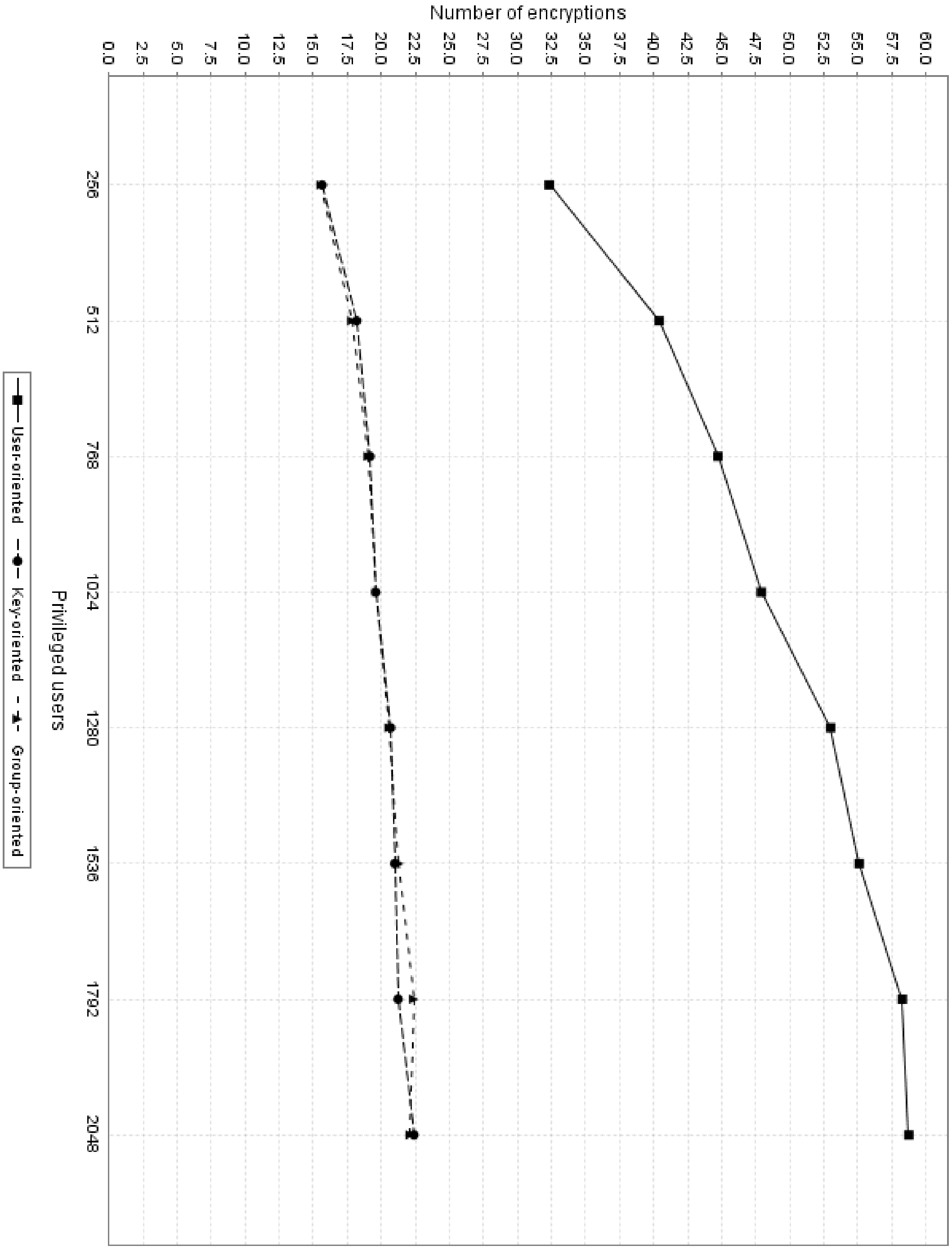


Figure 4.6: Comparison of the number of encryptions per join/leave for different LKH rekey strategies. Key tree degree is 4.

good as had it been written in another language. The main problem is the fact that while the Java garbage collection mechanism is convenient, it sometimes slows down program execution at random occations.

This is especially true when running simulations on the algorithms which allocate and deallocate large amounts of memory. To decrease the number of deallocations the algorithms attempt to reuse as many objects as possible.

**Algorithm Timing.** This is another, more specific, problem with the existing Java version. The resolution of the Java time measurement function *System.currentTimeMillis* is insufficient. The reason is that the time resolution is dependent on the Java Virtual Machine and/or the operating system. In some cases the resolution of *System.currentTimeMillis* is as low as 50 milliseconds (see [15]).

This means that the algorithm processing time measurement might be very unreliable. On some systems all measured times below 50 milliseconds will be truncated to 0, thus giving close to unusable results in some cases. This problem will hopefully be remedied in the next version of Java (1.5), since that version will introduce time measurement functions with resolution in nanoseconds.

