Israel de Souza Carlos

Implementação de uma Biblioteca de Segredo Compartilhado

Florianópolis (SC), Brasil
2013/2

Israel de Souza Carlos

Implementação de uma Biblioteca de Segredo Compartilhado

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Santa Catarina - UFSC

Departamento de Informática e Estatística - INE

Orientador: Prof. Dr. Jean Everson Martina

Coorientador: Armindo Antonio Guerra Júnior

Florianópolis (SC), Brasil

2013/2

Israel de Souza Carlos

Implementação de uma Biblioteca de Segredo Compartilhado

Este Trabalho de conclusão de curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação" e aprovado em sua forma final pelo Curso de Ciência da Computação.

Florianópolis (SC), 01 de Novembro de 2013.

Prof. Dr. Vitório Bruno Mazzola
Coordenador

Banca Examinadora:

Prof. Dr. Jean Everson Martina
Orientador

Armindo Antonio Guerra Júnior
Coorientador

Prof. Dr. Ricardo Felipe Custódio

M.Sc. Dayana Pierina Brustolin Spagnuelo

Dedicado este trabalho aos meus pais, por todo amor, apoio e compreensão. Sei que esse sonho também é de vocês.

Agradecimentos

Primeiramente agradeço a Deus, o centro e o fundamento de tudo em minha vida, por renovar a cada momento a minha força e disposição, e pelo discernimento concedido ao longo dessa jornada.

A toda minha família pelo apoio e incentivo durante toda minha fase de graduação. Muito obrigado pelo carinho, por sempre me incentivarem dando forças mesmo que distantes.

Em especial, agradeço a meus pais por terem possibilitado a realização deste sonho. Obrigado por terem me dado toda a estrutura, pelas orações em meu favor, pela força e incentivo que foram fundamentais para que eu chegasse até aqui. Sem vocês, com toda certeza, nada disso teria sido possível.

Agradeço ainda aos meus amigos do LabSEC que contribuíram direta ou indiretamente com este trabalho. Em especial, aos professores Ricardo Felipe Custódio e Jean Everson Martina pela orientação ao longo da realização deste trabalho.

Por fim, agradeço aos meus amigos de uma forma geral. Vocês não contribuíram diretamente com este trabalho mas foram fundamentais ao longo dessa jornada.

Aprender é a unica coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende.

Leonardo da Vinci

Resumo

Na operação de um Módulo de Segurança Criptográfico por vezes é necessário que mais de uma pessoa autorize a realização de uma determinada tarefa para que a mesma possa ser executada como forma de garantir a segurança do conteúdo sensível nele armazenado. Essa é uma questão que pode ser resolvida através da utilização de mecanismos de segredo compartilhado.

Além da aplicação supracitada, mecanismos de segredo compartilhado têm aplicação em diversos outros contextos. Uma biblioteca de software que implemente mecanismos desse tipo é de grande importância para várias aplicações na área de segurança.

Este trabalho suprirá uma necessidade de implementação de alguns projetos desenvolvidos e mantidos pelo Laboratório de Segurança em Computação (LabSEC) que fazem uso de uma biblioteca de segredo compartilhado restrita, cujo código não foi desenvolvido no contexto do laboratório. Sendo assim, com o desenvolvimento de uma biblioteca desse tipo localmente será possível alcançar um maior controle sobre os sistemas desenvolvidos pelo LabSEC.

Como resultado deste trabalho, desenvolveu-se uma biblioteca de software na linguagem de programação Java que implementa dois mecanismos de segredo compartilhado distintos: os esquemas de segredo compartilhado de Shamir, e Blakley, respectivamente.

Palavras-chaves: Segredo Compartilhado, Biblioteca, Módulo de Segurança Criptográfico.

Abstract

The operation of a Cryptographic Security Module sometimes needs more than one person to authorize the execution of a particular task in order to ensure the security of sensitive content stored therein. This is an issue that can be solved through the use of secret sharing mechanisms.

In addition to the aforementioned application, secret sharing mechanisms has their uses in numerous other contexts. A software library that implements such mechanisms is of great importance for various applications in the security area.

This work will fill a need for the implementation of some projects developed and maintained by The Laboratory for Computer Security (LabSec) that makes use of a restricted secret sharing library, the code of which was not been developed in the context of the laboratory. Thus, with the local development of such a library, the LabSEC will have greater control over its own developed systems.

As a result of this work, we have developed a software library in Java programming language that implements two different secret sharing mechanisms: Shamir's Secret Sharing and Blakley's Secret Sharing, respectively.

Key-words: Secret Sharing, Library, Hardware Security Module.

Lista de ilustrações

Lista de abreviaturas e siglas

AC	Autoridade Certificadora
AES	*Advanced Encryption Alghorithm*
API	*Application Programming Interface*
ASI-HSM	*Advanced Security Initiative - Hardware Security Module*
DES	*Decryption and Encryption Standard*
HSM	*Hardware Security Module*
ICP	Infraestrutura de Chaves Públicas
ICPEdu	Infraestrutura de Chaves Públicas para Ensino e Pesquisa
IDE	*Integrated Development Environment*
INE	Departamento de Informática e Estatística
JVM	*Java Virtual Machine*
LabSEC	Laboratório de Segurança em Computação
PIN	*Personal Identification Number*
RNP	Rede Nacional de Ensino e Pesquisa
RSA	Rivest, Shamir e Adleman
SHA	*Secure Hash Algorithm*
SVN	*Subversion*
UFSC	Universidade Federal de Santa Catarina
UML	*Unified Modeling Language*

Lista de símbolos

$\neq$	Diferente
$\geq$	Maior ou igual que
$>$	Maior que
$\leq$	Menor ou igual que
$<$	Menor que
$\in$	Pertence
$\prod$	Produtório
$\sum$	Somatório

Sumário

1 Introdução

1.1 Contextualização

Segredo compartilhado, do inglês *secret sharing*, é uma importante ferramenta na área de segurança computacional e criptografia. É comum nessas áreas a existência de uma única *master secret* que provê acesso a uma série de informações ou recursos sigilosos. Nesse caso, é desejável que a *master secret* seja armazenada de forma segura afim de evitar qualquer exposição da mesma, seja acidentalmente ou por meios maliciosos. Se uma *master secret* é perdida ou destruída, então todas as informações acessadas através da mesma tornam-se indisponíveis. Esse é um cenário típico onde o método do segredo compartilhado pode ser utilizado [1].

Esquemas de segredo compartilhado são ideais para o armazenamento de informações de alta importância e sensibilidade. Como exemplos dessas informações podemos citar: chaves criptográficas, códigos para lançamento de mísseis, entre outras. Esse tipo de informação requer armazenamento altamente confiável, pois se expostas podem provocar resultados desastrosos. Além disso, é também crítico que estas informações não sejam perdidas.

Métodos tradicionais de criptografia, por sua vez, não são adequados para a proteção de informação de alta sensibilidade. Isso acontece porque os mesmos não são capazes de atingir elevados níveis de confidencialidade e confiabilidade simultaneamente, haja vista que ao armazenar uma chave criptográfica, por exemplo, é preciso escolher entre armazenar uma única cópia da chave para obter máxima segurança ou manter diferentes cópias da chave em diferentes lugares para maior confiabilidade. Aumentar a confiabilidade de uma chave criptográfica mantendo múltiplas cópias da mesma pode diminuir sua confidencialidade à medida que cria novas opções de ataque; há mais oportunidades que uma das cópias venha a cair em mãos erradas.

