12-1-2009

An Efficient Time-Bound Hierarchical Key Assignment Scheme with a New Merge Function: A Performance Study

Rajasree Shyam
Department of Computer Science, Boise State University

AN EFFICIENT TIME-BOUND HIERARCHICAL KEY ASSIGNMENT SCHEME WITH A NEW MERGE FUNCTION: A PERFORMANCE STUDY

by

Rajasree Shyam

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

December 2009

APPROVAL TO SUBMIT PROJECT REPORT

This project presented by Rajasree Shyam entitled *An Efficient Time-Bound Hierarchical Key Assignment Scheme with a New Merge Function: A Performance Study* is hereby approved:

Jyh-haw Yeh Date
Advisor

Teresa Cole Date
Committee Member

Liljana Babinkostova Date
Committee Member

John R. Pelton Date
Dean of the Graduate College

This work is dedicated to my Beloved Family

ACKNOWLEDGMENTS

I owe the success of this project to many people, and I publicly thank them here. I am deeply indebted to my project committee. Dr. Yeh provided valuable theoretical guidance and technical expertise to help deliver analytical results for this project. He encouraged me every step along the way and was always a few steps ahead of me. He is a man of compassion and good knowledge. Dr. Cole inspired me by reviewing this document and suggesting pedagogical improvements to the layout of this report. I received valuable learning about the fundamentals of cryptology after having attended two cryptology courses delivered by Dr. Babinkostova.

The faculty and staff at the Computer Science department were very supportive and helped me as I juggled professional, educational and family priorities. They made graduate school life at BSU an enjoyable and memorable one. My heartfelt thanks to Professors Amit Jain, Tim Anderson, Jim Buffenbarger, Teresa Cole, Gang-Ryung Uh and Jhy-haw Yeh for providing a solid foundation in the various classes which I attended. Finally, my sincere thanks to my husband Shyam Venkataraman and my parents who supported and encouraged me to reach greater heights.

AUTOBIOGRAPHICAL SKETCH

Rajasree was born in Alleppey, Kerala, India. She received her BS degree in Chemistry at University of Kerala, India. Rajasree immigrated to Rochester, NY in 1998. She worked as a Test Engineer for Tobin Associates in Rochester. Rajasree moved to Boise, ID in 2003 and joined Hewlett Packard Company as a contract Test Lead where she worked for 3 years. She is currently working a permanent employee at Hewlett Packard Company as a Software Engineer in the Technology Services Group.

ABSTRACT

The advent of digital age has resulted in more television consumers switching to Digital TV with considerable improvement in image quality and ease-of-use. Consumers are able to select and view television programs and channels of choice by using a pay-per-use model or streaming video from their computer terminals. In all these use cases, the media provider requires a means by which they can restrict the consumers from watching selected programs for a pre-approved temporal interval. The consumer needs to be prevented access to certain pay-per-use channels and programs upon expiry of this pre-approved access. This necessitates the media provider to have a way to generate and assign time-bound secure access keys which could be granted and removed easily. In conventional key assignment schemes, one has to renew the keys periodically and redistribute the keys to the users accordingly. To allow a user to access all the authorized data over some temporal window, this straightforward implementation requires him/her to keep a lot of keys which is very inefficient.

In contrast to conventional schemes, a time-bound hierarchical key assignment scheme updates the keys periodically according to the class hierarchy and an entity only keeps a small amount of information for deriving all his entitled keys. Wang and Laih (WL) proposed a scheme with a concept of merging, which provides a systematic way to solve the problem. Yeh-Shyam (YS) scheme has improved on the WL scheme and has theoretically shown polynomial improvement in both memory and performance requirement. In this project, we will compare and contrast these secure key generation techniques and provide comparative analytical results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CA	– Central Authority
WL	– Wang-Laih Scheme
YS	– Yeh-Shyam Scheme
KA	– Key Assignment
gcd	– Greatest Common Denominator
mod	– Modulo operator

LIST OF SYMBOLS

C	Class
m	Number of classes in the global hierarchy
z	Time period
p, q	Secret parameters used by the CA
N	Public modulus
g	Base $g \in Z_N$
ε	Prime numbers stored by WL scheme
e_c	Public parameter used by YS scheme
d_c	Private parameter used by YS scheme
e_t	Public parameter used by YS scheme
d_t	Private parameter used by YS scheme
$K_{i,t}$	Encryption Key
$I(i, T)$	User Key
$\|T\|$	Number of time intervals in T

CHAPTER 1

INTRODUCTION

1.1 Access Control and Key Management

Access control is a central element of computer security. It is defined as the prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner. The principal objectives of computer security are as follows:

- Prevent unauthorized users from gaining access to resources
- Prevent legitimate users from accessing resources in an unauthorized manner
- Enable legitimate users to access resources in an authorized manner

The objective of key management is to assign keys to users and resources such that access rights are controlled efficiently and enforced correctly. The metrics used to measure the efficiency of key management schemes include:

- The size of private information stored for each user, this is the number of secret keys
- The computation involved to allow the user obtain access to the desired resource, this is the process of key derivation
- The additional computation involved when either the set of users change or the position of the user changes relative to other users in an hierarchy

- The amount of information held by the Central Authority (CA)

A solution to key management problem is to assign a key to each and every user for each class they are entitled to access by the Central Authority. This solution though simple will require each user to store an enormous number of keys and the storage requirement will grow out of control. Hence, a more efficient approach is to use key derivation schemes which allows the user to derives keys based on need.

1.2 Hierarchical Access Control Systems

Hierarchical access control exist in a number of computer science applications. For example, the Filesystem Hierarchy Standard (FHS) defines the main directories and their contents in Linux Operating Systems. In early 1996, the goal of developing a more comprehensive version of filesystem standard to address not only Linux, but other Unix-like systems was adopted with the help of members of Berkeley Software Distribution (BSD) development community. A typical filesystem provides the following roles and authorization:

1. User: a person who uses the system
2. User account: the foundation of security, a means of identifying users on a system
3. Group: a grouping of users sharing a common need
4. Group account: method used to group user accounts on a system

The operating system has the ability to control what resources users (or groups of users) can access. Different permission paradigms exist:

- No permissions: DOS, Win9x, MacOS9
- Simple permissions: default in UNIX and Linux
- Access Control Lists (ACLs): Windows NT/2K/XP/Vista; optional in Unix and Linux

The concept of filesystem permissions in UNIX operating system is managed with the three permission classes - *Owner*, *Group* and *World*

1. Owner: This may be the creator of a resource such as a file. For system resources, ownership may belong to a system administrator. For project resources, a project administrator or leader may be assigned ownership.
2. Group: In addition to the privileges assigned to an owner, a named group of users may also be granted access rights, such that the membership in a group is sufficient to exercise access rights. In most schemes, a user may belong to multiple groups.
3. World: The least amount of access is granted to users who are able to access the system but are not included in the categories owner and group for this resource.

There are a wide range of services and applications which require authentication and authorization at various hierarchical levels:

- With the advent of high-definition television system, there are various providers such as cable TV and dish network who provide Video-On-demand. Here the subscriber is allowed access to television programs for a pre-paid time period. In addition, there are also packaged services which allow subscribers to watch certain channels for an extended period of time.

- Enterprise Software which allow administrators and users to access for certain portions of the software application based on pre-paid subscription. There are also applications especially demo programs which work for 1-month allowing users access during trial periods.
- Computer and video games which allow access based on role and authorization.
- Digital libraries such as books and music which allow access to different levels of subscribers.
- Internet banking systems which allow access to certain portions of the banking system based on authentication, authorization and access control. For example, joint account users will be able to access certain areas of their account but lock out access to certain other areas. Online bill payment requires a different level of authorization.

1.3 Project Motivation

The motivation of this project is to describe and compare the implementations of the following two key generation algorithms:

- Wang-Laih (WL) Merging Function and Scheme [14]
- Yeh-Shyam (YS) New Merging Function and Scheme [16]

The goal of the project is three-fold:

- Describe the theory and design steps for the two schemes
- Implement the two schemes using the Java Programming language to generate the time-sensitive keys for a number of classes

- Measure the performance metrics of the two schemes, specifically memory requirement and computational complexity

1.4 Project Plan

The following is a list of itemized tasks which was planned and delivered for this project.

Item	**Task**	**Task Duration**
1.0	**Preliminary Investigation**	2 weeks
1.1	Read prior art and research papers	2 weeks
1.2	Understand the algorithms	4 weeks
2.0	**Design and Implement Schemes**	
2.1	Wang-Laih (WL) Scheme	4 weeks
2.1.1	Implement WL scheme and merge function	2 weeks
2.1.2	Generate WL time-sensitive keys	1 week
2.1.3	Test Key Assignment and Key Derivation	1 week
2.2	Yeh-Shyam (YS) Scheme	4 weeks
2.2.1	Implement YS scheme and merge function	2 weeks
2.2.2	Generate YS time-sensitive keys	1 week
2.2.3	Test Key Assignment and Key Derivation	1 week
3.0	**Data Analysis**	
3.1	Storage requirements	9 days
3.1.1	Identify parameters which will be used for storage	2 days
3.1.2	Compute and capture storage requirement results	1 week
3.2	Performance Requirements	9 days
3.2.1	Identify parameters which will be used for performance	2 days
3.2.2	Compute and capture performance requirement results	1 week
3.3	Compare WL and YS merge functions	3 days
4.0	**Documentation**	3 weeks
4.1	Project Write up	2 weeks
4.2	Project Presentation	1 week

Table 1.1: Project Plan

1.5 Project Organization

Chapter 2 describes the literature search done on various key generation algorithms based on prior art. This includes the seminal paper by Akl-Taylor which was optimized by MacKinnon et al. and a number of other researchers. We will describe the progression of algorithms and theoretically describe the performance and storage requirements. In Chapter 3 we will review Wang-Laih key assignment scheme, its encryption key generation, user key derivation including security and performance requirements. In Chapter 4, we will review our proposed update to WL merging technique - define the encryption key generation, user key derivation including YS scheme's security and performance requirements. In Chapter 5, we will describe the design and implementation details of the two schemes using the Java object oriented programming language. Here we will analyze the implementations in WL and YS schemes, compare and contrast the performance and storage results for the two afore-mentioned schemes. The solution will consist of the merging technique used by both WL and YS schemes and will show an improvement in security and performance analysis of our YS scheme. In Chapter 6 we will provide results and conclusions and provide recommendations for future research.

CHAPTER 2

KEY GENERATION IN HIERARCHICAL SYSTEMS

This chapter defines the problem of key generation and assignment in hierarchical systems. We will review and compare a number of schemes and the pros and cons of each approach. The main idea behind the key assignment scheme is to assign derivable encryption key to classes so that legal data flow can be controlled by these keys. A key assignment scheme ensures that a key K_i in class C_i is computationally feasible to derive another key K_j in class C_j if and only if users in C_i are allowed to access data in C_j. There is the concept of a central authority (CA) which maintains the keys and distributes any related public information. Most of these schemes operate on prime numbers and grow proportionately large with the depth of the hierarchy. During key generation and derivation, they perform operations such as multiplication, division, modular exponentiation and other complex crypto operation. Fig. 2.1 depicts a sample class hierarchy where class C_1 can derive keys for class C_4 but not the other way around.

The pioneering work which provided a solution to the problem of key management in hierarchical access control systems was put forth by Akl and Taylor [1] in 1983. They defined the term *security class* which divided the hierarchical system into a set of $C = C_1$,C_2,...C_m of classes with the binary relation $C_j \preceq C_i$. Users in C_j have *security clearance* lower than or equal to those in C_i. In other words, users in C_i have access

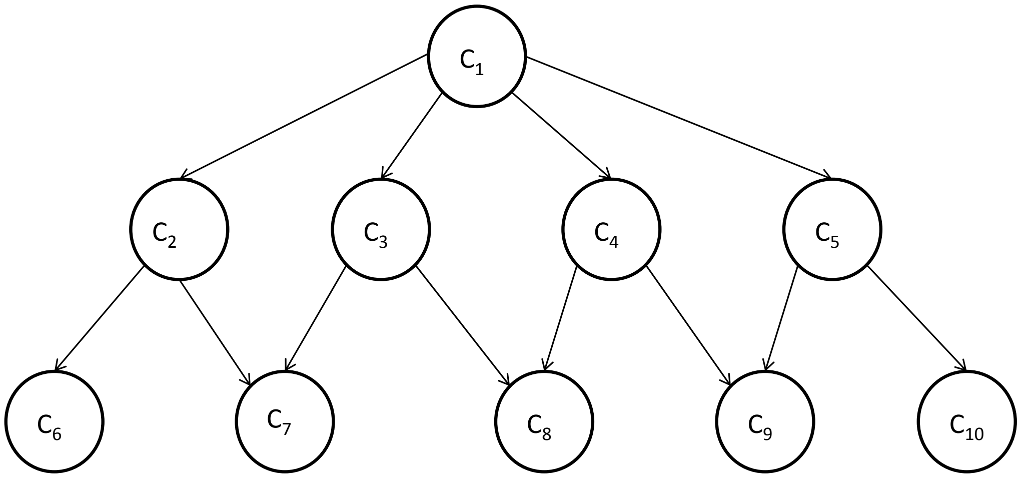


Figure 2.1: Class Hierarchy

to information held by users in C_j while the opposite is not allowed. A larger number of publications have been put forth which have provided improvements to Akl-Taylor key assignment scheme [8], [11], [15], [5], [17], [18], [3], [4] . Most researchers have proposed schemes that have better performance while inserting and deleting classes in the hierarchy. Some of the schemes have been proven to be vulnerable to security attacks [13], [10].