**Simulated Broadcasting.** The client-server architecture does not currently use true broadcasting or multicasting, but instead uses a single socket connection (unicast) to each user. This implies that it is not currently meaningful to gather network traffic statistics to use for any type of analysis. This will also reduce system and network performance if a large number of clients are connected simultaneously.

**LKH Algorithm Functionality.** The current implementation of LKH lacks one detail. The possibility to do a single signature per set of rekey messages is currently not implemented. Instead all rekey messages are signed individually.

For user- and key-oriented rekeying it might therefore be desirable to completely disable the signing of rekey messages, to lessen the impact of this flaw. This is done by selecting *None* as signature algorithm in the LKH parameters. Since group-oriented rekeying only sends a single rekey message (and thus only requires one signature), this flaw can be ignored for the cases where this rekeying strategy is used.

## 4.4 Advantages

Some advantages of this system are described below:

**Algorithm Comparison.** The possibility to execute simulations on different algorithms in the same framework, and show the result in a common chart, can be used to do performance comparisons of algorithms that are too different to allow for easy theoretical comparisons (for example the CS algorithm compared to LKH). This is the main advantage of this system.

**Extension.** The framework makes it easy to implement additional broadcast encryption algorithms. Only a few classes have to be implemented and modified when adding a new algorithm, see section 3.4.

**Algorithm Support.** The framework does not put much restrictions on the algorithms so most types of algorithms can be implemented. Both stateless and stateful algorithms can be added, as demonstrated by the currently implemented algorithms.

**Visualization.** The graphical interface makes it possible to visualize the algorithms in real-time to better understand how they work. An example of this is the possibility to see how the subsets are structured in the Complete Subtree and Subset Difference algorithms.

**Client/Server Architecture.** The client/server architecture used by the system is very similar to a real broadcasting environment, where data is actually being sent over a network. This means that the simulation results should be similar to a real-life scenario.

**Multiple Platform Support.** Because the system is written completely in Java it can be executed on any platform that implements a Java Virtual Machine.

# Chapter 5

# Summary

*This chapter contains a brief evaluation of the results presented in this report and summarizes the application areas and benefits of this system.*

## 5.1 Evaluation

The focus of this project has for the most part been to develop a good environment for testing broadcasting algorithms. The results in chapter 4 show that the theory behind the algorithms is correct and that this system can be used to gather other theoretical results about the algorithms.

The requirements for this system that were stated during the start of this project (see section 3.2) have all been fulfilled and the program is complete in this aspect.

## 5.2 Applications

The primary use of this system is to test algorithm performance by running a series of simulations to emulate real usage scenarios. By doing so for more than one algorithm, the algorithms' performance can be compared and visualized using graphs. Besides running simulations, the algorithms can also be experimented with in real-time through the algorithms' own graphical user interface. This might be used for demonstrational purposes.

## 5.3 Future Work

Below are a few suggestions for how the system can be improved:

**Additional Algorithms.** Additional algorithms could be implemented, tested and compared to the other algorithms already in the system.

**Additional Simulations.** Additional simulation types could be implemented to allow simulations of more real-life scenarios, where the user set varies in a certain way. See [8] for some examples of possible simulation types.

**Real Broadcasting.** To better simulate network conditions, the server could be rewritten so that it uses real broadcasting when communicating with the clients. This would allow for accurate measurements of the real network loads generated by the different algorithms.

# References

[1] Shimson Berkovits: *How to broadcast a secret.* Advances in Cryptology: EUROCRYPT '91, pages 536-541. Springer-Verlag, 1992.

[2] Amos Fiat, Moni Naor: *Broadcast encryption.* Advances in Cryptology: CRYPTO'93, LNCS 773, pages 480-491. Springer-Verlag, 1994.

[3] Dalit Naor, Moni Naor, Jeff Lotspiech: *Revocation and Tracing Schemes for Stateless Receivers.* Advances in Cryptology - CRYPTO '01, volume 2139 of Lecture Notes in Computer Science, pages 41-62. Springer Verlag, 2001.

[4] Dani Halevy, Adi Shamir: *The LSD Broadcast Encryption Scheme*, The Weizmann Institute of Science, 2002.

[5] Chung Kei Wong, Mohamed Gouda, Simon S. Lam: *Secure Group Communications Using Key Graphs.* Technical Report TR-97-23, Department of Computer Sciences, The University of Texas at Austin, 1997.

[6] C. K. Wong and Simon S. Lam: *Keystone: A group key management service.* Proceedings of the International Conference on Telecommunications, 2000.

[7] X. Steve Li, Y. Richard Yang, Mohamed G. Gouda, and Simon S. Lam: *Batch rekeying for secure group communications.* Proceedings of Tenth International World Wide Web Conference (WWW10), Hong Kong, China, May 2001.

[8] Mattias Johansson: *Practical Evaluation of Revocation Schemes.* Master's Thesis in Computer Science, TRITA-NA-E04058, Royal Institute of Technology, Sweden, 2004.

[9] Jeffrey Lotspiech, Stefan Nusser, Florian Pestoni: *Broadcast Encryption's Bright Future.* IEEE Computer vol. 35, pages 57-63, 2002.