A melhor solução para o caso de uma *master secret* seria a utilização de mecanismos de segredo compartilhado. Segredo compartilhado refere-se ao método de distribuir um segredo entre um grupo de participantes, onde cada um destes fica em posse de uma parte do segredo, também chamada *share*. De acordo com um dos métodos mais tradicionais de segredo compartilhado, o segredo pode ser reconstruído apenas se um número suficiente de participantes, definido previamente, combinarem em conjunto suas respectivas partes do segredo. Partes individuais do segredo ou ainda a combinação de um número insuficiente de *shares* não fornecem nenhuma informação a respeito do segredo original.

Por isso a importância da implementação de mecanismos de segredo comparti-

lhado, pois tratam-se de esquemas capazes de resolver o problema citado anteriomente, permitindo que se alcance simultaneamente altos níveis de confiabilidade e confidencialidade de informações de sigilosas. Esquemas de segredo compartilhado são muito úteis: no gerenciamento de chaves criptográficas, tarefa realizada, por exemplo, através de um Módulo de Segurança Criptográfico; e em *multiparty secure protocols* [1].

Um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM) é dispositivo composto por um hardware, software e *firmware* que possui a finalidade de prover proteção e gerência ao ciclo de vida de chaves criptográficas [2]. Para a execução das funções de um HSM é necessária a autenticação em conjunto de um número mínimo de indivíduos habilitados a realizar a função desejada. Normalmente isto é realizado através da criação de grupos de usuários que estarão habilitados a realizar operações específicas no HSM. A autenticação dos indivíduos do grupo é possível através da utilização de mecanismos de segredo compartilhado.

Podemos dar um exemplo de caso na história onde mecanismos de segredo compartilhado foram utilizados: de acordo com *Time Magazine*[1], o controle de armas nucleares na Rússia no início da decada de 1990 dependia de um mecanismo de acesso popularmente chamado "dois de três". Este termo significa que haviam três partes envolvidas no controle das armas nucleares, sendo que sempre duas partes deveriam autenticar-se em conjunto para a realização de qualquer operação de controle sobre os armamentos. Nesse caso específico, as três partes envolvidas eram o Presidente do país, o Ministro da Defesa e o Ministério de Defesa [3].

O presente projeto objetiva a implementação de uma biblioteca de software que implemente dois mecanismos de segredo compartilhado distintos. Foram escolhidos para implementação dois mecanismos de segredo compartilhado baseados em *threshold*: o esquema de Shamir e o esquema de Blakley, detalhadamente explicados na Capítulo 4.

1.2 Objetivos

Os objetivos do presente trabalho foram subdivididos em objetivo geral e objetivos específicos, como apresentado a seguir.

1.2.1 Objetivo Geral

O objetivo geral do presente trabalho é o desenvolvimento de uma biblioteca de software que implemente mecanismos de segredo compartilhado de forma reconfigurável e reutilizável.

[1] Time Magazine, 4 de Maio de 1992, p. 13

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral é necessário que os seguintes objetivos específicos sejam atingidos:

- Construir um referencial teórico sobre o tema;
- Realizar um levantamento do que possa já existir implementado sobre segredo compartilhado;
- Modelar a biblioteca a ser implementada utilizando diagramas da UML adequados;
- Desenvolver a biblioteca;
- Realizar a análise e teste da biblioteca implementada.

1.3 Justificativa e Motivação

O desenvolvimento de uma biblioteca que implemente mecanismos de segredo compartilhado é importante por alguns motivos. O primeiro deles é que este trabalho servirá como uma prova de conceito de que é possível a implementação de uma biblioteca que implemente mecanismos de segredo compartilhado de forma reconfigurável e reutilizável. Uma segunda razão que justifica a importância deste trabalho é a inexistência de uma biblioteca padrão confiável que implemente diferentes mecanismos de segredo compartilhado. O que existe são sempre implementações pontuais. Pretendemos que a bibliteca desenvolvida possa atender a essa necessidade.

A existência de uma bilioteca desse tipo também suprirá uma necessidade de implementação de vários trabalhos do Laboratório em Segurança da Computação (LabSEC)[2] da Universidade Federal de Santa Catarina (UFSC)[3], local em que este trabalho foi realizado. Há projetos dentro do laboratório que necessitam de implementações de mecanismos de segredo compartilhado. No entanto, os mesmos fazem uso de uma biblioteca restrita sobre a qual não há mais suporte ou documentação, e cujo código não foi desenvolvido no contexto do laboratório. O desenvolvimento da biblioteca proposta nesse trabalho permitirá, quando esta passar a ser utilizada pelos sistemas desenvolvidos pelo LabSEC, que exista um maior controle sobre os mesmos.

1.4 Metodologia

Inicialmente, pretende-se fazer um estudo minucioso teórico com foco nos principais mecanismos de segredo compartilhado existentes, assim como uma pesquisa detalhada

[2] http://www.labsec.ufsc.br/
[3] http://ufsc.br/

para levantamento do que já possa existir implementado sobre segredo compartilhado. Com isso, pretende-se construir um referencial teórico completo sobre as informações necessárias ao desenvolvimento do trabalho.

Após a primeira etapa, serão levantados e documentados os requisitos que precisarão ser atendidos pelo sistema a ser implementado. Posteriomente, será realizado a modelagem do sistema com a utilização de diagramas UML, do inglês *Unified Modeling Language*.

A partir do levantamento de requisitos e modelagem do sistema, o mesmo será desenvolvido utilizando a linguagem de programação *Java*. Desenvolvido o código, o mesmo será testado e avaliado para que seja assegurado o correto funcionamento e segurança das rotinas implementadas.

Cumpridos os objetivos de implementação da biblioteca, dois mecanismos de segredo compartilhado distintos terão sido implementados e testados.

1.5 Estrutura do Documento

Este trabalho foi estruturado conforme descrito a seguir.

No capítulo 2, é apresentada a fundamentação teórica do trabalho. Iniciamos apresentando alguns conceitos sobre criptografia e infraestrutura de chaves públicas. Ainda neste capítulo, temos uma introdução ao ASI-HSM, módulo de segurança criptográfico que faz a proteção e gerência do ciclo de vida de chaves criptográficas. Esta é umas das possíveis áreas de aplicação para a biblioteca de mecanismos de segredo compartilhado proposta neste trabalho.

No capítulo 3, introduzimos o conceito de segredo compartilhado com uma introdução básica ao assunto. Abordamos tópicos tais como: histórico dos mecanismos de segredo compartilhado; estruturas de acesso; os diferentes modelos de mecanismos deste tipo; questões de segurança; assim como a questão de corpos finitos, conceito matemático importante para a definição desses mecanismos.

No capítulo 4, explicamos detalhadamente os dois mecanismos de segredo compartilhado que são estudados neste trabalho e que estão disponíveis na biblioteca implementada.

No capítulo 5, temos a descrição completa da implementação da biblioteca de mecanismos de segredo compartilhado proposta. Abordamos a modelagem estrutural, detalhes da implementação de cada um dos mecanismos de segredo compartilhado, as tecnologias utilizadas para o desenvolvimento deste trabalho, entre outros.

Por fim, no capítulo 6 são apresentadas as considerações finais obtidas através da realização deste trabalho e sugestões de trabalhos futuros.