In 2002, Tzeng [13] extended the problem by adding an extra dimension where in the data flow not only depends on security classes but is also time senstive. He proposed a *time-bound* cryptographic key assignment scheme in which the cryptographic keys of a class are different for each time period, that is, the cryptographic key of class C_i at time t is $K_{i,t}$. This meant that a user in class C_i from time t_1 to t_2 is provided information $I(i, t_1, t_2)$ where $1 \leq i \leq m$, $t_1 \leq t_2$ such that with this information, he/she will be able to compute the key $K_{j,t}$ of C_j at time t if and only if $C_j \preceq C_i$ and $t_1 \leq t \leq t_2$. Without further modifications and optimizations, schemes which use the Akl-Taylor methodology would require users to keep track of a large number of keys which would need to be regenerated for various time intervals. Tzeng considered the time-versus-space tradeoff and proposed a scheme where the size of $I(i, t_1, t_2)$ was independent of the number of total classes and time period in the hierarchy. The scheme is efficient in key storage and computation time due to the fact that the same key can be utilized with varied time bounds and for multiple sessions. This time-bound hierarchical access control scheme extended to applications such as pay-on-demand cable TV system and cryptographic key backup systems. Unfortunately, Chien [4] proved that Tzeng's scheme was vulnerable to collusive attack and proposed a solution. Further work by Chien [5], Huang and Chang [7] have also been shown to be insecure against collusion.

There have been a number of schemes which have attempted to further optimize the storage and performance requirements needed by key generation and derivation schemes. A few such optimizations include batching and merging. Batching is a technique where key reassignment requests are accumulated until a predetermined threshold of requests is reached. A second technique called merging creates greater correlation between group keys by compressing them to single aggregate keys and works even for time-bound classes. Merging is very similar to image compression algorithms which use correlation between neighboring (past) pixels to help predict a value for the consecutive (future) pixel. An efficient key management requires a deterministic algorithm that ensures scalable growth in key size.

Recent developments. In 2006, Wang and Laih in [14] proposed a more optimal solution referred hereto as **WL** scheme for solving time-bound hierarchical access control system. The core aspect of the WL scheme is the merging technique. The system constructs a global hierarchy which consists of z sub-hierarchies, one for each time interval $t \in [1, z]$. Each sub-hierarchy in time interval t is the same for the original class hierarchy, except that each class C_i in the sub-hierarchy has an additional index t. A class C_i in each time interval t will be indexed as $C_{i,t}$. Each class $C_{i,t}$ will be assigned a piece of secret information called its primitive key $K'_{i,t}$. The encryption key $K_{i,t}$ or the user key $I(i, T)$ is an aggregate key which is a merge of the primitive keys of lower classes in the hierarchy. The merging technique finds its roots in the area of image compression. The encryption key $K_{i,t}$ for class $C_{i,t}$ is a merge of the primitive keys for all classes $C_{j,t} \preceq C_{i,t}$ in the global hierarchy. In a similar fashion, the user key $I(i, T)$ for a user in class C_i is a merge of the primitive keys for all the classes $C_{j,t} \preceq C_{i,t}$ for all time intervals $t \in T$ in the global hierarchy. Only the Central Authority which has knowledge of the secret information will be able to feasibly perform the merge operation. However, any user with the aggregate key $I(i,T)$ will be able to recover any of its sub-aggregate keys $K_{j,t}$, if the set of primitive keys for $I(i,T)$ is a superset of those for the sub-aggregate key $K_{j,t}$, that is when $C_{j,t} \preceq C_{i,t}$ and $t \in T$. Thus the key assignment scheme development becomes nothing but designing an optimal merge function. The merge function needs to be designed such that legal key generation is derivable but an illegel key derivation is not viable even if there is large amount of time.

CHAPTER 3

AKL-TAYLOR SCHEME

This scheme is a seminal key assignment scheme and is not time sensitive. Suppose CA wants to assign secret keys to a number of disjoint classes: $C_1, C_2, \cdots, C_m$ which are partially ordered with a binary relation "$\preceq$". Let K_i denote the key assigned to the class $C_i, i \in [1, m]$. The meaning of $C_j \preceq C_i$ is that it is feasible to derive K_j from K_i. In addition, from all K_js where $C_i \preceq C_j$, the users cannot derive K_i. The relationship between the classes may be described by a partial-order hierarchy, as shown in Fig. 3.1.

3.1 Setup

CA chooses two distinct large primes p and q which are secret and makes $N = p \cdot q$ a public value. The CA chooses a secret random $g \in Z_N^*$. For each class C_i, CA chooses a distinct prime ε_i and publishes it publicly.

3.2 Key Generation and Distribution

For each $i \in [1, m]$, the CA computes the secret key K_i by,

$$K_i = g^{e_i} (mod\ N) \tag{3.1}$$

where

$$e_i = \begin{cases} 1 & \text{if } C_i \text{ is a root} \\ \prod_{\{j:c_j \not\preceq c_i\}} \varepsilon_j & \text{otherwise} \end{cases} \tag{3.2}$$

3.3 Key Derivation

Let $C_j \preceq C_i$ and suppose the user of K_i wishes to compute K_j. Then he computes,

$$(K_i)^{e_j/e_i}\,(mod\ N) = (g^{e_i})^{e_j/e_i}\,(mod\ N) = g^{e_j}(mod\ N) = K_j \tag{3.3}$$

This computation is feasible only if $e_i|e_j$ which by design is valid since $C_j \preceq C_i$ iff $e_i|e_j$. Hence the user of K_i can always compute K_j if $C_j \preceq C_i$.

3.4 Example

Consider the class hierarchy depicted in Fig.3.1. The number of classes $m = 6$. First the CA chooses N and g. Then the CA makes the global hierarchy and assigns public primes $\varepsilon_1 = 2, \varepsilon_2 = 3, \varepsilon_3 = 5, \varepsilon_4 = 7, \varepsilon_5 = 11$ and $\varepsilon_6 = 13$ as shown in Fig.3.2. Fig.3.3 depicts the values of e_i for each class i.

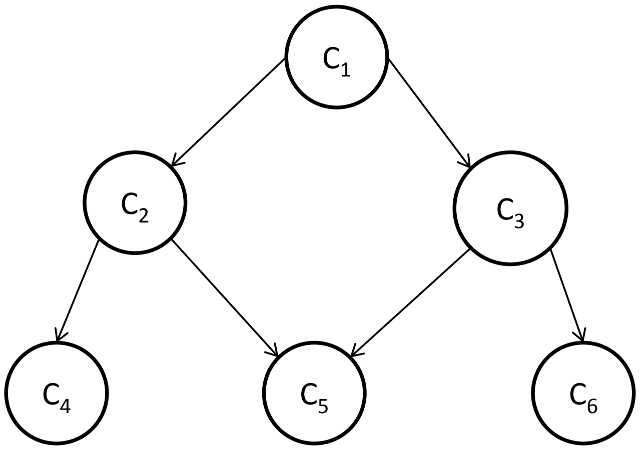


Figure 3.1: Class Hierarchy with 6 classes

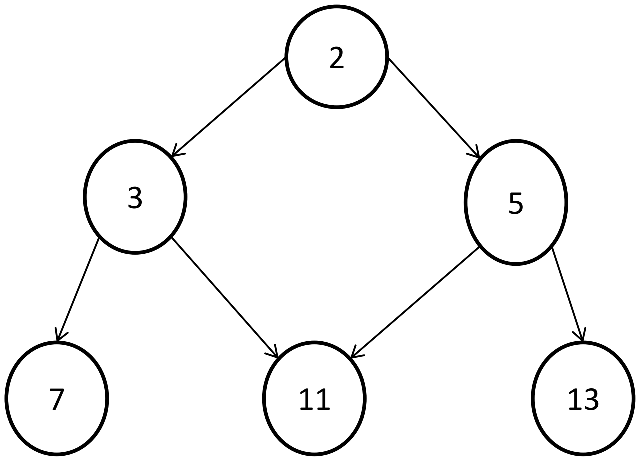


Figure 3.2: Class Hierarchy with ε values

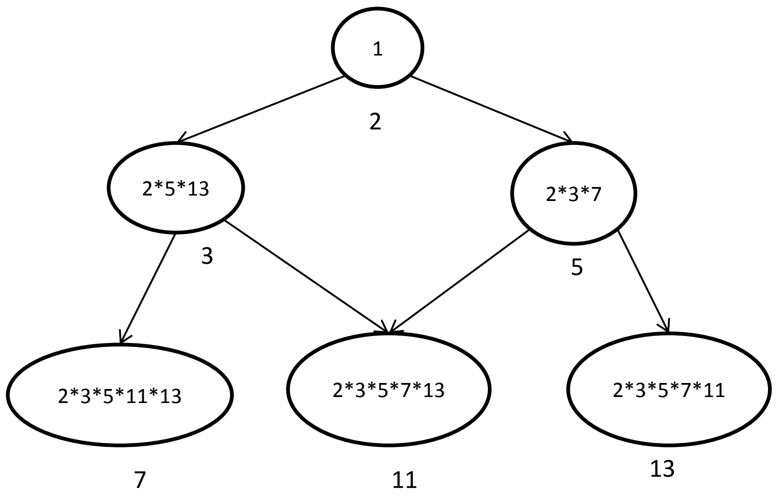


Figure 3.3: Class Hierarchy with values for e_i

CHAPTER 4

WANG-LAIH SCHEME

This scheme is a recent time-bound hierarchical key assignment scheme which has no cryptoanalysis done against it yet.

4.1 Setup and Initialization

The CA chooses a proper modulus N and g. For each class $C_{i,t}$ in the global hierarchy, the CA assigns a distinct prime $\varepsilon_{i,t}$. The primitive key $K^{'}_{j,t}$ for each class $C_{j,t}$ is defined as,

$$K^{'}_{j,t} = g^{E/\varepsilon_{j,t}} (mod\ N) \tag{4.1}$$

where,

$$E = \prod_{i \in [1,m], t \in [1,z]} \varepsilon_{i,t} \tag{4.2}$$

Note that the primitive keys are not real keys. The CA does not actually assign primitive keys to classes. The aggregate key is a merge of a set of primitive keys below the aggregate key in the global hierarchy. Let $S_{i,t}$ be the set of primitive keys for those classes below (and including) $C_{i,t}$ in the global hierarchy, that is,

$$S_{i,t} = \left\{ K^{'}_{j,t} : C_{j,t} \preceq C_{i,t} \right\} \tag{4.3}$$

Similarly, let $S_{i,T}$ be the union of primitive keys in all $S_{i,t}$, $t \in T$. That is,

$$S_{i,T} = \left\{ K'_{j,t} : C_{j,t} \preceq C_{i,t}; t \in T \right\} \tag{4.4}$$

4.2 Merge Function

Let $< g >$ denote the cyclic group generated by g in the ring Z_N. Given a modulus N and base $g \in Z_N^*$ with the input S being a set of integers in $< g >$, the merge function is defined as:

$$M_{g,N}(S) = g^{gcd\left\{log_{g,N}(k):k \in S\right\}}(mod\ N) \tag{4.5}$$

where $log_{g,N}(k)$ is the logarithm of k with respect to the base g in $< g >$.