[10] E. W. Weisstein. "Steiner Tree". MathWorld - A Wolfram Web Resource. URL: http://mathworld.wolfram.com/SteinerTree.html

[11] W. Chen, L. R. Dondeti: *Performance comparison of stateful and stateless group rekeying algorithms*. Proceedings of the Fourth International Workshop on Networked Group Communication - NGC '02, October 2002.

[12] Nick Galbreath: *Cryptography for Internet and Database Applications - Developing Secret and Public Key Techniques with Java*. ISBN 0-471-21029-3, Wiley Publishing Inc., 2002.

[13] M. Naor and B. Pinkas: *Effcient Trace and Revoke Schemes*. Financial Cryptography FC 2000, LNCS 1962, pages 1-20.

[14] T. Asano: *A Revocation Scheme with Minimal Storage at Receivers*. ASIACRYPT 2002, LNCS 2501, pages 433-450.

[15] Vladimir Roubtsov: *My kingdom for a good timer! Reach submillisecond timing precision in Java*. JavaWorld, January 2003. URL: http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html

[16] *JFreeChart*. URL: http://www.jfree.org/jfreechart/

[17] *EPSGraphics*. URL: http://www.jibble.org/epsgraphics/

# Appendix A

# Glossary

**Batch Rekeying** An algorithm for updating the LKH key graph with several join/leave requests before sending rekey messages. More efficient than individual rekeying.

**Body** The part of a broadcast message that contains the encrypted message.

**Broadcast** Distributing encrypted/unencrypted data through a public or shared media.

**Client** The receiver in a broadcast scenario.

**Cover** A set of subsets in a subset-cover algorithm.

**CS** The Complete Subset algorithm.

**DES** Data Encryption Standard. A 56-bit block cipher.

**Header** The part of a broadcast message that contains key and access rights information.

**Individual Key** A k-node in a key graph which is only known to a specific user and the server. Every user in the key graph has an individual key.

**Individual Rekeying** When the LKH algorithm sends rekeying messages after each join/leave request. Less efficient than batch rekeying.

**Intermediate Label** A label that is used by the SD algorithm.

**k-node** A node in a key graph that represents an encryption key.

**Key Graph** The graph that contains the encryption keys for the LKH algorithm.

**Key Tree** A key graph that is constructed as a tree.

**Label** A bitset that is used to generate keys in the SD algorithm.

**LKH** The Logical Key Hierarchy algorithm.

**LSD** The Layered Subset Difference algorithm.

**Polymorphism** A design technique which hides the implementation of a component behind a generic interface.

**Privileged** Access right for users that are permitted to access broadcast message.

**Rekeying Message** A message sent by the LKH algorithm when the key graph has been updated, to update the clients' encryption keys.

**Rekeying Strategy** A rekeying strategy is used by the LKH algorithm to decide how to construct and transmit the rekeying messages.

**Revoked** Access right for users that do not have access to broadcast message.

**SD** The Subset Difference algorithm.

**Secret Information** Secure information in the receiver (usually key material).

**Server** The sender in a broadcast scenario. Also called broadcast center.

**Session Key** The key used to encrypt the broadcast message.

**Smartcard** A Java-enabled memory chip that protects confidentiality of content.

**Stateful** Opposite of stateless.

**Stateless** Characteristic of an algorithm that does not need to update its secret information when the access rights are updated.

**Steiner tree** A subset of a tree graph which also is a tree.

**Subset** A set of privileged users.

**u-node** A node in a key graph that represents a user.

# Appendix B

# User Manual

*This appendix describes the user interface and system functionality in more detail.*

## B.1 Server

The user interface is divided into five functional parts: the *menu bar*, and the *algorithm*, *control*, *statistics* and *client* panels (see figure B.1).

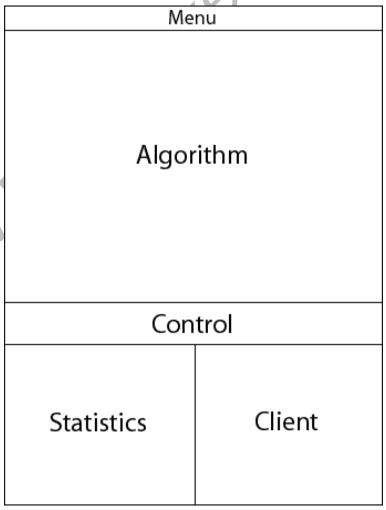


Figure B.1: Functional layout of the server GUI.

### B.1.1 Menu Bar

The menu bar controls the general functionality of the application: Changing algorithm, setting algorithm parameters, storing and loading of user configurations and executing simulations.