2 Fundamentação Teórica

2.1 Criptografia

Criptografia (do grego kryptós, "escondido", e gráphein, "escrever") é geralmente entendida como sendo o estudo dos princípios e das técnicas pelas quais uma informação pode ser transformada da sua forma original para outra ilegível. Usualmente, essa transformação envolve o uso de uma "chave secreta", o que faz com que a informação original não possa ser recuperada por alguém não autorizado sem posse dessa chave [4].

Os objetivos principais da criptografia, quando utilizados para troca de mensagens, são [4]:

1. **confidencialidade** da mensagem: só o destinatário autorizado deve ser capaz de extrair o conteúdo da mensagem da sua forma cifrada;
2. **integridade** da mensagem: o destinatário deverá ser capaz de determinar se a mensagem foi alterada durante a transmissão;
3. **autenticação** do remetente: o destinatário deverá ser capaz de identificar o remetente e verificar que foi mesmo ele quem enviou a mensagem;
4. **não-repúdio** do remetente: não deverá ser possível ao remetente negar o envio da mensagem.

Nem todos os sistemas ou algoritmos criptográficos atingem todos os objetivos listados acima. Normalmente, existem algoritmos específicos para cada uma destas funções. Mesmo em sistemas criptográficos bem concebidos, bem implementados e usados adequadamente, alguns dos objetivos acima não são práticos (ou mesmo desejáveis) em algumas circunstâncias. Por exemplo, o remetente de uma mensagem pode querer permanecer anônimo, ou o sistema pode destinar-se a um ambiente com recursos computacionais limitados [4].

2.1.1 Criptografia Simétrica

Na criptografia simétrica a cifragem e decifragem de dados utilizam a mesma chave, que pode ser chamada de chave simétrica. O fato de ser usada apenas uma chave para as duas operações leva ao maior problema da criptografia simétrica, que é o compartilhamento dessa chave. Sempre que um arquivo cifrado for enviado a alguém, a pessoa que o recebe deve ter posse da chave simétrica para visualizar o arquivo real, o

problema está em como essa chave será compartilhada. Muitas vezes o meio de transmissão utilizado para o compartilhamento dessa chave não é seguro, comprometendo o sigilo. Qualquer meio de se compartilhar a chave simétrica é suscetível a uma falha que pode fazer com que o destinatário errado receba a chave, assim qualquer informação que foi cifrada com ela estará comprometida [5].

Na Figura 1, Alice deseja trocar uma mensagem particular com Bob.

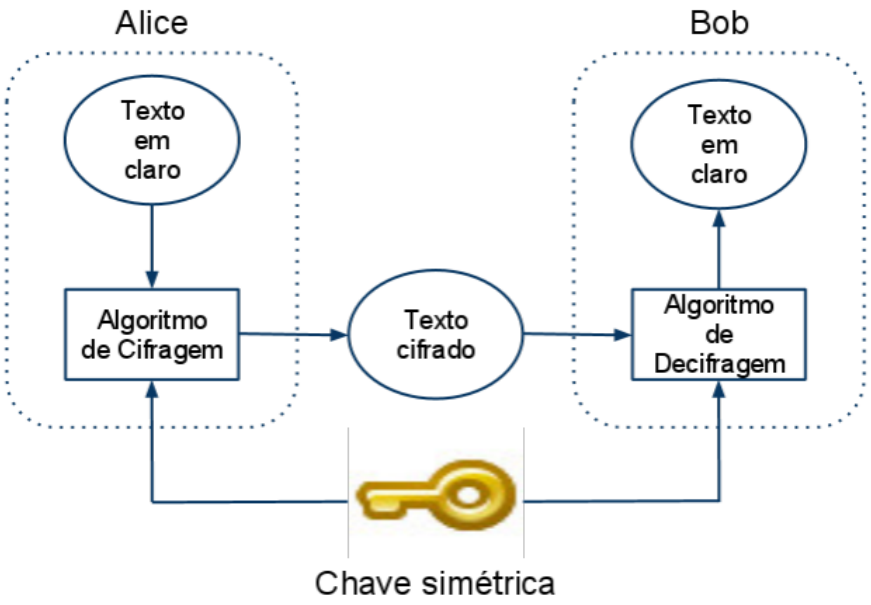

Figura 1 – Criptografia Simétrica

Alice aplica um algoritmo de cifragem na mensagem, fornecendo uma chave e envia para Bob a saída do algoritmo, que é a mensagem cifrada. Para Bob obter a mensagem original, ele aplica um algoritmo de decifragem na mensagem cifrada, fornecendo a mesma chave utilizada por Alice na cifragem e obtém a mensagem em claro.

A criptografia simétrica é uma abordagem simples, muito utilizada para cifrar grandes quantidades de dados, devido ao seu alto desempenho computacional. Apesar de sua simplicidade, existem alguns pontos críticos desta abordagem que devem ser considerados. A gerência das chaves pode se tornar muito custosa para uma rede com muitos usuários, pois cada entidade necessita de uma chave diferente para se comunicar de forma segura com cada uma das outras entidades. Por exemplo, em uma rede com x entidades, existiram $\frac{x(x-1)}{2}$ chaves.

Uma chave criptográfica não pode ser vulnerável a ataques de força bruta, onde um atacante tenta decifrar um documento usando todas as possíveis combinações de chave, geralmente se baseando em palavras de dicionário. Uma das características de uma boa chave é aquela que dificulta esse tipo de ataque, isso é definido pelo tamanho da chave, que torna o conjunto de chaves possíveis muito grande ou pela imprevisibilidade

(aleatoriedade) da chave gerada [5].

Dentre os algoritmos de criptografia simétrica atuais se destacam o DES [6] e o AES [7], ambos padronizados para o uso pelo governo norte-americano. Para evitar ataques tal qual a análise de frequência, os algoritmos simétricos são dispostos na forma de modos de operação [8].

Os modos de operação definem como os textos de saída ou de entrada dos algoritmos serão encadeados, de forma que de posse de pares de texto em claro e texto cifrado, um atacante não possa determinar a relação entre os textos.

2.1.2 Funções de Resumo Criptográfico

As funções resumo, conhecidas como funções de HASH, têm o objetivo de transformar uma sequência de bits em outra sequência de tamanho fixo. Essa função criptográfica não precisa de uma chave para ser realizada [5]. A Figura 2 exemplifica o funcionamento das funções de hash.

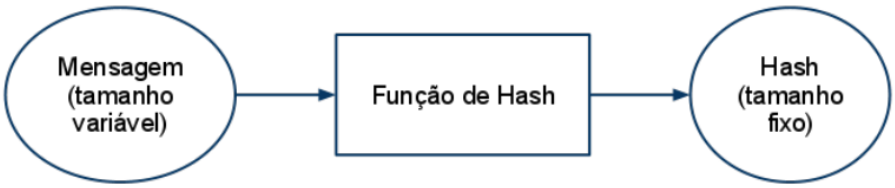

Figura 2 – Função de Hash

Como visto na figura anterior, a entrada para uma função resumo é uma sequência de bits de qualquer tamanho, o que torna o domínio da função um conjunto muito grande, maior que o contra-domínio que é uma sequência de bits de tamanho fixo. Isso faz com que na maioria dos casos o HASH calculado seja menor que a sequência de bits inicial.

Para um bom algoritmo de HASH, mesmo a alteração de um único bit em uma sequência de bits não deve resultar em um HASH igual ao calculado antes da alteração. Quando duas sequências diferentes de bits produzem um mesmo HASH, dizemos que ocorreu uma colisão de HASH. Existem meios de se prevenir colisões deste tipo como é o caso do “efeito avalanche” aplicado nos algoritmos. Esse efeito consegue dificultar a geração de um HASH igual a partir de duas sequências diferentes de bits [5].