4.3 Encryption Key Generation

For each class $C_{i,t}$, an encryption key $K_{i,t}$ is assigned by computing,

$$K_{i,t} = g^{\alpha_{i,t}}(mod\ N) \tag{4.6}$$

where,

$$\alpha_{i,t} = E / \left(\prod_{C_{j,t} \preceq C_{i,t}} \varepsilon_{j,t} \right) \tag{4.7}$$

By the definition of merging, the encryption key $K_{i,t}$ in equation 4.6 is nothing but the aggregate key merged from a set of primitive keys in $S_{i,t}$, that is,

$$K_{i,t} = M_{g,N}(S_{i,t}) \tag{4.8}$$

since

$$M_{g,N}\left(S_{i,t}\right) = g^{gcd(E/\varepsilon_{j,t}:C_{j,t}\preceq C_{i,t})}(mod\ N) \tag{4.9}$$

$$= g^{\alpha_{i,t}}(mod\ N) \tag{4.10}$$

4.4 User Registration

If the user is assigned to class C_i for a set of time intervals T, the CA gives the user an aggregate key $I(i,T)$ as a user key by computing,

$$I(i,T) = g^{\alpha_{i,T}}(mod\ N) \tag{4.11}$$

where

$$\alpha_{i,T} = E/\left(\prod_{C_{j,t}\preceq C_{i,t},t\in T} \varepsilon_{j,t}\right) \tag{4.12}$$

Similarly, $I(i,T)$ defined by 4.11 is again a merge of the set of primitive keys in $S_{i,T}$, that is,

$$I(i,T) = M_{g,N}(S_{i,T}) \tag{4.13}$$

since

$$M_{g,N}(S_{i,T}) = g^{\mathrm{gcd}(E/\varepsilon_{j,t}:C_{j,t}\preceq C_{i,t},t\in T)}(mod\ N) = g^{\alpha_{i,T}}(mod\ N) \tag{4.14}$$

4.5 Key Derivation

If a user with a user key $I(i,T)$ would like to derive an encryption key $K_{\kappa,\tau}$, he can compute,

$$I(i,T)^{\frac{\alpha_{\kappa,\tau}}{\alpha_{i,T}}} = (g^{\alpha_{i,T}})^{\frac{\alpha_{\kappa,\tau}}{\alpha_{i,T}}} = g^{\alpha_{\kappa,\tau}}\,(mod\ N) = K_{\kappa,\tau} \tag{4.15}$$

The above computation is feasible to compute if and only if the exponent $\frac{\alpha_{\kappa,\tau}}{\alpha_{i,T}}$ is an integer. Furthermore, the exponent $\frac{\alpha_{\kappa,\tau}}{\alpha_{i,T}}$ is an integer if and only if $C_\kappa \preceq C_i$ and $\tau \in T$. The proof can be found in [14].

4.6 Example

Consider the class hierarchy depicted in Fig.4.1. The number of classes $m = 4$. First the CA chooses a proper modulus N and g as in the Akl-Taylor scheme. Then the CA creates a global hierarchy which consists of subhierarchy for each time slot $t \in [1,t]$. The subhierarchies are the same as the original class hierarchy except that the classes are additionally indexed by t as depicted in Fig.4.2 with $t = [1,4]$. Each class C_i is replaced by $C_{i,t}$ for all i and time period. The constructed time-bound global hierarchy is depicted in Fig.4.3.

4.7 Security Analysis

In the WL scheme, the assigned keys $K_{i,t}$ are the result of applying Akl-Taylor scheme recursively, it is obvious that, from all $K_{j,\tau}$s where $\tau \neq t$ and all $K_{j,t}$s where $C_i \npreceq C_j$ (or $C_{i,t} \npreceq C_{j,t}$ equivalently), it is not feasible to derive $K_{i,t}$. In this scheme, the CA gives each user an aggregate key. A subset of users can join together, use each of their aggregate key collectively but will not be able to derive unauthorized keys. This

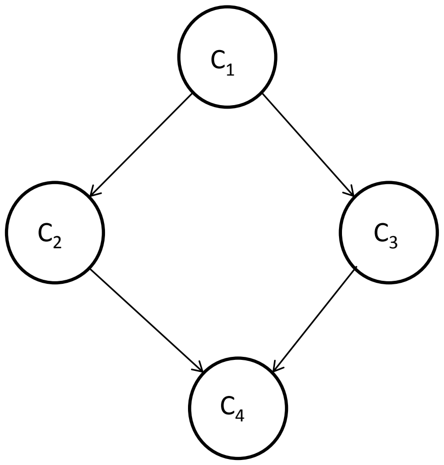


Figure 4.1: Class Hierarchy [1]

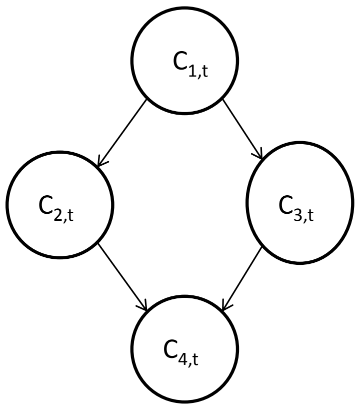


Figure 4.2: Sub-hierarchy for time period $t \in [t_1, t_2]$ from [1]

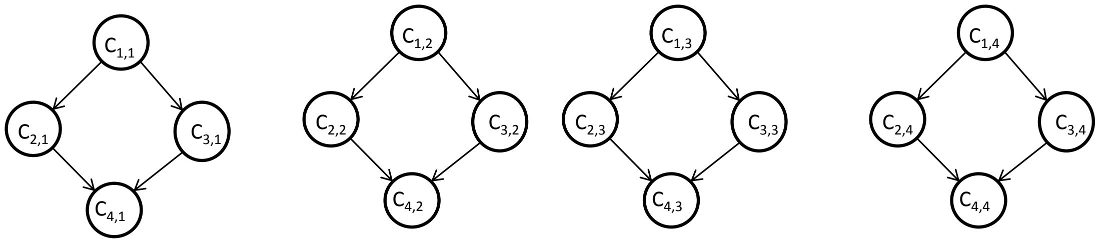


Figure 4.3: An example of the global hierarchy of time-bound WL scheme for time period $[1, 4]$

is due to the fact that aggregate keys are just equivalent to some authorized set of assigned keys, it is impossible to get any more from them. The derivation of the security analysis propositions can be found in [14].

4.8 Performance Analysis

The main advantage of the WL scheme is that the CA only needs to transmit one single aggregate key $I(i,T)$ to a user of class C_i whose length is as long as the public parameter N indenendent of the number of time slots T. To optimize complexity-memory-time tradeoff, the CA might choose to store N, g, all public primes and the assigned keys for the current time slot. The term $\varphi^{'}$ denotes the average bit length of a public prime in the WL scheme where,

$$\varphi^{'} = mz\log_2(mz) \tag{4.16}$$

m denotes the number of classes in the global hierarchy and z represents the time period. In most applications, z is much larger than m. $|N|$ denotes the bit length N. The term $|T|$ denotes the number of time slots in T. The term α_i denotes the number of classes lower than or equal to class C_i in the hierarchy, i.e., $\alpha_i = |\{C_j : C_j \preceq C_i\}|$. Given this information, the CA's computation requirement of each assigned key and aggregate key is equivalent to one modular exponentation. Similarly, the computation required by each user to derive an assigned key from an aggregate key is one modular exponentiation. If the CA computes all assigned keys from a given time slot, it can derive them in an iterative fashion to speed up computation. It first computes the assigned key of the root class in the hierarchy and then computes those of the child classes. With the keys for the child classes, it can then compute the keys for the lower classes using the derived child keys.

Tables 4.1 and 4.2 provide a summary of the storage and computation requirements needed by WL scheme.

	Item	Storage(bits)
CA	N, g, public and secret parameters	$2\,\|N\| + (mz\varphi^{'})$
	Encryption keys (current time period)	$m \cdot \|N\|$
Users (of class C_i)	$I(i,T), N$ and public parameters	$2 \cdot \|N\| + (\alpha_i + \|T\|)\varphi^{'}$

Table 4.1: Storage Requirement for Wang and Laih (WL) scheme

	Item	# of modular mult.
CA	Per encryption key in average	$1.5\varphi^{'} z$
	A user key $I(i,T)$	$1.5\varphi^{'}(mz - \alpha_i\,\|T\|)$
	Adding a time interval to a user key $I(i,T)$	$0.5\ \alpha_i\varphi^{'} + 1.5\|N\|$
Users (of class C_i)	A key derivation: from $I(i,T)$ to $K_{\kappa,\tau}$	$1.5\varphi^{'}(\alpha_i\,\|T\| - \alpha_\kappa)$

Table 4.2: Computational Complexity for WL scheme

CHAPTER 5

YEH-SHYAM SCHEME

The YS scheme is a recently proposed time-bound key assignment scheme.

5.1 Initialization and Setup

1. The CA chooses two large secret primes p and q, and computes $N = pq$ and $\phi(N) = (p-1)(q-1)$.

2. For each security class C_i, the CA chooses a distinct prime e_{C_i} which is relatively prime to $\phi(N)$. The CA then determines the multiplicative inverse d_{C_i} for each $e_{C_i}(mod\ \phi(N))$, i.e., $e_{C_i}d_{C_i} = 1(mod\ \phi(N))$ for $1 \leq i \leq m$.

3. For each interval t, the CA chooses another distinct prime e_t which is relatively prime to $\phi(N)$. Then it determines the multiplicative inverse d_t for each $e_t \cdot mod(\phi(N))$, i.e., $e_t d_t = 1(mod\ \phi(N))$ for $1 \leq t \leq z$.

4. The CA chooses an integer $g \in Z_N$ as a base modulo N.

5. Let $D = \{d_{C_1}, d_{C_2}, \ldots d_{C_m}, d_1, d_2, \ldots d_z\}$ and $E = \{e_{C_1}, e_{C_2}, \ldots, e_{C_m}, e_1, e_2, \ldots, e_z\}$. The CA only publishes the parameters in E and the product N. All other parameters will be kept secret.

5.2 The New Merge Function

Based on Fermat's Little Theorem, for each $i \in [1, u]$, we have

$$g^{d_i e_i} = g^{d_i e_i (mod\ \phi(N))} = g(mod\ N) \tag{5.1}$$

Note that each parameter in both D and E has an identity so that, given an identity of a parameter $d_i \in D$, even if the value of d_i is unknown (since it is a secret parameter), the identity of the corresponding parameter $e_i \in E$ can be identified, as well as e_i's value can be retrieved. For simplicity, let $\{z_1, z_2, \cdots, z_k\}(mod\ N)$ denote the set $\{z_1(mod\ N), z_2(mod\ N), \cdots, z_k(mod\ N)\}$. The following define the merge function, please refer to [16] for further details.

Definition 1. Given a modulus $N = pq$, a base $g \in Z_N$ and a set D of u secret parameters as defined above, let $S = \{g^{x_1}, g^{x_1}, \cdots, g^{x_w}\}(mod\ N)$ denote a set of w integers, where, for each $i \in [1, w]$, x_i is a product of some integers in D. That is, if $x_i = d_{i_1}, d_{i_2}, \cdots, d_{i_k}$, let $\text{FD}(x_i) = \{d_{i_1}, d_{i_1}, \cdots, d_{i_k}\}$ denote the factors of x_i in D.

Definition 2. Given a set of integers $S = \{g^{x_1}, g^{x_2}, \cdots, g^{x_w}\}(mod\ N)$ as in definition 1, define $R(S) \subseteq D$ as follows: $R(S) = FD(x_1) \cup FD(x_2) \cup \cdots \cup FD(x_w)$.

Definition 3. Given a set of integers $R(S) \subseteq D$ as in definition 2, let $R'(S) \subseteq E$ denote the corresponding set of integers to $R(S)$. That is, for each parameter $d \in R(S)$, there is one and only one parameter $e \in R'(S)$ such that $g^{de} \equiv g(mod\ N)$ for any $g \in Z_N$.

Definition 4. Given a modulus $N = pq$ and a base $g \in Z_N$, with a set of integers $S = \{g^{x_1}, g^{x_1}, \cdots, g^{x_w}\}\,(mod\ N)$ as in definition 1, the merge function $M_{g,N}(S)$ is defined as:

$$M_{g,N}(S) = g^{(\prod_{d \in R(S)} d)} (mod\ N) \tag{5.2}$$

For example, if $S = \{g^{d_1 d_2}, g^{d_2 d_3}, g^{d_3 d_4}, g^{d_1 d_4}\}\ (mod\ N)$, then $M_{g,N}(S) = g^{\prod_{d \in R(S)}} d = g^{d_1 d_2 d_3 d_4}(mod\ N)$ where $R(S) = FD(d_1 d_2) \cup FD(d_2 d_3) \cup FD(d_3 d_4) \cup FD(d_1 d_4) = \{d_1, d_2\} \cup \{d_2, d_3\} \cup \{d_3, d_4\} \cup \{d_1, d_4\} = \{d_1, d_2, d_3, d_4\}$. If S is a set of keys, then the above merge function merges them together into a single aggregate key, and each individual key can be retrieved from the aggregate key. In the example, the key $g^{d_1 d_2}(mod\ N)$ can be retrieved by computing

$$(M_{g,N}(S))^{e_3 e_4} \equiv (g^{d_1 d_2 d_3 d_4})^{e_3 e_4} \equiv g^{d_1 d_2} (mod\ N) \tag{5.3}$$

The proofs for the propositions can be found in [16].