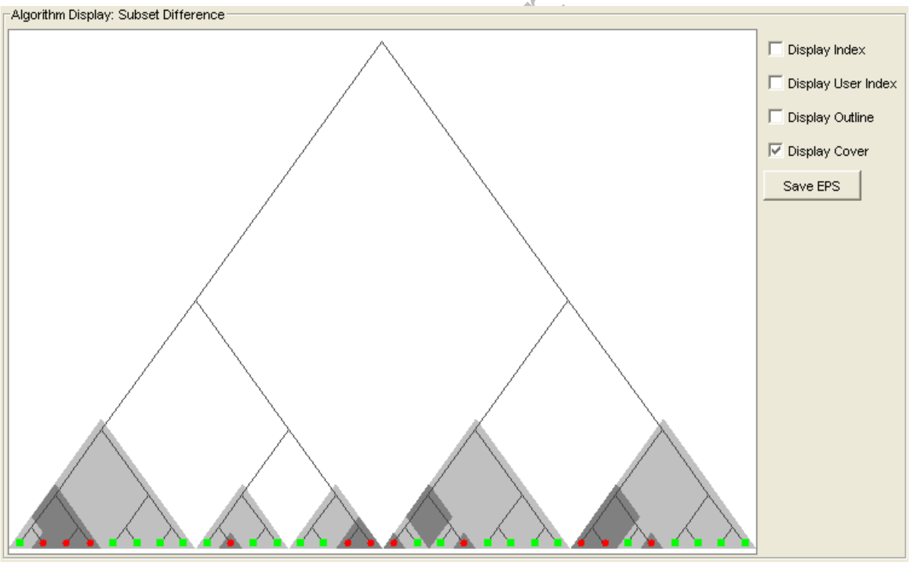
The *Algorithm* menu is used to change the current algorithm, setting the number of users, and changing algorithm parameters. The (x y) prefix, where x and y are integers, in the parameters submenu are used when setting a parameter in a batch file, see sections C.3.4 and C.3.5.

The *User Configuration* menu is used for generating random user configurations, and for loading and storing the current configuration. This can be used for commonly used configurations, to avoid having to set the configuration manually every time it should be used.

The *Simulation* menu is used for executing simulations, see section B.3 for information about the individual simulation types.

### B.1.2 Algorithm Display

The next part of the user interface is the *algorithm display*, used by the algorithms to present algorithm-specific information. Figure B.2 shows an example of a existing algorithm UI (in this case for the Subset Difference algorithm). Each algorithm can have a unique user interface, and we leave out the details.



Algorithm Display: Subset Difference

□ Display Index

□ Display User Index

□ Display Outline

☑ Display Cover

Figure B.2: The user interface for the Subset Difference algorithm.

### B.1.3 Control Panel

Below the algorithm frame is the *control panel* (figure B.3). This contains general algorithm controls that are applicable to most algorithms. This includes adding and removing authorized users, and forcing the application to initialize or broadcast with the current algorithm state.

The *Add* and *Remove* buttons are used in conjunction with the *Users* text field to the left of them, to add and remove users from the current privileged set. To do this, enter a string describing what user(s) to add or remove in the form *"5, -3, 8-12, 20-"*, and click Add or Remove. Any number of combinations of these are allowed, for example *"1-5, 10-15, 20-25, 8, 9"*.

The *Initialize* button forces the current algorithm to re-initialize with the current user configuration. This will rebuild the internal data of most algorithms. This function can be used for testing purposes, but is normally not used. Any client that connects to the server will automatically receive initialization data without the user having to do this manually.

The *Broadcast* button will broadcast the string entered in the text field to the right of it. The framework does this by calling the *BroadcastEncryptionAlgorithm.broadcast* method.

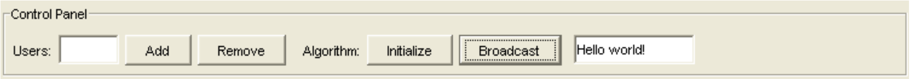


Figure B.3: The control panel.

### B.1.4 Statistics Panel

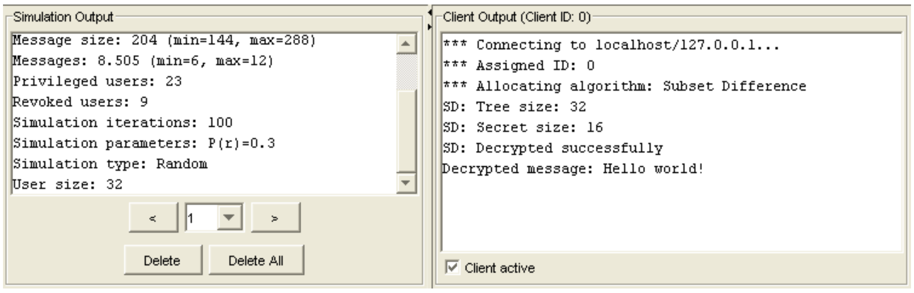
At the bottom of the user interface are the *statistics* and *client* panels, see figure B.4. The statistics panel is used for displaying the statistics generated by the simulations and algorithms.

The "arrow" buttons and the number list can be used to look on specific simulation outputs, and the *Delete* and *Delete All* is used to delete the current or all existing statistics.