Não é possível evitar total totalmente as colisões de HASH, visto que o conjunto de entradas é infinito e o conjunto de saídas é finito. Porém, um bom algoritmo de HASH deve evitar ao máximo essa colisão.

Resumidamente, uma função de resumo criptográfico, ou simplesmente, uma função de *hash* H precisa ter as seguintes propriedades [9]:

1. H tem que ser aplicável a um bloco de dados x de qualquer tamanho.

2. H precisa produzir uma saída de tamanho fixo.

3. $H(x)$ é relativamente fácil de ser computada para qualquer x, fazendo com que as implementações de hardware e software sejam práticas.

4. Para qualquer valor de resumo criptográfico h, é impossível descobrir o bloco de dados x que o gerou. Isso pode ser considerado como um função de propriedade *one-way*, ou seja, é impossível descobrir o valor que gerou o resultado tomando apenas o resultado como base de busca.

5. Para qualquer x, deverá ser computacionalmente inviável encontrar um bloco y $\neq$ x em que $H(y) = H(x)$.

O resumo criptográfico, por ser uma forma canônica de representar um bloco de dados de tamanho arbitrário, pode ser usado como uma forma de garantir a integridade de uma mensagem, pois é possível enviar a mensagem por completo a um destinatário e junto com ela enviar um resumo criptográfico dessa mensagem. Assim, antes der ler a mensagem recebida, o destinatário, conhecendo a função de resumo criptográfico usada, poderia aplicar essa função à mensagem e assim conferir se o resumo criptográfico recém gerado é o mesmo que o enviado. Agora, ele poderia ler a mensagem e ter certeza sobre sua integridade.

Atualmente, os algoritmos de resumo criptográfico mais utilizados são da família Secure Hash Algorithm (SHA) [10]: SHA-1, SHA-224, SHA-256, SHA-383, E SHA-512 [11].

2.1.3 Criptografia Assimétrica

Para resolver o problema do compartilhamento de chaves na criptografia simétrica, foram criados algoritmos de criptografia assimétrica. A criptografia assimétrica, também conhecida como criptografia de chave pública, foi inicialmente proposta por Diffie e Hellman [12] em 1976, e sua criação constituiu um importante marco na história da Criptografia que teve, a partir de então, seu enfoque mudado.

Neste sistema de criptografia existe um par de chaves formado por uma chave pública e uma chave privada, podendo a pública ser de conhecimento de todos. A chave privada deve ser conhecida e utilizada apenas pelo seu proprietário. Normalmente as chaves públicas são mantidas em diretórios e bases de dados públicas para que qualquer pessoa tenha acesso a esta informação, facilitando assim a cifragem e decifragem na troca de mensagens [13].

Existe uma relação matemática entre as chaves pública e privada, porém qualquer pessoa que obtenha uma chave pública não é capaz de deduzir a chave privada correspondente. Assim é possível dizer que um atacante que detenha a chave pública de qualquer pessoa não será capaz de obter a chave privada da mesma pessoa através de funções matemáticas [13].

Neste paradigma de criptografia, o mecanismo de cifragem de decifragem de dados foi concebido de forma que as chaves pública e privada operem sempre de forma complementar, ou seja, tudo o que é cifrado com a chave pública só pode ser decifrado com a respectiva chave privada.

É possível utilizar tanto uma chave privada quanto uma chave pública para criptografar uma informação, o importante é saber que para se decifrar a mensagem sempre se deve utilizar a chave oposta a qual foi utilizada no processo de cifragem. Na Figura 3, temos um exemplo: se Alice cifra uma mensagem com a chave pública de Bob, Bob só conseguirá decifrá-la utilizando sua chave privada.

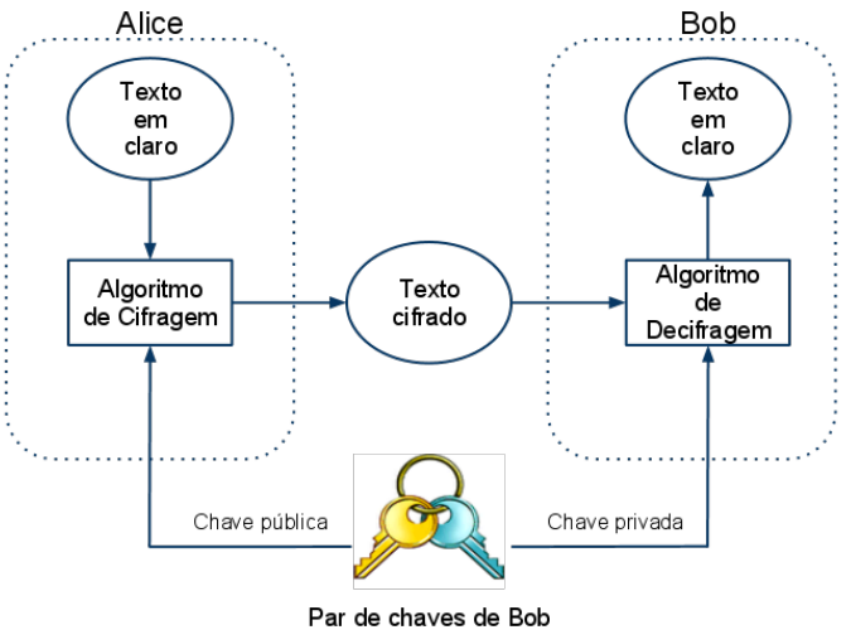

Figura 3 – Criptografia Assimétrica

A escolha da chave para cifragem irá depender do requisito de segurança que o emissor está empregando. Se o serviço de confidencialidade é mais importante, a chave que deve ser utilizada para cifragem é a pública. Assim, só o destinatário possui a chave privada correspondente, e somente esse conseguirá decifrar a mensagem. Já se o serviço de segurança prioritário é a autenticidade, o emissor deve cifrar a mensagem com sua chave privada. Com isso, qualquer pessoa que utilizar sua chave pública conseguirá decodificar a mensagem, provando assim que o emissor é autêntico. Isso acontece pelo simples motivo de

que só o detentor daquela chave privada seria capaz de cifrar uma mensagem decodificada pela respectiva chave pública.

Alguns exemplos de algoritmos de chave assimétrica [13]:

- RSA (Rivest-Shamir-Adleman)
- Elliptic curve cryptosystem (ECC)
- Diffie-Hellman
- El Gamal
- Knapsack

Dentre os acima citados, o sistema de criptografia assimétrica mais amplamente conhecido na atualidade é o RSA [14], sistema desenvolvido por Rivest, Shamir e Adleman. Baseado nas idéias de Diffie e Hellman, a segurança do RSA é baseada na solução de um difícil problema matemático, baseado na teoria de números, que é a fatoração de produtos de números primos em módulo n.

2.1.4 Certificados Digitais e Infraestrutura de Chaves Públicas

Uma Infraestrutura de chaves públicas (ICP) é formada por programas, formatos de dados, procedimentos, protocolos de comunicação, políticas de segurança, e mecanismos de criptografia de chave pública que trabalham em conjunto para possibilitar que pessoas se comuniquem de forma segura. Em outras palavras, uma ICP é responsável por estabelecer o nível de confiança em um ambiente [13].