Proposition 1. Let S_1 and S_2 be two integers as in definition 1. Given only the identities of parameters in sets $R(S_1)$ and $R(S_2)$, $M_{g,N}(S_1)$ can be derived from $M_{g,N}(S_2)$ if $S_1 \subseteq S_2$ (or more specifically $M_{g,N}(S_1)$ can be derived from $M_{g,N}(S_2)$ if $R(S_1) \subseteq R(S_2)$).

Proposition 1 shows than an aggregate key merged from a set of keys S_1 can be derived from its super-aggregate key merged from another set of keys S_2, where $S_1 \subseteq S_2$.

Definition 5. Let S_1 and S_2 be two sets of integers as in definition 1. If $R(S_1) \subseteq R(S_2)$, then $M_{g,N}(S_2)$ is a super-aggregate key of $M_{g,N}(S_1)$ or equivalently $M_{g,N}(S_1)$ is a sub-aggregate key of $M_{g,N}(S_2)$.

Proposition 2. Let r be a positive integer. Given sets $S, S_1, S_2, \cdots, S_r$ as defined in definition 1, where $S = S_1 \cup S_2 \cup \cdots \cup S_r$, then $M_{g,N}(S) = M_{g,N}(M_{g,N}(S_1), M_{g,N}(S_2), \cdots, M_{g,N}(S_r))$.

Proposition 2 indicates that an aggregate key can be merged directly from a set of primitive keys or can be merged from a set of sub-aggregate keys. It also shows that an aggregate key will not be changed by merging duplicate keys.

5.3 Encryption Key Generation

There is a primitive key $K'_{j,t}$ (not a real key) assigned to each class $C_{j,t}$, where

$$K'_{j,t} = g^{d_{c_j} \cdot d_t} (mod\ N) \tag{5.4}$$

The CA assigns an aggregate key $K_{i,t}$ as an encryption key for each class C_i in each time interval t, where

$$K_{i,t} = g^{\left(\prod_{c_j \preceq c_i} d_{c_j}\right) d_t} (mod\ N) \tag{5.5}$$

Equation 5.5 is based on the merge concept, where $K_{i,t}$ is the merge of the primitive keys in $S_{i,t}$, that is

$$K_{i,t} = M_{g,N}\left(S_{i,t}\right) \tag{5.6}$$

where

$$S_{i,t} = \left\{K'_{j,t} : C_j \preceq C_i\right\} \tag{5.7}$$

since

$$M_{g,N}\left(S_{i,t}\right) = g^{\left(\prod_{d \in R\left(S_{i,t}\right)} d\right)} (mod\ N) = g^{\left(\prod_{C_j \preceq C_i} d_{C_j}\right) d_t} (mod\ N) \tag{5.8}$$

5.4 User Registration

When a user is assigned to class C_i for a set of time interval T, the CA gives the user an aggregate key $I(i,T)$ as a user key, where

$$I(i,T) = g^{\left(\Pi_{C_j \preceq C_i} d_{C_j} \cdot \Pi_{t \in T} d_t\right)} (mod\ N) \tag{5.9}$$

Again, the aggregate key $I(i,T)$ is a merge of the primitive keys in $S_{i,T}$, that is,

$$I(i,T) = M_{g,N}\left(S_{i,T}\right) \tag{5.10}$$

where

$$S_{i,T} = \left\{K'_{j,t} : C_j \preceq C_i \text{ and } t \in T\right\} \tag{5.11}$$

since

$$M_{g,N}\left(S_{i,T}\right) = g^{\left(\Pi_{d \in R\left(S_{i,T}\right)} d\right)} (mod\ N) = g^{\left(\Pi_{C_j \preceq C_i} d_{C_j} \cdot \Pi_{t \in T} d_t\right)} (mod\ N) \tag{5.12}$$

Note that the aggregate key $I(i,T)$ can be expressed as

$$I(i,T) = M_{g,N}\left(K_{i,t} : t \in T\right) \tag{5.13}$$

since according to Proposition 2,

$$M_{g,N}\left(K_{i,t} : t \in T\right) = M_{g,N}\left(M_{g,N}\left(S_{i,t}\right) : t \in T\right) = M_{g,N}\left(S_{i,t} : t \in T\right) = M_{g,N}\left(S_{i,T}\right) \tag{5.14}$$

5.5 Key Derivation

A user who is in class C_i for a set of time intervals T can use the assigned user key $I(i,T)$ to derive an encryption key $K_{\kappa,\tau}$ of class C_κ in time interval τ if $C_\kappa \preceq C_i$ and $\tau \in T$. If $C_\kappa \preceq C_i$ and $\tau \in T$, then $S_{\kappa,\tau} = \left\{K'_{j,\tau} : C_j \preceq C_\kappa\right\} \subseteq S_{i,T} = \left\{K'_{j,t} : C_j \preceq C_i, t \in T\right\}$ and thus, by proposition 1, the key $K_{\kappa,\tau}$ can be derived from the user key $I(i,T)$. To derive the key one needs to perform the following modular exponentiation:

$$I(i,T)^{\left(\prod_{e \in R'\left(S_{i,T}\right) - R'\left(S_{\kappa,\tau}\right)} e\right)} (mod\ N) \tag{5.15}$$

which is equivalent to

$$I(i,T)^{\left(\prod_{C_j \preceq C_i, C_j \not\preceq C_\kappa} e_{C_j} \cdot \prod_{t \in T, t \neq \tau} e_t\right)} (mod\ N) \tag{5.16}$$

5.6 Example

Let us refer to the hierarchy in Figure 4.3, there are 4 classes and the time is divided into 4 intervals. There are $2(m+z) = 16$ pairs of parameters
$(d_{c_1}, e_{c_1}), \cdots, (d_{c_4}, e_{c_4}), (d_1, e_1), \cdots, (d_4, e_4)$. Suppose the CA authorizes a user the access right of C_2 for time interval 2 and time interval 4. The CA assigns the user a user key $I(2,\{2,4\})$, where,

$$I(2,\{2,4\}) = g^{(\prod_{c_j \preceq c_2} d_{c_j} \cdot \prod_{t \in T} d_t)} (mod\ N) = g^{d_{c_2} d_{c_4} d_2 d_4} (mod\ N)$$

If the user wants to derive an encryption $K_{4,4}$, the user would compute
$I(2,\{2,4\})^{e_{c_2} e_2} (mod\ N) = K_{4,4}$.

5.7 Security Analysis

We will briefly provide the security analysis of our scheme, detailed information can be found in [16]. Let us consider a scenario if the public parameters are not relatively prime to each other, then it is possible to have two classes C_i and C_j such that $C_i \npreceq C_j$ and C_j's public parameter e_{c_j} is a multiple of C_i's public parameter e_{c_i}. This scenario is not valid in our scheme due to the fact that all public parameters are primes, which ensures that they are not divisible by one another. The following two propositions show other basic security properties of our scheme.

Proposition 3. In our scheme, without knowing the secret parameters, it is not feasible to merge a set of keys if none of the keys is a super-aggregate key of all other keys in the set.

Proposition 4. In our scheme, without knowing the secret parameters, a key can be feasibly derived if and only if any of its super-aggregate keys is known.

Proposition 5. In our scheme, it is feasible for a user with a user key $I(i,T)$ to derive an encryption key $K_{\kappa,\tau}$ if and only if $C_\kappa \preceq C_i$ and $\tau \in T$.

Proposition 6. In our scheme, if it is feasible to derive $K_{\kappa,\tau}$ from aggregate keys $I(i_1,T_1), I(i_2,T_2), \cdots, I(i_r,T_r)$, where r is a positive integer, then there is at least one $\rho \in [1,r]$ such that $C_\kappa \preceq C_{i_\rho}$ and $\tau \in T_\rho$.

5.8 Performance Analysis

5.8.1 Storage Requirements

A pair of public and secret parameters are assigned to each class and to each time interval. There are $m + z$ pairs of parameters. Public parameters are selected from a sequence of prime numbers starting with 3. Primes in the sequence which are not relatively prime to $\phi(N)$ are skipped. The advantage of picking small primes is that it not only reduces the storage requirements but also increases the key derivation efficiency. We make the assumption that the n-th public parameter is approximately the same as the n-th prime. secret parameters are multiplicative inverses of the public parameters ($mod\ \phi(N)$) and are normally the size of N denoted as $|N|$. Hence the storage requirement for secret parameters is $(m + z)|N|$ bits. The CA only stores the encryption keys of the current time interval. All aggregate keys including the encryption keys and user keys are of size $|N|$. Thus m current encryption keys need $|N|$ bits. Table 5.1 summarizes the storage requirement for the CA and users for our scheme. The term α_i denotes the number of classes lower than or equal to class C_i in the hierarchy, i.e., $\alpha_i = |\{C_j : C_j \preceq C_i\}|$. The term $|T|$ denotes the number of time intervals in T. The term φ denotes the average bit length of a public prime in our scheme, i.e., $\varphi' = m\varphi$. In most applications, z is much larger than m. Thus, the average size of a public prime in the WL scheme is about m times longer than the average size of a public parameter in our scheme. For systems with very small number of classes and time intervals, the CA in our scheme may require more storage. When the number of classes m or the number of time intervals z grows, the storage requirement in our scheme will grow much slower than in the WL scheme. The CA storage requirement in our scheme is about $O(1/m^2)$ fraction of those in the WL scheme.

	Item	Storage(bits)
CA	N, g, public and secret parameters	$(m + z + 2)\,|N| + (m + z)\varphi$
	Encryption keys (current time period)	$m \cdot |N|$
Users (of class C_i)	$I(i, T), N$ and public parameters	$2 \cdot |N| + (\alpha_i + |T|)\varphi$

Table 5.1: Storage Requirement for the proposed Yeh-Shyam (YS) scheme

5.8.2 Computational Complexity

The initialization step is a one time process which involves assigning parameters followed by encryption and user key generation or modification. The key generation and modification can be handled on the fly. For the initialization step, the YS scheme requires to identify $(m+z)$ primes and computes their multiplicative inverses $(mod\ \phi(N))$. It requires $(m+z)\varphi$ divisions to compute all $m+z$ multiplicative inverse d's. As compared to YS scheme, the WL scheme does not require computing the multiplicative inverse in the initialization step. However, the WL scheme requires mz primes which is more than $m+z$ primes needed by the YS scheme. Identifying a large prime e is much more difficult than computing the multiplicative inverse d of e. Thus the YS scheme is more efficient than WL scheme during the initialization step.

The key generation and modification in the YS scheme is nothing but a modular exponentiation. The key derivation from a user key to an encryption key is also a modular exponentiation. The average number of modular multiplications to compute a modular exponentation with an exponent is about 1.5x (bit length of the exponent). To assign all encryption keys for a time interval, the CA computes the keys in a top-down fashion. Then CA first computes the encryption key of the root class and then computes the encryptions keys for all other classes in a level-order sequence. The computation of the encryption key for the root class is the most expensive process since the exponent of the modular exponentiation is a product of $m+1$ secret parameters. That is, the size of the exponent is $(m+1)|N|$. Thus, the number of required modular multiplications to compute the root's encryption key is about $1.5(m+1)|N|$. It requires about $1.5m\varphi$ modular multiplications to compute all remaining encryption keys in the hierarchy. On average, about $1.5(|N|+\varphi)$ modular multiplications are required to compute each encryption key.

The CA computes the user key $I(i,T)$ by taking advantage of some existing encryption keys. First, the CA chooses an already assigned encryption key $K_{i,\tau}$ where $\tau \in T$. It takes about $1.5|N||T|$ modular multiplications to compute $I(i,T)$ [16]. On the other hand, a user who has access to a user key $I(i,T)$ will be able to compute an encryption key $K_{\kappa,\tau}$, where $C_\kappa \preceq C_i$ and $\tau \in T$. The number of modular multiplications required for a key derivation is approximately $1.5\varphi(\alpha_i - \alpha_\kappa + |T| - 1)$ where $\alpha_i - \alpha_\kappa$ is the number of classes lower than or equal to C_i but higher than C_κ. Table 5.2 summarizes the computational complexity for the YS scheme.

	Item	# of modular mult.
CA	Per encryption key (in average)	$1.5(\lvert N\rvert + \varphi)$
	A user key $I(i,T)$	$1.5\lvert N\rvert\,\lvert T\rvert$
	Adding a time interval to a user key $I(i,T)$	$1.5\lvert N\rvert$
Users (of class C_i)	A key derivation: from $I(i,T)$ to $K_{\kappa,\tau}$	$1.5\varphi(\alpha_i - \alpha_\kappa + \lvert T\rvert - 1)$

Table 5.2: Computational Complexity for the proposed Yeh-Shyam (YS) scheme

CHAPTER 6

SETUP, ANALYSIS AND RESULTS

6.1 Implementation Overview

The WL and YS schemes have been implemented using the Java^{TM} Programming language specifically version 1.6. The Java platform provides a number of classes which provide support for cryptography, public key infrastructure (PKI), authentication, secure communication and access control. The Java Cryptography Architecture (JCA) is a core piece of the platform and contains "provider" architecture and a set of public accessible API's for digital signatures, hashs or message digests, certificates and certificate validation, encryption (symmetric, asymmetric both for block and stream ciphers), key generation and management and secure random number generation to name a few. For researchers in the cryptology field, these API's allow developers to easily design, integrate, re-use and call security API's from their application level programs with minimal custom extensions. Some of the Java classes which are core to this project include:

- `java.math.BigInteger` class provides methods which allow developers to compute operations such as two's-complement, modular arithmetic, GCD calculation, primality testing, prime number generation, bit manipulation and other miscellaneous operations.