### B.1.5 Client Panel

The client panel (figure B.4) displays the output of a client that is run locally as a separate thread. This client can be disabled at will, for example when it is preferred that the server has full access to the available processing power.

The output displayed in this panel is the same as would be from a client executed separately (see section B.1.6), except that the algorithm timing statistics reported might be somewhat higher, due to sharing the processing power with the server application (compared to running a client on a separate machine). This is important to remember when accurate algorithm timing is wanted, and the local client should be disabled with the checkbox at the bottom of the panel on these occations.



```
Message size: 204 (min=144, max=288)
Messages: 8.505 (min=6, max=12)
Privileged users: 23
Revoked users: 9
Simulation iterations: 100
Simulation parameters: P(r)=0.3
Simulation type: Random
User size: 32
```

```
*** Connecting to localhost/127.0.0.1...
*** Assigned ID: 0
*** Allocating algorithm: Subset Difference
SD: Tree size: 32
SD: Secret size: 16
SD: Decrypted successfully
Decrypted message: Hello world!
```

☑ Client active

Figure B.4: The statistics (left) and client (right) panels.

### B.1.6 Client

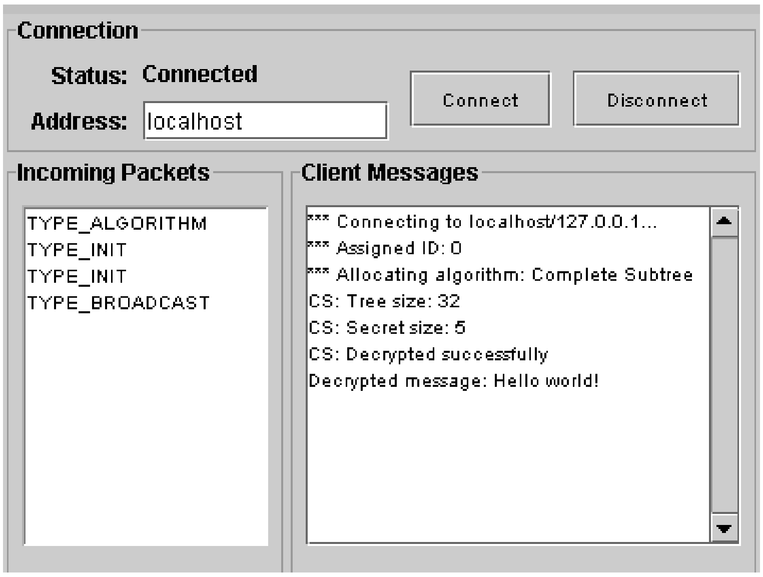
The client is, apart from the locally run client in the main application, available in two versions: as an applet (figure B.5), and as a command line version. Both versions displays nearly the same output, the difference is that the applet version also shows what types of packets are received, and not just the output from the algorithm itself.

The command line version takes the address of the server as an argument, while the applet version has a text box available for this.

## B.2 Statistics Chart

The *statistics chart* (figure B.6) can be accessed from the main menu. This is a separate window that can draw charts from the collected statistics data. Several options are available for selection of what should be drawn, and for some customizations of the chart.

(the chart UI will be changed soon, detailed description of the statistics chart will be added after that is done)



```
TYPE_ALGORITHM
TYPE_INIT
TYPE_INIT
TYPE_BROADCAST
```

```
*** Connecting to localhost/127.0.0.1...
*** Assigned ID: 0
*** Allocating algorithm: Complete Subtree
CS: Tree size: 32
CS: Secret size: 5
CS: Decrypted successfully
Decrypted message: Hello world!
```

Figure B.5: The client applet user interface.

## B.3 Simulations

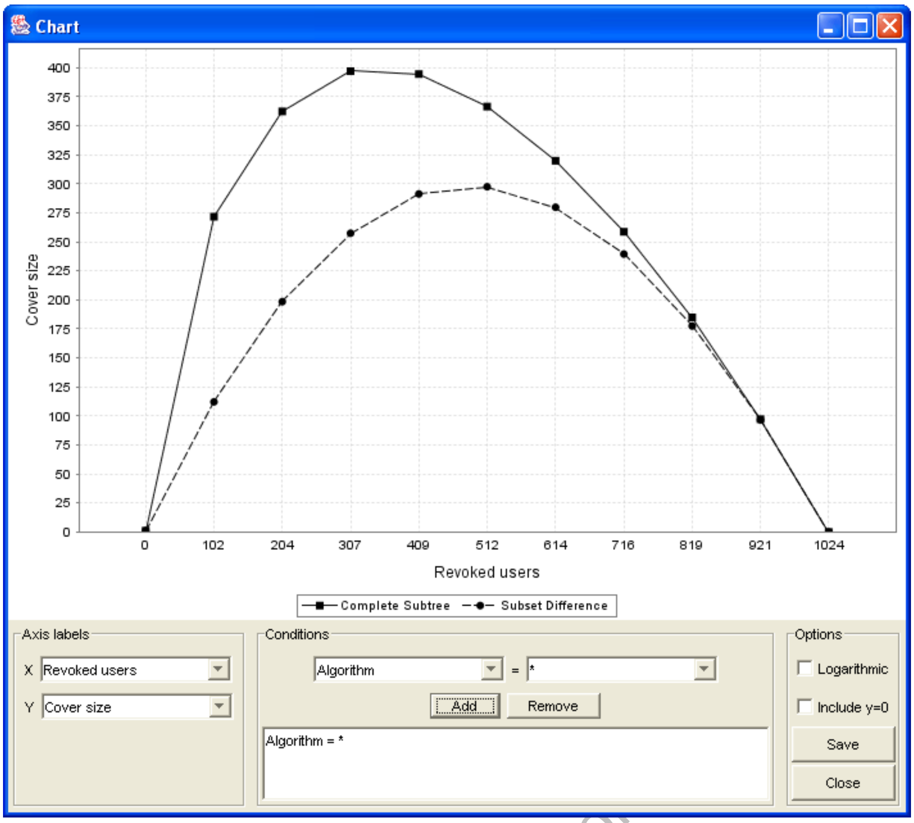
The application can perform different types of simulations on the algorithms. This can be done either by using the *Simulation* menu (see section B.1.1), or executing a *batch file*. A batch file is simply a text file describing what simulations to execute, and what parameters to use when doing this. See appendix C for a description of the batch files.