Uma ICP provê suporte à serviços de autenticidade, confidencialidade, não repúdio, e integridade, cujo conceitos vimos anteriormente. Existe uma diferença entre criptografia de chave pública e infraestrutura de chave pública. A primeira, como já foi explicado anteriormente, é um outro nome para algoritmos de criptografia assimétrica, enquanto a segunda é o que o próprio nome já diz, uma infraestrutura. Esta infraestrutura assume que a identidade do receptor pode ser assegurada através de certificados digitais e algoritmos assimétricos. Portanto, a ICP contém as peças necessárias para identificar usuários, criar e distribuir certificados, manter e revogar certificados, distribuir e manter as chaves de criptografia, e todas as tecnologias necessárias para se alcançar o objetivo da comunicação criptografada e autêntica [13].

Qualquer pessoa que deseja participar de uma ICP deve requisitar um certificado digital, que é uma credencial que contém a chave pública daquele indivíduo, juntamente com outras informações de identificação. O certificado é criado e assinado por uma terceira parte confiável, conhecida como Autoridade Certificadora (AC). Quando a AC assina um

certificado, vincula-se a identidade do proprietário a uma chave pública, e a AC assume a responsabilidade pela autenticidade do indivíduo. É essa terceira parte confiável (AC) que permite que pessoas que nunca se encontraram possam se identificar umas com as outras para obter uma comunicação segura. Para isso basta que as partes envolvidas na comunicação confiem na mesma parte confiável (AC).

A Autoridade Certificadora é o elemento chave na construção de uma Infraestrutura de Chaves Públicas. Ela é uma composição de mecanismos de software e de hardware e tem como função principal a emissão de certificados digitais.

Pode existir uma hierarquia de ACs, formada por uma AC-Raiz e demais ACs subordinadas. As ACs intermediárias ou subordinadas serão confiáveis se no caminho de certificação existir uma AC confiável e, também, existirem repositórios de certificados de todos os nodos do caminho, até chegar na AC confiável.

A rede de confiança de uma ICP é formada pelas Autoridades Certificadoras (AC) que a compõe, sendo que cada AC deve manter a sua chave privada em segurança, para que a confiabilidade dos certificados gerados por ela seja garantida. A guarda da chave privada se tornou uma função crucial para manter a ICP íntegra. Na seção 2.2, falamos sobre um Módulo de Segurança Criptográfico (HSM, da sigla em inglês), que é um equipamento produzido especialmente para este fim.

2.2 O ASI-HSM

Com a crescente utilização do meio digital para o armazenamento e trânsito de informações, a necessidade de proteção de dados e transações têm se tornado uma preocupação constante. Com a adoção de algoritmos públicos, padronizados e de eficácia comprovada para a criptografia de dados, as chaves critográficas são a principal preocupação e devem ser mantidas sob rígido controle. Surgiu daí a necessidade de um dispositivo dedicado à proteção e gerência de tais artefatos.

Em infraestruturas de chaves públicas, as chaves privadas são de vital importância. O custo de perda, roubo ou indisponibilidade de uma chave privada para uma autoridade certificadora cresce de maneira inversamente proporcional ao nível da AC em questão, ou seja, em ACs de último nível, a implicação seria a revogação de todos os certificados emitidos aos usuários finais. Já em uma AC de nível 0 (Raiz), as consequências seriam catastróficas, visto que toda a infraestrutura estaria comprometida.

Um Módulo de Segurança Criptográfico (do inglês, *Hardware Security Module*, HSM) abrange, no mínimo, um conjunto composto por um *hardware* (o módulo em si) e *software* (para sua gerência e configuração). A principal finalidade de um HSM é prover proteção e efetuar a gerência do ciclo de vida de chaves criptográficas, como a chave

privada de uma autoridade certificadora, por exemplo. Para cumprir esta finalidade, um HSM conta com a implementação de inúmeros algoritmos criptográficos, bem como de proteções físicas contra violação e acesso indevido ao material protegido.

Esquemas de segredo compartilhado são muito úteis no gerenciamento de chaves criptográficas, sendo fundamentais na operação de um HSM, como explicado melhor nas seções seguintes.

Este trabalho foi desenvolvido no contexto do LabSEC, como é chamado o Laboratório de Segurança em Computação da Universidade Federal de Santa Catarina (UFSC). O laboratório, que tem por objetivo estudar, pesquisar, avaliar e implementar soluções na área de segurança em computação, produziu o o ASI-HSM (Figura 4) em parceria com a Rede Nacional de Ensino e Pesquisa (RNP)[1] e com a empresa brasileira Kryptus[2], como como parte da iniciativa GT ICPEdu II.

Figura 4 – Módulo de Segurança Criptográfico ASI-HSM

Fonte: http://www.kryptus.com/#!asi-hsm/c11e6

O ASI-HSM é um módulo de segurança criptográfico com tecnologia brasileira, de código aberto e de menor custo. O ASI-HSM possui um perímetro criptográfico composto por uma Unidade de Segurança e uma Unidade de Gerência. A Unidade de Segurança conta com sensores de tensão, temperatura, luminosidade e detecção de intrusão física, cujo objetivo é detectar qualquer tentativa de violação. Já a Unidade de Gerência hospeda o *firmware* responsável pela gerência do ciclo de vida das chaves criptográficas, as ferramentas e bibliotecas necessárias, assim como as próprias chaves [15].

2.2.1 Ciclo de vida de Chaves Criptográficas

Para minimizar todos os riscos anteriormente citados para as chaves criptográficas, o HSM deve gerenciar uma série de aspectos relacionados com o ciclo de vida de chaves criptográficas, desde a criação de uma chave, o seu uso e a sua correta destruição. São eles:

[1] http://www.rnp.br/
[2] http://kryptus.com.br

- **Geração**: O HSM deve garantir uma geração confiável, baseada em valores o mais aleatórios possíveis, visando a total imprevisibilidade de suas chaves. O HSM deve garantir que as chaves geradas terão as qualidades necessárias para não comprometer os algoritmos nas quais estarão sendo usadas. Isto normalmente compreende, além de gerar uma chave, testar se a mesma não é considerada fraca para o algoritmo criptográfico no qual estará sendo utilizada.

- **Armazenamento**: É papel do HSM controlar o armazenamento e o acesso aos parâmetros críticos de segurança e chaves criptográficas contidas em seu interior. Para tanto, deve impossibilitar o vazamento de tais informações para o exterior do perímetro criptográfico, além de implementar um controle de acesso forte, garantindo disponibilidade para papéis autorizados.

- **O uso de chaves**: Deve ser garantido o correto uso das chaves criptográficas, garantindo que elas estarão sempre disponíveis aos seus detentores, mesmo no caso de desastres, e além disso, deve ser garantido que o acesso será controlado e restrito a somente pessoas autorizadas.

- **A auditoria do uso de chaves**: O HSM deve manter um registro seguro das operações por ele executadas. Dessa maneira é possível, em caso de mal uso, a recuperação dos registros e avaliação dos prejuízos trazidos por ele. Deve ser garantido a criação de um histórico completo e rastreável de tudo o que foi feito com uma chave, desde o momento de sua criação até o momento de sua destruição.

- **Backup/Recuperação**: Um HSM deve permitir a realização de procedimento de backup de seu material sensível, para que em caso de falha, o ambiente possa ser restaurado sem maiores contratempos.

- **Destruição de chaves gerenciadas**: Quando uma chave criptográfica atinge seu objetivo, ela deve ser destruída, ou seja, apagada da memória interna do HSM afim de que não possam ser utilizadas fora do ambiente de controle do seu ciclo de vida. Apesar de ser uma tarefa aparentemente trivial, este ainda é um problema em aberto, no que diz respeito à gerência de chaves. Isto porque não é fácil garantir a destruição de todas as cópias e arquivos de backup que contêm uma determinada chave, podendo levar a uma falsa sensação de segurança por parte do usuário.