- `java.security.SecureRandom` class provides a cryptographically strong random number generator. SecureRandom produces non-deterministic output and therefore any seed passed to a SecureRandom object must be unpredictable and all output sequences are cryptographically strong.
- `java.security.KeyPair` class is a simple holder for a key-pair (a public key and a private key) and does not enforce any security when initialized.
- `java.security.KeyPairGenerator` class is used to generate pairs of public and private keys for common algorithms such as RSA, DSA, etc.
- `java.security.KeyPairGeneratorSpi` defines a *Service Provider Interface* (SPI) for KeyPairGenerator class which is used to create custom pairs of public and private keys. All abstract methods must be implemented by each cryptographic service provider who wishes to supply the implementation of a key pair generator for a particular algorithm. The default modulus size (keysize) is 1024 bits.

6.2 Tools and Methodology

The following is the list of software packages used in the design and implementation of WL and YS schemes as well as documentation of this project:

- Java^{TM} JDK 6 Update 16 with NetBeans^{TM} 6.7.1: This distribution of the JDK includes the NetBeans IDE which is a powerful integrated development environment for developing applications in the Java platform.
- Java^{TM} Platform Standard Edition 6 API Specification
- TeXnicCenter: A feature rich easy to use IDE for creating LaTeX documents on the Windows platform

- MiKTeX: An up-to-date implementation of TeX and related programs for Windows

- A PC running Vista^{TM} operating system using Intel^{TM} processor with 2G RAM

- Microsoft Office 2007 Application suite specifically Excel and Powerpoint

The common four high level steps which captures the essence of WL and YS scheme are initialization, encyption key generation, user registration and key derivation. Appendix A describes the object oriented design details of the Java classes which were created namely AklTaylor, WangLaih, YehShyam, StopWatch, RSA and the core driver class called Main. The Main driver allows the user to input bit length which is used by the random number generator, the number of classes in a hierarchy as well as the time period. Experiments were conducted by varying these three vectors with the analysis provided in the following sections.

6.3 Storage Requirement Analysis

Tables 6.1, 6.2, 6.3 and 6.4 provides a summary of the storage requirements captured upon executing WL and YS schemes. In our examples below, we increase the time interval z = 4, 100, 500 and 1000. As expected, the storage requirement for WL scheme grows much faster than the YS scheme. For systems with very small number of time interval $z = 4$, we validated that the CA in the YS scheme indeed requires more storage than WL scheme. We needed 173 bits for YS scheme versus 96 bits for WL scheme for N, g and public primes. From a theoretical perspective, we only compare the dominant terms $(m+z)\varphi$ and $mz\varphi'$ in both schemes. The CA's storage requirement in the YS scheme is about $O(1/m^2)$ fraction of those in the WL scheme.

	Item	WL (bits)	YS (bits)
	N, g, public and secret parameters	96	173
CA	Encryption keys	148	162
Users (of class C_i)	$I(i,T), N$ and public parameters	103	66

Table 6.1: Storage bits measured for $m = 4$ and $z = 4$

	Item	WL (bits)	YS (bits)
	N, g, public and secret parameters	4119	2321
CA	Encryption keys	4290	4247
Users (of class C_i)	$I(i,T), N$ and public parameters	4125	870

Table 6.2: Storage bits measured for $m = 4$ and $z = 100$

	Item	WL (bits)	YS (bits)
	N, g, public and secret parameters	25827	12881
CA	Encryption keys	20920	22589
Users (of class C_i)	$I(i,T), N$ and public parameters	25833	5402

Table 6.3: Storage bits measured for $m = 4$ and $z = 500$

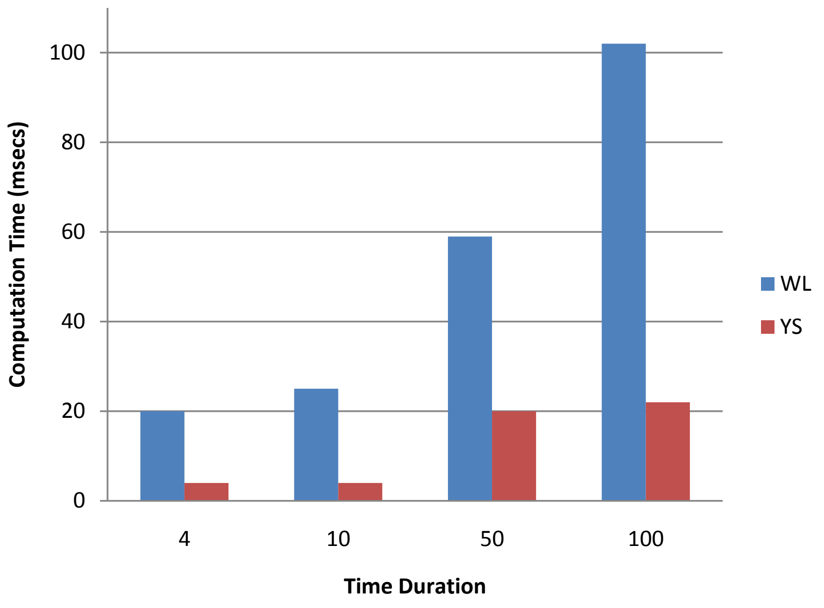
	Item	WL (bits)	YS (bits)
	N, g, public and secret parameters	56316	25540
CA	Encryption keys	44212	42223
Users (of class C_i)	$I(i,T), N$ and public parameters	56321	11848

Table 6.4: Storage bits measured for $m = 4$ and $z = 1000$

6.4 Performance Analysis

Figures 6.1 to 6.6 provides a summary of the performance requirements captured upon executing WL and YS schemes. Figures 6.1 to 6.5 compare the time needed to perform the initialization step. We varied the time period z values as 4, 10, 50 and 100 but kept $m = 4$ constant. In the second case, we varied the number of classes m with values 4, 8 and 16 and kept $z = 4$ constant. As evident from the figures, the WL scheme was much slower than YS scheme during the initialization step. This is due to the fact that the number of classes and time period was varied, the YS scheme requires $2(m + z)$ parameters versus mz parameter required by WL scheme. This mapped to the CA generating more prime numbers for the WL scheme during the initialization step. Other common initialization parameters such as p, q, N and g required the same amount of time to generate. For the YS scheme, the initialization step involved computing modular inverse of the generated prime numbers.

Fig 6.6 depicts all the four steps - initialization, encryption key generation, user registration and key derivation with $z = 4$ and a bit length of 1024 bits. In the top figure, Y-axis represents the computation time (in millisecs) for each of the four steps and in the bottom figure, we have represented the logarithm of the computation time. The experimental data is captured in Appendix A. As noted, the computation time for the initialization step was higher for WL scheme and increased with the number of classes m. The computation time needed to generate encryption keys was almost the same for both WL and YS schemes. The time for computing user registration shows an increasing trend for WL scheme. The computation time for key derivation was comparable for both WL and YS schemes.



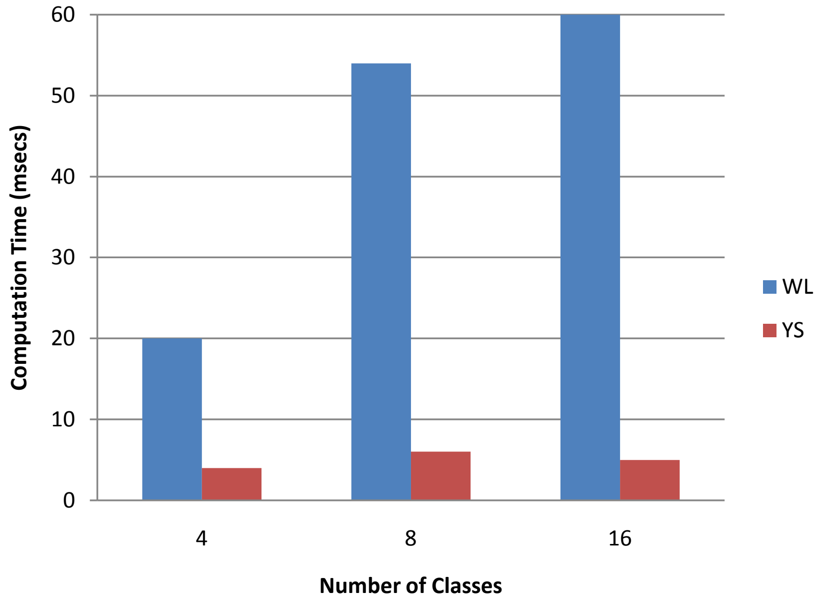
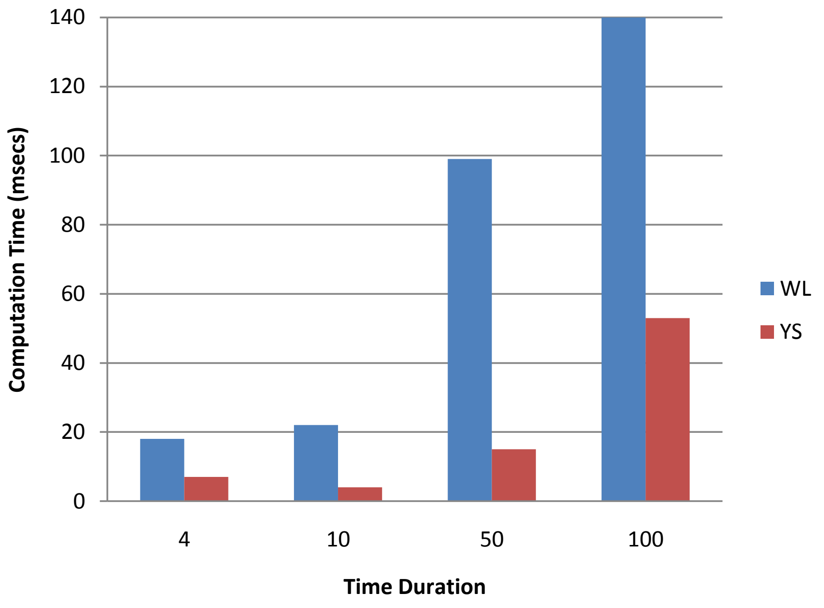


Figure 6.1: Comparison of Initialization step at bit length = 8. In the top figure, m = 4 while the number of time intervals was varied. In the bottom figure, z = 4 and the number of classes was varied



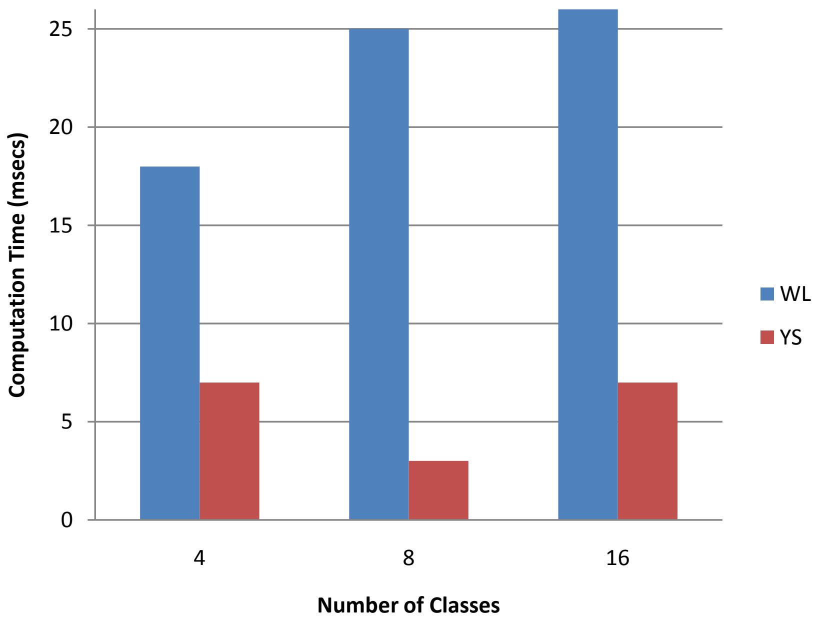
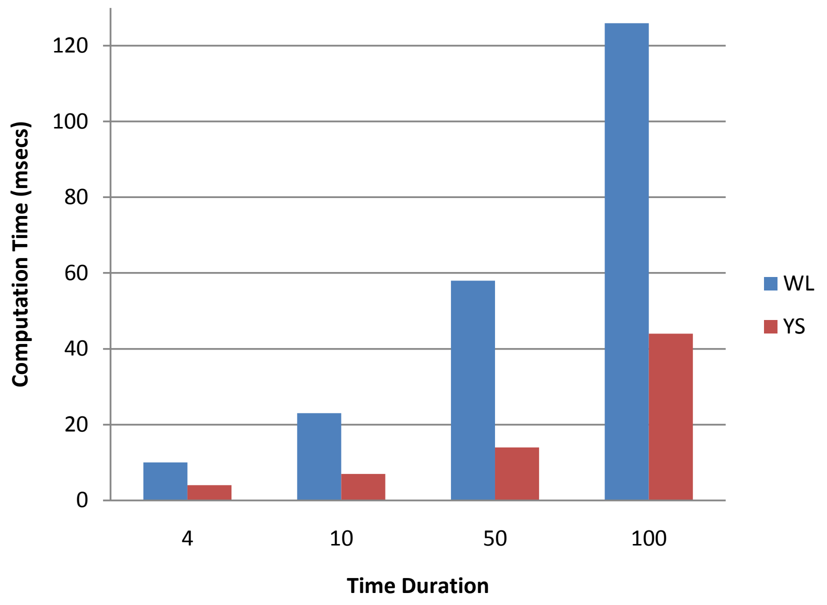