The available simulation types are described below:

**Random.** A random simulation will generate randomly generated user configurations, with the specified probability $P$ for a user being revoked, and do this for the specified number of iterations. The statistics output from the simulation will be the mean value of the statistics values generated by the algorithm at each iteration.

The user configuration will contains exactly the specified amount of revoked users. Thus, if $P$ is set to 0.3, exactly 30% of the users will be revoked for every iteration.

**Neighbour Dependent.** A neighbour dependent simulation will generate random user configurations where the probability of a user being revoked depends on the state of the previous user. The input probabilities are the probability for the first user being revoked, the probability for a user being



Axis labels

X Revoked users

Y Cover size

Conditions

Algorithm = *

Algorithm = *

Options

□ Logarithmic

□ Include y=0

Figure B.6: The statistics chart frame.

revoked if the previous user is revoked, and finally the probability for a user being revoked if the previous user is privileged. This will be done for the specified number of iterations, and the output from the simulation will, as with the random simulation, be the mean value of the statistics generated by the algorithm at each iteration.

# Appendix C

# Batch Files

*This appendix describes the structure and syntax of the batch files.*

## C.1 Introduction

The BEAF batch language is created to allow the user to run a series of simulations within given parameter limits. An example of this could be to perform the following simulation:

1. For both the CS and SD algorithms...

2. For 64, 128, 256 and 512 users...

3. Perform ”random” simulations of 1000 iterations each and the probability for a user being revoked from 0% to 100% with 10% increments.

The batch file contents for this simulation would be:

```
beaf
algorithm complete subtree
users (from 64 to 512 mult 2)
simulation random (from 0.0 to 1.0 add 0.1) 1000
do
algorithm subset difference
do
```

The general structure of a batch file, and the meaning and syntax of all batch commands are described in the following sections.

## C.2 Batch File Structure

A batch file has the following structure:

- The first non-comment and non-blank line in a batch file has to be the word *beaf* to identify the file as a batch file.
- The batch file can contain an unlimited amount of batch commands.
- Both single- and multi-line comments are allowed using Java-style comments, namely // for single line and /* ... */ for multi line comments.
- Empty lines are allowed and ignored by the parser.
- Both upper- and lowercase characters are allowed (the parser treats all characters as lowercase).

Many of the commands can take *enumerations* as parameters. An enumeration works much like a Java for-loop, describing the start, stop and increment values of the enumeration. The syntax for the enumerations are as follows (observe the parenthesis):

```
(from X to Y add Z)
(from X to Y mult Z)
X
```

Where X, Y and Z are numbers. Note that a single value is valid input for an enumeration.

## C.3 Batch Command Syntax

All available batch commands are described below, in order of appearance in a normal batch file. Some notes about the command arguments: *string*-parameters accept any alphanumeric string, *number* only accepts a single value (no enumerations), and *enumeration* accepts all types of enumerations (the syntax of an enumeration is described in the previous section).

### C.3.1 beaf

This is the batch file identifier that must be on the first non-comment and non-empty line in the batch file. If not, the file will not be recognized as a valid batch file.

### C.3.2 algorithm *string*

The `algorithm` command sets the current algorithm to use in the simulations. Subsequent algorithm commands will override the previous algorithm commands, and clear all previous `parameter/parameters` settings (see sections C.3.4 and C.3.5).

Example: `algorithm Subset Difference`

### C.3.3 users *enumeration*

This command defines the maximal number of users used in the following simulations. Should be a power of two.

Example: `users 1024` will set the maximum users to 1024. `users (from 512 to 4096 mult 2)` will iterate the user size from 512 to 4096 by multiplying the size by two each iteration.