2.2.2 Grupos de Gerenciamento

Para a execução das funções do HSM é necessária a autenticação dos indivíduos responsáveis por aquela função, ou seja, aqueles que tem permissão para executá-la.

O ASI-HSM suporta a criação de grupos que podem ter de 1 a M membros, sendo o $M \geq 1$. Para os casos onde há a necessidade da autorização de várias pessoas para

a execução da tarefa, o M será maior do que 1. A autenticação de um grupo é melhor explicada na subseção 2.2.3.

No ASI-HSM existem três tipos de grupos de gerência, e os grupos de gerência fazem parte de um perfil de gerência. Não é possível afirmar que um perfil no ASI-HSM seja mais importante que o outro, pois suas funções se complementam para a operação e funcionamento do mesmo.

2.2.2.1 Administração

O perfil de administração no ASI-HSM só possui um grupo, o qual é responsável por todas as tarefas administrativas relativas ao HSM, excetuando-se as atividades de auditoria. Não é possível haver mais de um grupo ativo administrando o HSM simultaneamente, mas existe a possibilidade de se alterar o grupo existente, excluindo o grupo antigo e criando um novo grupo. Este é o perfil responsável por criar todos os outros grupos do HSM. Além disso, os administradores serão responsáveis por processos administrativos internos ao HSM, os quais consistem em:

- Inicialização e Operacionalização do HSM.
- Configuração do provedor e determinação de todos os seus parâmetros de operação, entre os quais estão os tamanhos de chaves assimétrica, endereços de acesso, etc.
- Criação de Operadores, os quais serão os detentores e utilizadores das chaves assimétricas.
- Gerência das Chaves, tais como a sua delegação a um conjunto de operadores, a sua destruição ou inutilização.
- Criação de Chaves assimétricas para uso em aplicações externas ao provedor.
- Participação de forma ativa nos procedimentos de cópias de segurança.
- Criação do Conjunto de Auditores que vão gerenciar os processos de auditoria de registros e a sua legitimidade.

O grupo de administradores é o primeiro grupo a ser criado no ASI-HSM, e pode ser trocado, mas nunca excluído. Com os administradores efetivamente criados e aptos a configurar o HSM, temos a necessidade da execução de um novo procedimento para dar andamento à operacionalização do provedor, que é a criação do conjunto de operadores.

2.2.2.2 Auditoria

O perfil de auditoria no ASI-HSM tem a responsabilidade de verificar o bom andamento das operações do módulo. Ao perfil de auditoria cabe a monitoração da utilização do módulo, sendo este responsável pela extração e análise dos logs gerenciais e técnicos, contendo, respectivamente, registro de operações efetuadas pelo módulo e logs dos sensores, no caso de alguma invasão ter sido detectada.

É possível que um HSM possua mais de um grupo de auditores e é necessário que exista pelo menos um grupo. Suas principais funções são:

- Exportar logs da unidade de gerência.
- Exportar logs da unidade de segurança.
- Bloqueio do equipamento.
- Recuperar backup, juntamente com administradores.
- Apagar logs, juntamente com administradores.

Após a criação de um grupo de auditores, ele não pode ser excluído.

2.2.2.3 Operação

O perfil de operação no ASI-HSM é o responsável por gerenciar o uso de chaves criptográficas. A criação de chaves criptográficas é função do perfil de administração, mas a decisão de quando utilizar as chaves e quantas vezes elas serão utilizadas fica a cargo do grupo de operadores responsável pela chave. Suas principais funções no HSM são:

- Carregar chave para uso.
- Definir políticas de utilização da chave:
  - Tempo de uso
  - Número de usos

Após a a criação de um grupo de operadores, ele não pode ser excluído.

2.2.2.4 Funções Comuns a Todos os Perfis

Existem várias operações no ASI-HSM que não necessitam de autenticação por se tratarem de operações onde não existe o tráfego de informações sigilosas. São operações como verificar a versão do software do ASI-HSM, mostrar seu estado atual, realizar auto-testes, listar grupos e membros de grupos de gerência, entre outras.

As duas funções mais importantes que podem ser executadas sem autenticação são o descarregamento de uma chave carregada e a configuração do ASI-HSM. O carregamento de uma chave privada para uso só pode ser realizado pelo grupo de operadores responsável por aquela chave, mas o descarregamento pode ser executado sem autenticação. A configuração do ASI-HSM é uma tarefa de inicialização do equipamento para uso e ela ocorre antes mesmo da criação do grupo de administradores, ou seja, não há meios de se executar uma autenticação visto que não existe qualquer grupo criado nesse momento.

2.2.3 Autenticação

No ASI-HSM existem três tipos de grupos que necessitam de autenticação como foi descrito na seção subseção 2.2.2. A autenticação de um grupo de gerência deve ser realizada toda vez que uma função que necessita de autenticação é invocada.

O tamanho do grupo de gerência é definido no momento de sua criação. O grupo deve ter um M, que é o número máximo de participantes, e um N, que é o número de participantes necessários para a autenticação, tal que $1 \leq N \leq M$.

A autenticação de membros de grupos do ASI-HSM é composta por dois fatores de autenticação, que são a prova de posse e a prova de conhecimento. Para a prova de posse é utilizado um *smartcard* para cada membro de um grupo, e a prova de conhecimento é o PIN do membro.

Todo usuário faz parte de um grupo que detém a posse das partes de um segredo compartilhado. Essas partes do segredo ficam armazenadas no HSM, cifradas com a chave pública do usuário. No momento da autenticação de um perfil, a parte cifrada do segredo compartilhado é entregue para o smartcard, que é o único capaz de decifrá-la por conter a chave privada do usuário. Para fazer uso da chave privada do usuário, é necessário decifrá-la com o PIN.

Na operação do HSM, uma operação que necessita de autenticação pode ser realizada somente após a autenticação de N membros do grupo designado para realizar tal operação. Esta forma de autenticação só é possível devido a utilização de mecanismos de segredo compartilhado, que são explicados adiante e constituem o assunto principal deste trabalho.

3 Segredo Compartilhado: uma Introdução

3.1 Breve histórico

No ano de 1979, George Blakley [16] e Adi Shamir [17] propuseram independentemente esquemas de segredo compartilhado através dos quais um segredo pode ser dividido em várias partes as quais podem ser distribuídas a participantes membros de um grupo. O esquema proposto por Shamir baseia-se na interpolação polinomial de Lagrange enquanto o esquema proposto por Blakley é fundamentado na geometria projetiva linear [18]. Nestes primeiros esquemas de segredo compartilhado criados, apenas o número de participantes era importante na fase de recontrução do segredo. Tais esquemas foram chamados *threshold secret sharing schemes* [19]. Em uma tradução livre, podemos chamá-los de "esquemas de segredo compartilhado baseados em um limiar".

O objetivo final de um esquema como o citado acima é dividir um segredo em n *shares*, mas de tal forma que qualquer subconjunto de k *shares*, onde k é o chamado *threshold* ou limiar do esquema, seja suficiente para determinar o segredo. Adicionalmente, qualquer subconjunto de k-1 ou menos *shares* não será suficiente para a reconstrução do mesmo. Este esquema é definido como um *(k,n)-threshold scheme*, significando que das n partes em que o segredo foi dividido, ao menos k delas são necessárias para que o segredo possa ser determinado.