Figure 6.2: Comparison of Initialization step at bit length = 16. In the top figure, m = 4 while the number of time intervals was varied. In the bottom figure, z = 4 and the number of classes was varied



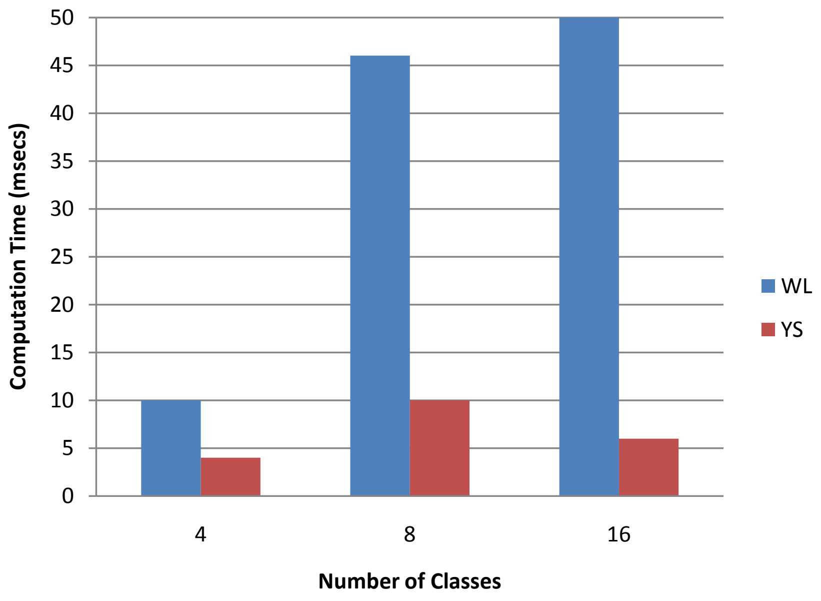
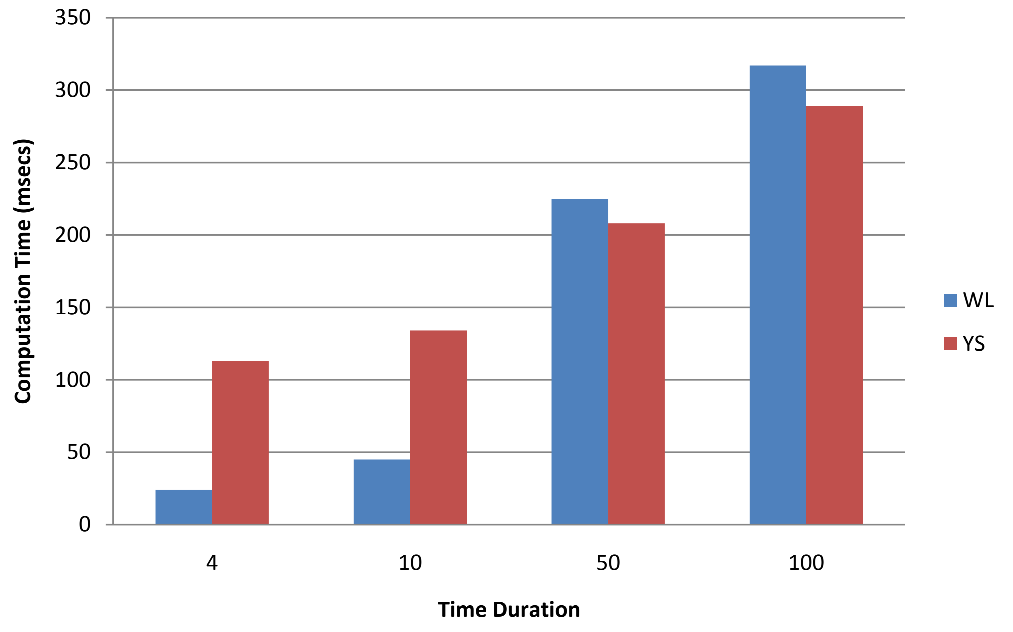


Figure 6.3: Comparison of Initialization step at bit length = 32. In the top figure, m = 4 while the number of time intervals was varied. In the bottom figure, z = 4 and the number of classes was varied



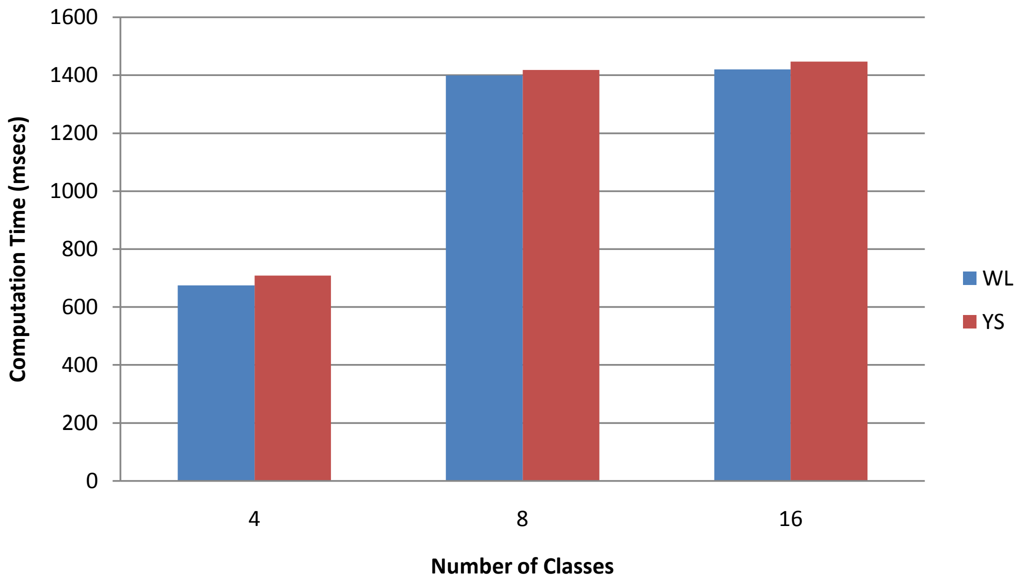
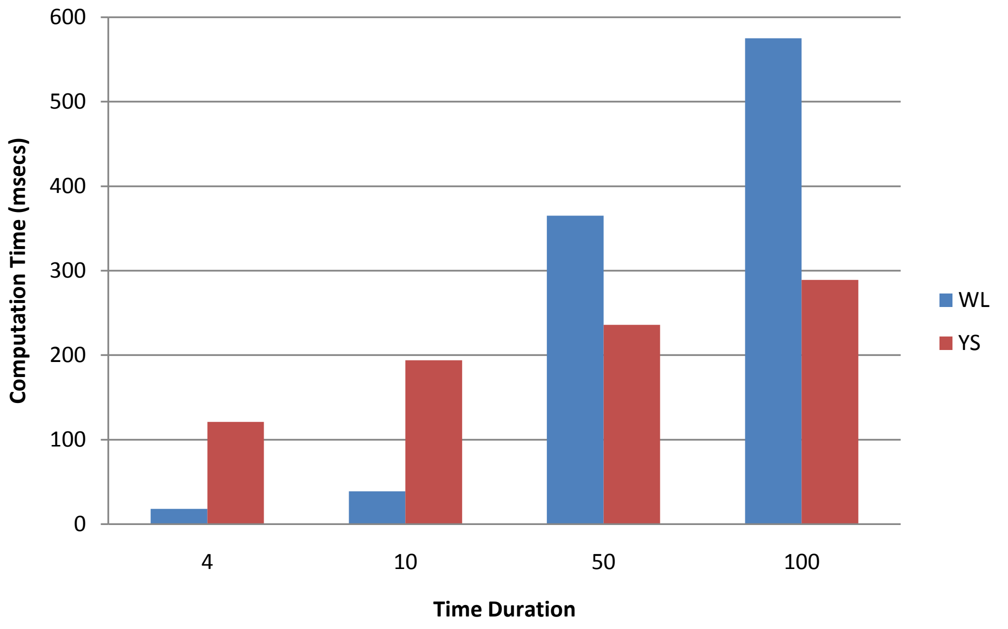


Figure 6.4: Comparison of Initialization step at bit length = 1024. In the top figure, $m = 4$ while the number of time intervals was varied. In the bottom figure, $z = 4$ and the number of classes was varied



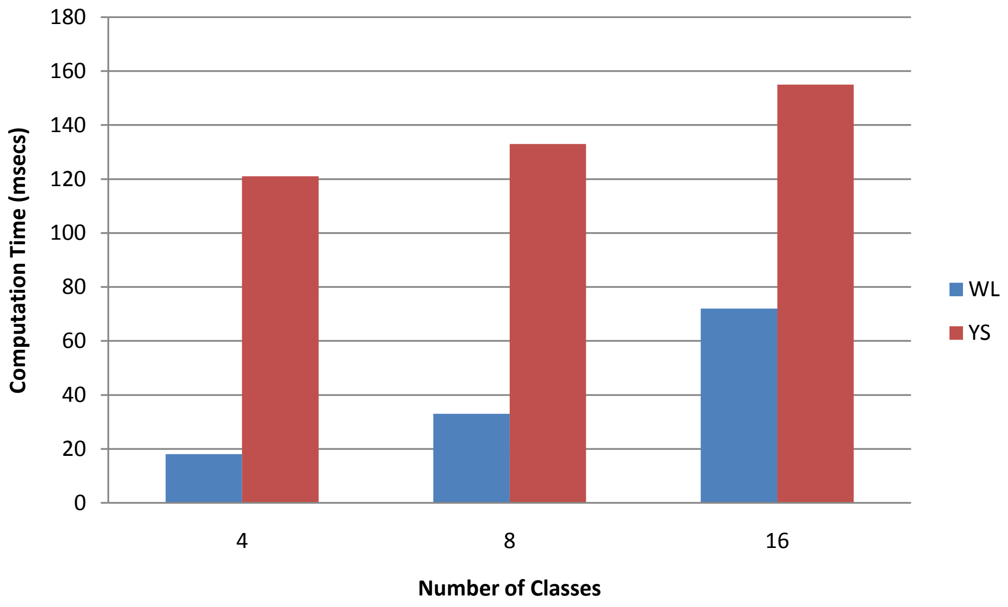
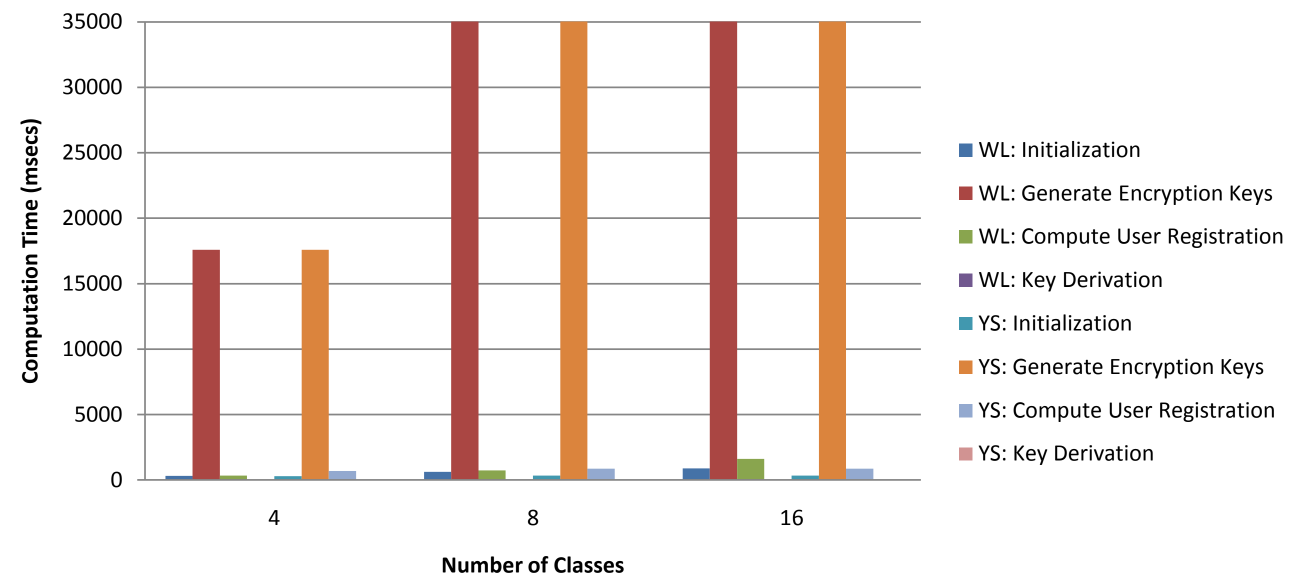