### C.3.4 parameter *number enumeration*

The `parameter` command will set the parameter specified by the first argument to the value of the second argument. For possible parameter values, see the *Algorithm/Parameters* menu in the main application. Subsequent `parameter` commands will override the previous if they set the same parameter. Also note that the `algorithm` command will clear all previous calls to `parameter`.

Example: `parameter 0 3` will set parameter 0 to 3. `parameter 1 (from 0 to 2 add 1)` will iterate parameter 1 from 0 to 2 in the following simulations.

### C.3.5 parameters *enumeration enumeration ... enumeration*

The `parameters` (observe the 's') command sets all algorithm parameters at once. The number of arguments is the same as the number of possible parameters for the current algorithm. Subsequent `parameters` will override the previous. Also note that the `algorithm` command will clear all previous calls to `parameters`.

Example: Setting all parameters for an algorithm that has four parameters can be done with `parameters 1 0 (from 0 to 3 add 1) 3`. This will set parameter 0 to 1, parameter 2 to 0, parameter 3 will iterate through the values 0-3 when doing a simulation, and parameter 3 will be set to 3.

### C.3.6 simulation

The `simulation` command sets the current simulation type with specified parameters. The parameters depends on the simulation type, as described below.

See section B.3 for descriptions of the simulation types. Subsequent `simulation` commands will override the previous.

**simulation random** *enumeration enumeration* will perform a simulation of type "random" where the first argument is the probability for a user being revoked, and the second argument is the number of iterations.

Example: `simulation random 0.3 1000`

**simulation neighbour** *enumeration enumeration enumeration enumeration* will perform a simulation of type "neighbour dependent" where the first argument is the probability for user 0 being revoked, the second argument is the probability for a user being revoked if the previous neighbour is revoked, the third is the probability for a user being revoked if the previous neighbour is privileged, and the last argument is the number of iterations for this simulation.

Example: `simulation neighbour 0.5 0.8 0.8 1000`

### C.3.7 do

The `do` command will execute the current simulation. Since a simulation does not execute until the `do` command, this can be used to perform the same type of simulation for multiple algorithms, without having to repeat the `user` and `simulation` commands.

Example:

```
algorithm complete subtree
users 1024
simulation random 0.1 100
do
algorithm subset difference
do
```

# Appendix D

# Class Descriptions

*This appendix contains a listing of the most important public classes in BEAF and a description of each class. For a more detailed documentation check the generated javadoc for the project.*

| Class | Description |
|---|---|
| AlgorithmFactory | This class is used by the framework to create instances of all the implemented algorithms. The factory provides a BroadcastEncryptionAlgorithm and an AlgorithmPanel, which is used by the framework to execute and display the algorithm. When implementing a new algorithm this class should be extended to support the new algorithm. |
| AlgorithmPanel | This is the base class for all classes that want to have a graphical representation of the algorithm. |
| BinaryTreeAlgorithm | An abstract class providing a representation of a binary tree that can be used by subclasses to this class. The class contains a Node class which contains information about the nodes parent, left and right child etc. |
| BinaryTreeDisplayPanel | A panel containg both a binary tree panel and other GUI-components such as buttons and checkboxes. This panel is subclassed for the Complete Subtree and the Subset Difference algorithm. |
| BinaryTreePanel | A panel for drawing a graphical representation of a binary tree. This panel is subclassed for the Complete Subtree and the Subset Difference algorithm. It defines methods for drawing the tree, indexes and the users. |

Table D.1: Overview of package beaf.

| **Class** | **Description** |
|---|---|
| BroadcastDecryption-Algorithm | The base class for all broadcast decryption algorithms. It defines methods for initializing the algorithm and for decrypting messages on the clients. |
| BroadcastEncryption-Algorithm | The base class for all broadcast encryption algorithms. It defines methods for initializing the algorithm and for broadcasting messages to the clients. It also provides a general interface for getting and setting parameters for the algorithms. |
| BroadcastMessage | A broadcast message sent by a broadcast encryption algorithm to the clients. It contains a header part and a body part. The header contains algorithm- specific information to decrypt the message. The body only consists of an encrypted message. |
| ClientPanel | JPanel that runs a Client thread and displays the output from the client in a text area. The client thread is activated and deactivated by a checkbox. |
| Header | The base class for the header in a broadcast message. The data in this header is specific for the algorithm in use. This class provides an array of data items that can be used by subclasses. |
| Index | An index to a node in a binary tree. The internal representation of this index allows for very fast binary tree operations such as determining if a node is a descendant to another node. |
| MainApplication | The entry point and main application of BEAF. It initializes the server, the algorithm objects and creates the main GUI of the application. |
| MainFrame | The application's main GUI class. |
| Message | Contains the data that is transfered in a broadcast message. This may be either plaintext or ciphertext. The integrity of the data may be checked using a CRC-algorithm. This can also be used to check that the message is decrypted properly. |
| Packet | This class is used by the entire framework for when sending data over the network. The packet type defines the contents of the packet and determines how to interpret the data object. |