Posteriormente ao esquema inicial proposto por Blakley e Shamir, diversas outras teorias e esquemas de segredo compartilhado distintos foram desenvolvidos. Nesses esquemas, peculiaridades à parte, não determinamos um limiar, mas especificamos quais conjuntos de participantes podem relevar o segredo. Um conjunto de participantes apto a reconstruir o segredo fará parte da *estrutura de acesso* do esquema, conceito este melhor descrito na seção 3.2.2. Podemos citar, por exemplo, o *weighted threshold secret sharing scheme* [19] nos qual um peso positivo é associado a cada um dos usuários participantes do esquema e o segredo pode ser recontruído se, e somente se, a soma dos pesos dos participantes tentando a reconstrução do segredo for maior ou igual a um limiar fixado.

Um outro exemplo são os *hierarchical secret sharing schemes* [19], também chamados de esquemas de segredo compartilhado multiníveis. Nesse tipo de esquema o conjunto de usuários participantes é particionado em alguns níveis e o segredo pode ser recuperado se, e somente se, quando da reconstrução do segredo existir um nível de inicialização tal que o número de participantes deste nível ou níveis superiores é maior ou igual ao limiar do nível de inicialização estabelecido. Podemos citar ainda os *compartmented secret sharing schemes* [19] nos quais o conjunto de usuários é particionado

em compartimentos e o segredo pode ser recuperado se, e somente se, o número de participantes de qualquer compartimento é maior ou igual a um compartimento limiar e o número total de participantes é maior ou igual do que um limiar global estabelecido.

Para o escopo deste trabalho, nossos objetos de estudo serão os *threshold secret sharing schemes* propostos por Shamir e Blakley citados no início da seção.

3.2 Conceitos Preliminares

O funcionamento de um esquema de segredo compartilhado inicia-se sempre a partir de um segredo o qual é dividido em *shares* que serão distribuídas aos usuários de forma secreta, de modo que nenhum participante conheça a share dada a outro participante. O segredo poderá ser reconstruído apenas por grupos predeterminados.

Usualmente, um esquema de segredo compartilhado é coordernado por um participante especial chamado *dealer*, responsável, entre outras coisas, pela geração e distribuição das *shares*. No entanto, também existem esquemas de segredo compartilhado que podem ser configurados sem sua presença.

Em algumas situações, o valor do segredo é predeterminado. Nesse caso o segredo é chamado explícito. Nos casos de segredo explícito, o *dealer* recebe este valor, a partir dele deriva as *shares* e em seguida distribui as mesmas de forma segura aos usuários. No caso de o segredo ser implícito, não há qualquer restrição quanto ao seu valor desde que pertença a algum domínio. Nesse caso, o *dealer* primeiramente gera o segredo em algum domínio predeterminado, em seguida deriva as *shares* correspondentes e, finalmente, distribui as mesmas de forma segura aos usuários.

A reconstrução do segredo pode ser feita pelos próprios participantes após reunirem suas *shares*. Alternativamente, eles poderiam dar suas *shares* para uma autoridade confiável, chamada combinador, que realizaria a tentativa de reconstrução do segredo para eles.

Em esquemas de segredo compartilhado, as operações aritméticas são realizadas em um *corpo finito*. Damos uma breve introdução sobre este conceito matemático na subseção 3.2.1. Nas próximas seções temos o detalhamento de outros conceitos importantes relacionados a segredo compartilhado.

3.2.1 Corpos Finitos

Um *corpo* pode ser definido, de forma simplificada, como um conjunto no qual podemos somar, subtrair, multiplicar e dividir por não nulo, no qual valem todas as propriedades usuais de tais operações, incluindo a comutativa da adição e da multiplicação.

Em matemática e, em especial, na teoria dos corpos, um *corpo finito* é um corpo em que o conjunto dos elementos é finito.

A aritmética sobre corpos finitos é diferente da aritmética padrão sobre o conjunto dos inteiros. Sobre um corpo finito, todas as operações aritméticas realizadas resultam em um valor dentro do mesmo corpo finito [20].

A teoria dos corpos finitos desenvolveu-se extensivamente no século XIX, porém a sua origem data dos séculos XVII e XVIII. Os primeiros pesquisadores a considerar corpos finitos foram Pierre de Fermat, Leonhard Euler, Joseph-Louis Lagrange, Adrien-Marie Legendre e Carl Gauss. Na época, os únicos corpos finitos conhecidos eram os corpos com um número primo de elementos, denotados por F_p (onde p é um número primo). Os elementos de um corpo desse tipo podem ser vistos como os inteiros módulo p. Posteriormente, descobriu-se a existência de outros tipos de corpos finitos.

A aparição do artigo *"Sur la théorie des nombres"* de Évariste Galois, em 1830, foi fundamental para o surgimento de várias questões quanto à estrutura de corpos finitos em geral. Em 1857, Richard Dedekind caracterizou corpos finitos com p^n elementos, onde p é primo. Anos mais tarde, em 1893, Eliakim Moore mostrou que qualquer corpo finito contém p^n elementos.

O livro de Leonard Dickson, publicado em 1901, já tinha os resultados mais importantes sobre a estrutura de corpos finitos.

A seguir, apresentamos uma pequena lista com os principais resultados sobre corpos finitos:

1. O número de elementos num corpo finito (que é também chamado de cardinalidade) é uma potência da característica prima do corpo.

2. Se p é primo e n é um inteiro positivo, então existe um único corpo finito com p^n elementos, a menos de isomorfismos. O primo p é chamada a característica do corpo, e o número inteiro positivo n é chamado a dimensão do corpo.

3. O grupo multiplicativo dos elementos não nulos de um corpo finito é cíclico.

4. Seja F um corpo com p^n elementos. O número de elementos num subcorpo de F é da forma p^d, onde d é um divisor de n. Reciprocamente, se d divide n, então existe um subcorpo de F com p^d elementos.

5. Todo elemento a num corpo finito com q elementos satisfaz $a^q = \text{a}$.

No século XX, o uso de corpos finitos foi extramamente difundido, em parte devido à aparição dos computadores. Algumas áreas de aplicação de corpos finitos

são: criptografia, teoria de códigos, processamento digital de sinais, sequências pseudo-aleatórias, transformada discreta de Fourier, entre outras. Corpos finitos também aparecem relacionados às áreas da matemática como combinatória, geometria algébrica, geometria aritmética, geometria finita e teoria de números [21].

No contexto deste trabalho, corpos finitos são extremamente importantes pois, como será visto adiante, os esquemas de segredo compartilhado estudados e implementados baseiam-se nesse conceito matemático.

3.2.2 Estruturas de Acesso

Estruturas de acesso são usadas no estudo de sistemas de segurança onde várias partes têm de trabalhar juntas para conseguir obter um recurso. Este recurso pode ser, por exemplo, uma tarefa que um grupo de participantes só é capaz de completar em conjunto, tais como a criação de uma assinatura digital, ou decifrar uma mensagem criptografada.

O uso original de estruturas de acesso se dá na área da criptografia, onde o recurso é um segredo compartilhado entre os participantes. Apenas os subgrupos de participantes contidos na estrutura de acesso estão aptos para unirem suas *shares* a fim de recomputar o segredo. Sendo assim, a estrutura de acesso de um esquema de segredo compartilhado é o conjunto de todos os grupos que são designados para reconstruir o segredo. Os elementos da estrutura de acesso são comumente chamados de *grupos autorizados* e os demais não pertencentes a estrutura são chamados *grupos não autorizados* [19].