Figure 6.5: Comparison of Initialization step at bit length = 2048. In the top figure, $m = 4$ while the number of time intervals was varied. In the bottom figure, $z = 4$ and the number of classes was varied



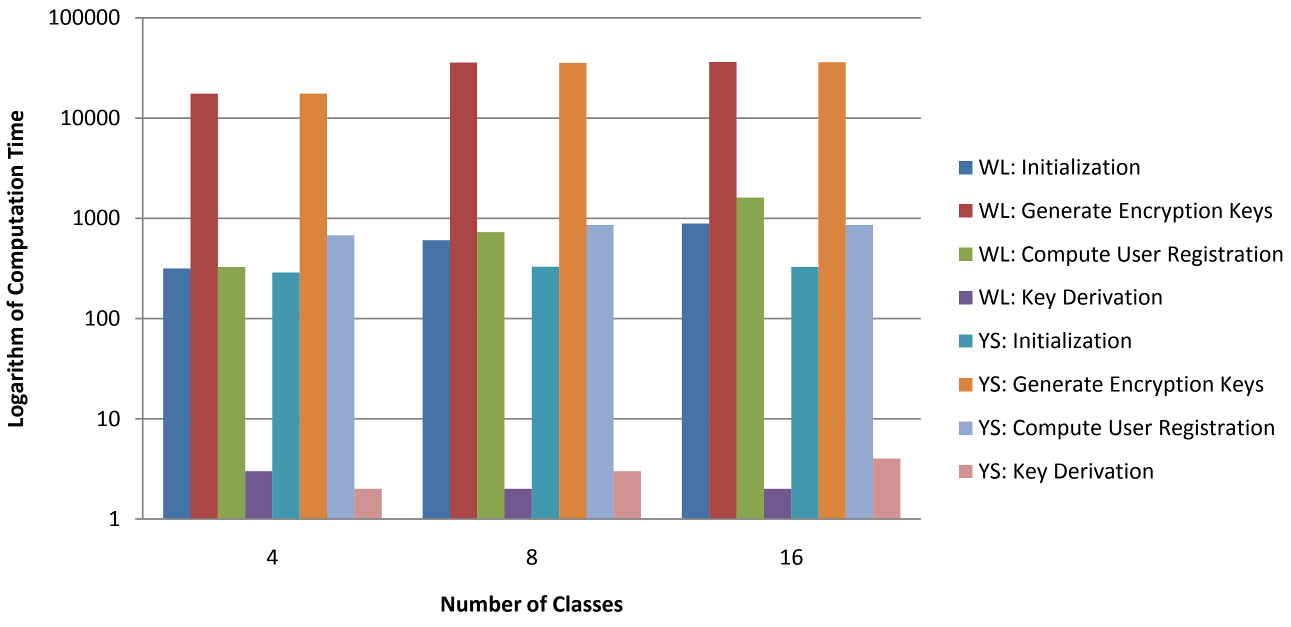


Figure 6.6: Comparison of all four steps specifically initialization, encryption key generation, user registration and key derivation with bit length = 1024 and m varied 4, 8 and 16. In the top figure, the Y-axis represents computation time (millisecs) and in the bottom figure, Y-axis represents log_{10} of computation time

CHAPTER 7

CONCLUSIONS

The time bound hierarchical key assignment first studied by Tzeng in 2002 was intended to solve the access control problem in a partially ordered hierarchy with time-dependent keys. Following Tzeng's scheme [13], two alternative schemes in [5] and [2] were proposed in 2004. Unfortunately, all of the above schemes have been proven insecure against collusive attacks in [17], [18]. In 2006, Wang and Laih proposed a new scheme [14] which applies Akl and Taylor's key assignment scheme [1] to a global hierarchy. The global hierarchy consists of z separate sub-hierarchies, one for each time interval. They also introduced the concept of merging. Any assigned key is just a merge, or in other words, a compression of a set of primitive keys such that the assigned key is able to recover any of its constituent primitive keys. We have defined our scheme also called the Yeh-Shyam scheme [16] including defining a new merge function. Our scheme requires less storage and performs better than the WL scheme.

This project has met all its goals:

- Described the theory and design steps for the two schemes
- Implemented the two schemes using the Java Programming language to generate the time-sensitive keys for a number of classes

- Measured the performance metrics of the two schemes, specifically storage and performance complexity

7.1 Recommendations for Future Research

This section points out few aspects of the theory and implementation which demand future research.

1. Perform cryptoanalysis of the Wang Laih scheme.
2. Implement, analyze and compare the performance of WL and YS scheme when classes are added and deleted at various places in the global hierarchy. There are specific procedures described in [16] related to addition and deletion of classes which can be implemented.
3. Implement the YS scheme in a real world scenario such as a client-server application.

REFERENCES

[1] S.G. Akl and P.D. Taylor. “Cryptographic Solution to a Problem of Access Control in a Hierarchy” *ACM Trans. Computer Systems*, vol. 1, no.3, pp. 239–248, 1983.

[2] L. Babinkostova. *Cryptology I Course Notes.* Boise State University, 2007.

[3] E. Bertino, N. Shang and S. Wagstaff. “An efficient time-bound hierarchical key management scheme for secure broadcasting” *IEEE Trans. Dependable and Secure Computing*, vol. 5, pp. 65-70, 2008.

[4] H.Y. Chien. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” *Comm. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[5] H.Y. Chien. “Efficient Time-Bound Hierarchical Key Assignment Scheme” *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no. 10, pp. 1302-1304, 2004.

[6] J. Crampton. “Lazy Updates in Key Assignment Schemes for Hierarchical Access Control” *Technical Report*, RHUL-MA-2006-11, University of London, 2006.

[7] H.F. Huang and C.C. Chang. “A New Cryptographic Key Assignment Scheme with Time-Constraint Access Control in a Hierarchy”, *Computer Standards and Interfaces*, vol. 26, pp.159-166, 2004.

[8] S.J. Mackinnon, P.D. Taylor, H. Meijer and S.G. Akl. “An Optimal Algorithm for Assigning Cryptographic Keys to Control Access in a Hierarchy” *IEEE Trans. Computers*, vol. C-34, no. 9, pp. 797–802, 1985.

[9] R. L. Rivest, A. Shamir and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems” *Comm. ACM*, vol. 21, no. 2, pp. 120-126, 1978.

[10] R.L. Rivest, A. Shamir and L. Adleman. “Efficient Time-Bound Hierarchical Key Assignment Scheme” *IEEE Trans. Knowledge and Data Engineering*, vol. 16, no. 10, pp. 1301–1304, 2004.

[11] R.S. Sandhu. “Cryptographic implementation of a tree hierarchy for access control” *Information Processing Letters*, 27 (1988), pp. 95-9.

[12] B. Schneier. *Applied Cryptography: Theory and Practice.* John Wiley and Sons, Inc., 1996.

[13] W.G. Tzeng. "A Time-Bound Cryptographic Key Assignment Scheme for Access Control in a Hierarchy". *IEEE Trans. Knowledge & Data Engineering*, vol. 14, no. 1, pp. 182–188, 2002.

[14] S.Y. Wang and C.S. Laih. "Merging: An Efficient Solution for a Time-Bound Hierarchical Key Assignment Scheme" *IEEE Trans. Dependable & Secure Computing*, vol. 3, no. 1, pp. 272–287, 2006.

[15] Q. Xu, M. He and L. Harn. "An improved time-bound hierarchical key assignment scheme" *IEEE Asia Pacific Services Computing Conference*, pp. 1489-1494, 2008.

[16] J.H. Yeh and R. Shyam. "An Efficient Time-Bound Hierarchical Key Assignment Scheme With a New Merge Function", submitted to *IEEE Trans. on Computers.*

[17] X. Yi. "Security of Chien's efficient time-bound hierarchical key assignment scheme" *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 9, pp. 1298-1299, 2005.

[18] X. Yi and Y. Ye. "Security of Tzeng's time-bound key assignment scheme for access control in a hierarchy" *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 4, pp. 1054-1055, 2003.

APPENDIX A

EXPERIMENTAL DATA

Bit Length	m	z	**Initialization**
8	4	4	20
8	4	10	25
8	4	50	59
8	4	100	102
8	8	4	54
8	16	4	60
16	4	4	18
16	4	10	22
16	4	50	99
16	4	100	140
16	8	4	25
16	16	4	26
32	4	4	10
32	4	10	23
32	4	50	58
32	4	100	126
32	8	4	46
32	16	4	52
1024	4	4	24
1024	4	10	45
1024	4	50	225
1024	4	100	317
1024	8	4	33
1024	16	4	72
2048	4	4	18
2048	4	10	39
2048	4	50	365
2048	4	100	575
2048	8	4	33
2048	16	4	72

Table A.1: WL Computation Time (in millisecs) for Initialization step in Figs 6.1-6.5

m	**Initialization**	**Encryption Keys**	**User Registration**	**Key Derivation**
4	317	17580	326	3
8	606	35773	726	2
16	887	36230	1605	2

Table A.2: WL Computation Time (in millisecs) with bit length = 1024 where z = 100 for data in Fig 6.6

Bit Length	m	z	**Initialization**
8	4	4	4
8	4	10	4
8	4	50	20
8	4	100	22
8	8	4	6
8	16	4	5
16	4	4	7
16	4	10	4
16	4	50	15
16	4	100	53
16	8	4	3
16	16	4	7
32	4	4	4
32	4	10	7
32	4	50	14
32	4	100	44
32	8	4	10
32	16	4	6
1024	4	4	113
1024	4	10	134
1024	4	50	208
1024	4	100	289
1024	8	4	133
1024	16	4	155
2048	4	4	121
2048	4	10	194
2048	4	50	236
2048	4	100	289
2048	8	4	133
2048	16	4	155

Table A.3: YS Computation Time (in millisecs) for Initialization step in Figs 6.1-6.5

m	**Initialization**	**Encryption Keys**	**User Registration**	**Key Derivation**
4	289	17583	678	2
8	330	35696	861	3
16	327	36168	860	4

Table A.4: YS Computation Time (in millisecs) with bit length = 1024 where z = 100 for data in Fig 6.6

APPENDIX B

PACKAGE RSA

Package Contents *Page*

B.1 Classes

B.1.1 Class **AklTaylor**

This class implements the Akl Taylor key generation scheme

Declaration

public class AklTaylor
extends java.lang.Object

Constructors

- *AklTaylor*

  `public` **AklTaylor**`( int` **bitlen**`, int` **class_size** `)`

  - **Usage**
    - This is the constructor where we instantiate objects and allocate memory for p, q, N, g, e, K
  - **Parameters**
    - `bitlen` - is input to generate keys for the passed bit lengths
    - `class_size` - is the hierarchical depth

Methods

- *displayKeys*

  `public void` **displayKeys**`( )`

  - **Usage**
    - displayKeys is a method which displays all the keys scheme

---

- *displayValue*

  `private String` **displayValue**`( java.math.BigInteger` **val** `)`

  - **Usage**
    - displayValue method displays the value of a selected key as a string
  - **Parameters**
    - `val` - is the key value

- *generateKeys*

  `public void` **generateKeys**`( )`

  - **Usage**
    - generateKeys is the method which is called by the Central Authority to generate keys

---

- *generateKeysFrom*

  `public void` **generateKeysFrom**`( int` **source**`, int` **destination** `)`

  - **Usage**
    - generateKeysFrom method generates a key for a class lower in the hierarchy from a key higher in the class hierarchy

- **Parameters**
    - `source` - is a class in the hierarchy which is above the class for which the key needs to be calculated
    - `destination` - is the class for which the key is calculated using the key of a class which is higher in the hierarchy

---

- *setSize*

  `private void` **setSize**`( int` **class_size** `)`

  - **Usage**
      - setSize method is a private method which copies the user input class size to an internal variable
  - **Parameters**
      - `class_size` - is the number of classes in the hierarchy

B.1.2 Class **Main**

Main driver class which is entry point for users who exercise the key generation algorithms.