Table D.2: Overview of package beaf, continued.

| Class | Description |
|---|---|
| Server | A thread that handles all communication with the clients. The server listens to a port and accepts incoming connections. It also provides an interface for sending and receiving data to and from the clients. |
| Simulation | Class containing all the available simulations. Each simulation has a its own thread to execute the simulation, so that it does not lock the application GUI updates (the progress dialog for example). All methods will show an input dialog for simulation data when called. |
| Statistics | This class is used for storing statistics data. All algorithms and simulations reports their specific statistics when executing. When reporting a statistic with addValue(...), please use the existing STATISTIC_xyz strings as statistic title if possible, to allow for comparision of statistics between different algorithms. |
| StatisticsChartFrame | This is the GUI for representing and interacting with the chart. It displays the information in a Statistics object. |
| StatisticsPanel | This object shows the data stored in a Statistics object. |
| Stopwatch | Allows timing of the execution of any block of code. |
| UserConfiguration | This class is used to define the access rights of a set of users. A user can either be privileged or revoked. The first user always has index 0. It also defines methods for saving/loading the configuration to/from a file. |
| VirtualServer | This class simulates the behaviour of a real server without using any TCP sockets or object serialization. It automatically contains one client with id=0. This class is good to use when running big simulations since it increases performance of the algorithms. |

Table D.3: Overview of package beaf, continued.

| Class | Description |
|---|---|
| Client | This class handles all communication with the server. It checks incoming packets and routes them to the correct decryption algorithm. |
| ClientApplet | This is the GUI for the client application which can be run in any browser. It displays algorithm output and connection status. |
| CompleteSubtreeClient | This class implements the Complete Subtree decryption algorithm on the client. It is directly dependant on the CompleteSubtreeAlgorithm class. |
| LKHClient | This class implements the LKH decryption algorithm on the client. It is directly dependant on the LKHAlgorithm class. |
| SubsetDifferenceClient | This class implements the Subset Difference decryption algorithm on the client. It is directly dependant on the SubsetDifferenceAlgorithm class. |

Table D.4: Overview of package beaf.client.

| Class | Description |
|---|---|
| CompleteSubtree-Algorithm | This class implements the Complete Subtree algorithm. It can handle any number of users depending on how much memory is available. (It needs approximately 32 bytes per node.) The encryption method for both subset keys and broadcast message is DES. |
| CompleteSubtree-DisplayPanel | A panel containing both a Complete Subtree algorithm panel and other GUI-components such as buttons and checkboxes. |
| CompleteSubtreeHeader | The broadcast message header for the Complete Subtree algorithm. The header contains a list of items; one for each subset in the cover. Each header item contains a subset index and an encrypted key for that subset. |
| CompleteSubtreePanel | A panel containing the GUI-representation of the Complete Subtree algorithm. It defines methods for drawing the subset-cover and allows the user to click on a node to switch the access right between privileged and revoked. |

Table D.5: Overview of package beaf.cs.

| Class | Description |
|---|---|
| Label | A label in the Subset Difference algorithm. This label is represented as a a 64-bit data block, which is used as key-material for a DES-key. An instance of this class may also be used to derive new intermediate labels through the use of a pseudo-random generator. This class provides a dummy implementation of this generator, which is only used for testing purposes. Note: The 64-bit data is not parity adjusted. |
| Subset | A subset in the Subset Difference algorithm. Contains an index pair (i,j) that reference two nodes in a binary tree. Node i must be an ancestor to node j for the subset to be valid. |
| SubsetDifference-Algorithm | This class implements the Subset Difference algorithm. It can handle any number of users depending on how much memory is available. (It needs approximately 32 bytes per node.) The encryption method for both subset keys and broadcast message is DES. |
| SubsetDifference-DisplayPanel | A panel containing both a Subset Difference algorithm panel and other GUI-components such as buttons and checkboxes. |
| SubsetDifferenceHeader | The broadcast message header for the Subset Difference algorithm. The header contains a list of items; one for each subset in the cover. Each header item contains a subset and an encrypted key for that subset. |
| SubsetDifferencePanel | A panel containing the GUI-representation of the Subset Difference algorithm. It defines methods for drawing the subset-cover and allows the user to click on a node to switch the access right between privileged and revoked. |

Table D.6: Overview of package beaf.sd.

| Class | Description |
|---|---|
| LKHAlgorithm | This class implements the Logical Key Hierarchy algorithm for the special case where the key graph is a tree or a star. Several parameters is available to set the algorithm behaviour. |
| LKHKey | Used by the LKH algorithm to represent a k-node's key, including description if the key is encrypted or not, and if the key is an individual key. |
| LKHKeyTree | This class represents a key tree to be used with the LKH algorithm. The special case of where the tree is a star node (a root node with unlimited maximum degree) is also handled. |
| LKHPacket | Packet class that is used by the LKH algorithm. |
| LKHPanel | The AlgorithmPanel for the LKH algorithm. |

Table D.7: Overview of package beaf.lkh.