Declaration

---

public class Main

extends java.lang.Object

Constructors

- *Main*

  `public` **`Main`**`( )`

METHODS

- *main*

  `public static void` **`main`**`( java.lang.String []` **`args`** `)`

  - **Usage**
    - Main method which instantiates various objects which are the Key Generation schemes (WL and YS) and RSA Cryptographic Encryption/Decryption Algorithm.
  - **Parameters**
    - `args` - the command line arguments

B.1.3 CLASS **RSA**

This class implements the RSA encryption and decryption technique

DECLARATION

---

public class RSA

extends java.lang.Object

FIELDS

- private BigInteger n
  -
- private BigInteger d
  -
- private BigInteger e
  -

Constructors

- *RSA*

  `public` **RSA**`( int` **bitlen** `)`

  - **Usage**
    * This is constructor for the RSA class
  - **Parameters**
    * `bitlen` - is the bit length at which the RSA scheme is executed

Methods

- *decrypt*

  `public BigInteger` **decrypt**`( java.math.BigInteger` **message** `)`

  - **Usage**
    * Decrypt method will decrypt the message which is passed to it
  - **Parameters**

    * `message` - is the data which needs to be decrypted

  - **Returns** - Decrypted message in BigInteger format

- *encrypt*

  `public BigInteger` **`encrypt`**`( java.math.BigInteger` **`message`** `)`

  - **Usage**
    * Encrypt method will take an ASCII message and encrypt it
  - **Parameters**
    * `message` - is the ASCII data which needs to be encrypted
  - **Returns** - Encrypted message in BigInteger format

- *toString*

  `public String` **`toString`**`( )`

  - **Usage**
    * This method prints the public and private information to the screen
  - **Returns** - Returns the concatenated string

B.1.4 Class **StopWatch**

The StopWatch class provides methods which will be used as a wrapper to measure the execution time for key generation algorithms.

Declaration

public class StopWatch

extends java.lang.Object

Fields

- private long startTime
  -
- private long stopTime
  -
- private boolean running
  -

Constructors

- *StopWatch*

  `public` **StopWatch**`( )`

Methods

- *getElapsedTimeInMilliSecs*

  `public long` **getElapsedTimeInMilliSecs**`( )`

  - **Usage**
    * The getElapsedTime method calculates the elapsed time by taking the difference between the end time and the start time.
  - **Returns** - Returns the elapsed time in milli seconds

- *getElapsedTimeSecs*

  `public long` **`getElapsedTimeSecs`**`( )`

  - **Usage**
    * The getElapsedTime method calculates the elapsed time by taking the difference between the end time and the start time.
  - **Returns** - Returns the elapsed time in seconds

- *start*

  `public void` **`start`**`( )`

  - **Usage**
    * The start method synchronizes with the system clock and gets the clock counting.

- *stop*

  `public void` **`stop`**`( )`

  - **Usage**
    * The stop method notes the end time and stops the stop watch from running further.

B.1.5 Class **WangLaih**

This class implements the Wang-Laih (WL) key generation scheme

Declaration

public class WangLaih

extends java.lang.Object

Fields

- private int bit_length
  -
- public BigInteger N
  -
- private BigInteger p
  -
- private BigInteger q
  -
- private BigInteger g
  -

- public BigInteger epsilon
  -
- private BigInteger e
  -
- private BigInteger bigE
  -
- public BigInteger K
  -
- private BigInteger Kprime
  -
- public int size
  -
- public int duration
  -
- private BigInteger alpha
  -
- private int class_rank
  -
- private BigInteger bigI
  -

- private BigInteger bigK
  –
- private BigInteger E_alpha
  –
- private int p_size
  –
- private int q_size
  –
- private int N_size
  –
- private int g_size
  –
- private int bigI_size
  –
- private int bigK_size
  –
- private int Kprime_avg_size
  –
- private int K_avg_size
  –

- private int epsilon_avg_size
  -
- private int e_avg_size
  -

Constructors

- *WangLaih*

  `public` **WangLaih**`( int` **bitlen**`, int` **class_size**`, int` **time_duration**`, boolean` **isDebug** `)`

  - **Usage**
    * This is the constructor where we instantiate objects and allocate memory for p, q, N, g, e, K and Kprime
  - **Parameters**
    * `bitlen` - is input to generate keys for the passed bit lengths
    * `class_size` - is the hierarchical class depth
    * `time_duration` - is the time duration
    * `isDebug` - enables the global debug flag

Methods

- *displayKeys*

  `public void` **displayKeys**`( )`

  - **Usage**

    * displayKeys is a method which displays all the keys generated by Wang Laih scheme

---

- *displayValue*

  `private String` **displayValue**`( java.math.BigInteger` **val** `)`

  - **Usage**
    * displayValue method displays the value of a selected key as a string
  - **Parameters**
    * `val` - is the key value
  - **Returns** - Returns the string value of val

- *generateEncryptionKeys*

  `private void` **generateEncryptionKeys**`( )`

  - **Usage**
    * generateEncryptionKeys is the method which is called by the Central Authority to generate encryption keys. This method uses the WL merge technique to merge the primitive keys of classes in the hierarchy to generate the aggregate key.

- *generateKeys*

  `public void` **generateKeys**`( )`

  - **Usage**
    * generatKeys is the method which is called by the Central Authority to generate Wang Laih private key methods

---

- *generatePrimitiveKeys*

  `private void` **generatePrimitiveKeys**`( )`

  - **Usage**
    - generatePrimitiveKeys method executes the steps to generate primitive keys

---

- *keyDerivation*

  `private void` **keyDerivation**`( int` **which_class**`, int` **time_slot** `)`

  - **Usage**
    - keyDerivation is the method which is called by the Central Authority to derive keys
  - **Parameters**
    - `which_class` - is the class for which key needs to be derived
    - `time_slot` - is the time slot for which key needs to be derived

---

- *setBitLength*

  `private void` **setBitLength**`( int` **bitlen** `)`

  - **Usage**
    - setBitLength method is a private method which copies the user bit length to an internal variable
  - **Parameters**
    - `bitlen` - is the bit length

- *setDebugFlag*

  `private void` **setDebugFlag**`( boolean` **isDebug** `)`

  - **Usage**
    - setDebugFlag enables/disables global debug flag for WL scheme.
  - **Parameters**
    - `isDebug` - is the debug flag passed by the main class.

- *setDuration*

  `private void` **setDuration**`( int` **time_duration** `)`

  - **Usage**
    - setDuration method is a private method which copies the user input time duration to an internal variable
  - **Parameters**
    - `time_duration` - is the number of time slots

- *setSize*

  `private void` **setSize**`( int` **class_size** `)`

  - **Usage**
    - setSize method is a private method which copies the user input class size to an internal variable
  - **Parameters**
    - `class_size` - is the number of classes in the hierarchy

- *userRegistration*

  `private void` **userRegistration**`( int` **which_class**`, int` **t_from**`, int` **t_to** `)`

  - **Usage**
    - userRegistration is the method which is called by the Central Authority to calculate user registrations I(i,T). If a user is assigned to a class Ci for a set of time intervals T, the CA gives the user an aggregate key I(i,T) as a user key by computing I(i,T)
  - **Parameters**
    - `which_class` - is the class for which user registration needs to be computed
    - `t_from` - is the time slot to start
    - `t_to` - is the time slot to end

B.1.6 Class **YehShyam**

This class implements the Yeh-Shyam (YS) key generation scheme

Declaration

public class YehShyam

extends java.lang.Object

Fields

- private int bit_length
  -
- public BigInteger N
  -
- private BigInteger p
  -
- private BigInteger q
  -
- private BigInteger g
  -
- private BigInteger phi_N
  -
- private BigInteger ec
  -
- private BigInteger dc
  -
- private BigInteger et
  -
- private BigInteger dt
  -

- public BigInteger epsilon
  -
- private BigInteger e
  -
- private BigInteger bigE
  -
- public BigInteger K
  -
- private BigInteger Kprime
  -
- public int size
  -
- public int duration
  -
- private BigInteger alpha
  -
- private int class_rank
  -
- private BigInteger bigI
  -

- private BigInteger bigK
  –
- private BigInteger E_alpha
  –
- private int p_size
  –
- private int q_size
  –
- private int N_size
  –
- private int g_size
  –
- private int bigI_size
  –
- private int bigK_size
  –
- private int Kprime_avg_size
  –
- private int K_avg_size
  –

- private int ec_avg_size
  –
- private int dc_avg_size
  –
- private int et_avg_size
  –
- private int dt_avg_size
  –

Constructors

- *YehShyam*

  `public` **YehShyam(** `int` **bitlen**, `int` **class_size**, `int` **time_duration**, `boolean` **isDebug**)

  – **Usage**
    - This is the constructor where we instantiate objects and allocate memory for p, q, N, g, e, K and Kprime

  – **Parameters**
    - `bitlen` - is input to generate keys for the passed bit lengths
    - `class_size` - is the hierarchical class depth
    - `time_duration` - is the time duration
    - `isDebug` - enables/disables the global debug flag

Methods

- *displayKeys*

  `public void` **displayKeys**`( )`

  - **Usage**
    - displayKeys is a method which displays all the keys generated by YS scheme

---

- *displayValue*

  `private String` **displayValue**`( java.math.BigInteger` **val** `)`

  - **Usage**
    - displayValue method displays the value of a selected key as a string
  - **Parameters**
    - `val` - is the key value
  - **Returns** - String for val in string format

- *generateEncryptionKeys*

  `private void` **generateEncryptionKeys**`( )`

  - **Usage**
    - generateEncryptionKeys is the method which is called by the Central Authority to generate encryption keys. This method uses the YS merge technique to merge the primitive keys in a class hierarchy to generate the aggregate keys.

- *generateKeys*

  `public void` **generateKeys**`( )`

  - **Usage**
    - generateKeys is the method which is called by the Central Authority to generate Yeh Shyam keys

---

- *generatePrimitiveKeys*

  `private void` **generatePrimitiveKeys**`( )`

  - **Usage**
    - generatePrimitiveKeys method executes the steps to generate primitive keys

---

- *keyDerivation*

  `private void` **keyDerivation**`( int` **which_class**`, int` **time_slot**`, int` **bigi_class**`, int` **bigi_t_from**`, int` **bigi_t_to** `)`

  - **Usage**
    - keyDerivation is the method which is called by the Central Authority to derive keys
  - **Parameters**
    - `which_class` - is the class for which key needs to be derived
    - `time_slot` - is the time slot for which key needs to be derived
    - `bigi_class` - is the class information from aggregate key
    - `bigi_t_from` - is the starting time slot from aggregate key

    * `bigi_t_to` - is the ending time slot from aggregate key

- *randomModValue*

  `private static BigInteger` **randomModValue**`( java.math.BigInteger t, java.math.BigInteger` **prev** `)`

  - **Usage**
    * randomModValue method finds the next prime number to be used in ec and dc. It ensures that the generated number is prime to phi(N)
  - **Parameters**
    * `t` - is phi_N
    * `prev` - is the previously used prime number in the sequence.

- *setBitLength*

  `private void` **setBitLength**`( int` **bitlen** `)`

  - **Usage**
    * setBitLength method is a private method which copies the user bit length to an internal variable
  - **Parameters**
    * `bitlen` - is the bit length for computation

- *setDebugFlag*

  `private void` **setDebugFlag**`( boolean` **isDebug** `)`

  - **Usage**
    * setDebugFlag enables/disables global debug flag for WL scheme.

- **Parameters**
  - `isDebug` - is the debug flag passed by the main class.

- *setDuration*

  `private void` **setDuration**`( int` **time_duration** `)`

  - **Usage**
    - setDuration method is a private method which copies the user input time duration to an internal variable
  - **Parameters**
    - `time_duration` - is the number of time slots

- *setSize*

  `private void` **setSize**`( int` **class_size** `)`

  - **Usage**
    - setSize method is a private method which copies the user input class size to an internal variable
  - **Parameters**
    - `class_size` - is the number of classes in the hierarchy

- *userRegistration*

  `private void` **userRegistration**`( int` **which_class**`, int` **t_from**`, int` **t_to** `)`

  - **Usage**

  * userRegistration is the method which is called by the Central Authority to calculate user registrations I(i,T). If a user is assigned to a class Ci for a set of time intervals T, the CA gives the user an aggregate key I(i,T) as a user key by computing I(i,T)

- **Parameters**
  * `which_class` - is the class for which user registration is performed
  * `t_from` - is the starting time slot
  * `t_to` - is the ending time slot