Intuitivamente, se um dado grupo G pertence a uma dada estrutura de acesso, todos os outros conjuntos que tenham G como subconjunto deverão também fazer parte da estrutura de acesso. Benaloh e Leichter chamaram tais estruturas de acesso de *monótonas* [22].

Existem ainda esquemas de segredo compartilhado que lidam com estruturas de acesso *não-monótonas* [19]. É o caso, por exemplo, dos esquemas de segredo compartilhado com capacidade de veto. Nesses esquemas, existem *shares* positivas e *shares* negativas, que correspondem a *shares* verdadeiras e falsas, sendo que as últimas permitem o veto, ou seja, que a reconstrução do segredo possa ser efetuada de forma correta. A possibilidade de veto é garantida apenas se assumirmos que a entidade responsável pela reconstrução do segredo é completamente confiável.

No momento da reconstrução do segredo em um esquema com possibilidade de veto, cada participante do esquema fornecerá sua *share* positiva ou negativa para a entidade responsável por tal função. Se a intenção do participante for permitir a reconstrução do segredo, o participante irá fornecer sua *share* positiva, que é uma *share* verdadeira. Se a intenção do participante é vetar a reconstrução do segredo, o participante irá fornecer uma *share* falsa, o que impedirá a reconstrução do segredo original.

Como comentado na seção 3.1, há diversos esquemas de segredo compartilhado possíveis, utilizando diferentes configurações de estruturas de acesso. Nos primeiros esquemas de segredo compartilhado, que foram independentemente propostos por Shamir e Blakley, o conceito de estruturas de acesso não é utilizado da forma convencional. Isso porque nesse tipo de esquema, chamado *(k,n)-threshold scheme*, não é possível que se determine quais grupos de participantes estarão aptos a reconstruir o segredo. Os grupos autorizados a reconstruir o segredo são todos aqueles que possuírem pelo menos k participantes, onde k é o limiar especificado para o esquema.

No entanto, existem esquemas de segredo compartilhado que lidam com estruturas de acesso mais complexas do que as citadas anteriormente. Podemos citar o caso dos, já citados, esquemas de segredo compartilhado hierárquicos, do inglês *hierarchical secret sharing schemes* [19]. Nesse tipo de esquema, o segredo é compartilhado entre um grupo de participantes que é particionado em níveis L_1, L_2, ..., L_m, de forma hierárquica, sendo L_1 o nível mais alto da hierarquia e L_m o nível mais baixo. A cada um dos níveis da hierarquia é atribuído um valor chamado *level-threshold*, que podemos denotar por k_i, de tal forma que $k_1 < k_2 < ... < k_m$. Este valor corresponde a quantidade de participantes deste nível que será necessário para reconstruir o segredo.

Em um esquema de segredo compartilhado hierárquico, um subconjunto de participantes é autorizado (a reconstruir o segredo) se possuir pelo menos k_1 membros do grupo mais elevado da hierarquia, bem como pelo menos $k_2 > k_1$ membros dos dois níveis mais altos da hierarquia e assim por diante. Esse tipo de esquema pode ser necessário em configurações onde os participantes possuem diferentes níveis de autoridade ou confiança e a presença de participantes de diferentes níveis dentro de uma hierarquia é determinante para a reconstrução do segredo.

Neste trabalho, o foco está sobre esquemas de segredo compartilhado baseados em *threshold*, que em suas formas originais não abrangem possibilidades como a existência de *shares* de veto ou ainda a construção de hierarquias.

3.2.3 Modelos de Segredo Compartilhado

Intuitivamente, podemos dizer que um esquema de segredo compartilhado é um método de dividir um segredo em partes de tal forma que o mesmo possa ser reconstruído apenas por grupos autorizados. Dependendo da "quantidade" de informação secreta a que um grupo não autorizado consegue ter acesso, os esquemas de segredo compartilhado podem ser classificados em duas categorias distintas:

- Esquemas de Segredo Compartilhado Perfeitos (do inglês, *Perfect Secret Sharing Schemes*): nesse tipo de esquema as *shares* de um grupo não autorizado não fornecem qualquer informação a respeito do segredo [19].

- Esquemas de Segredo Compartilhado Computacionalmente Seguros (do inglês, *Computational-Secure secret sharing schemes*): nesse tipo de esquema grupos não autorizados conseguem ter acesso a alguma informação sobre o segredo, mas o problema de encontrar qual o segredo é intratável.[1] Tais esquemas podem levar a shares de menor tamanho [19].

3.2.4 Tamanho das *Shares*

Um aspecto de grande importância na implementação de esquemas de segredo compartilhado é o tamanho das *shares*, uma vez que a eficiência e a segurança de um sistema descresce a medida que a quantidade de informação que precisa ser mantida em segredo aumenta. O problema de dar limites ao tamanho da *share* dada a um participante tem recebido considerável atenção nos últimos anos [23].

Infelizmente, em um *Perfect Secret Sharing Scheme* o tamanho das *shares* não pode ser menor que o tamanho do segredo. É esta propriedade que irá garantir que grupos não autorizados de participantes não terão absolutamente nenhuma informação sobre o segredo. No caso de um *Computational-Secure secret sharing scheme*, o tamanho das *shares* pode ser menor que o tamanho do segredo, justamente pelo fato de esse tipo de esquema não garantir que grupos não autorizados sejam impossibilitados de obter informações sobre o segredo [1].

3.2.5 Segurança de um esquema de segredo compartilhado

Seja um *(k,n)-threshold scheme*, a segurança geral do mesmo é medida pela quantidade de informação dada sobre o segredo por cada uma das *shares*, bem como pela quantidade de informação sobre o segredo que se pode determinar com k-1 *shares*. O ideal para um esquema de segredo compartilhado é que k-1 *shares* não forneçam absolutamente qualquer informação sobre o segredo. Como já foi dito anteriormente, quando um esquema de segredo compartilhado atende esse critério é conhecido como um esquema de segredo compartilhado perfeito. Em um esquema desse tipo, k-1 *shares* iriam fornecer a um possível grupo de usuários maliciosos a mesma probabilidade de encontrar o segredo do que a probabilidade de um grupo que não possua nenhuma *share* fazer o mesmo.

Uma outra definição importante sobre esquemas de segredo compartilhado é a chamada *information rate*. Trata-se de uma medida definida como a razão entre o tamanho do segredo em bits e o tamanho em bits de cada uma das *shares*, ou mais formalmente, para uma *share* com tamanho S_i e um segredo de tamanho K:

$$InformationRate = \frac{log_2 K}{log_2 S_i}$$

[1] Um problema é chamado *intratável* se não existe um algoritmo polinomial capaz de resolvê-lo, ou seja, não há um algoritmo que resolva-o em uma quantidade razoável de tempo.

Em um esquema no qual o tamanho das *shares* seja igual o tamanho do segredo, a *information rate* é igual a 1. Nesse caso, se o esquema de segredo compartilhado for perfeito, ele pode ser classificado como *ideal* [19]. À medida que o tamanho das *shares* aumenta, a *information rate* diminui. Um maior tamanho das *shares* também faz com que uma maior quantidade de informação necessite ser transferida aos participantes no momento de distribuí-las na fase de divisão do segredo, bem como no momento de combiná-las para efetuar a reconstrução do segredo.

Adicionalmente, para uma *information rate* maior do que 1 alguns detalhes sobre o segredo devem ser liberados, o que significa que alguma informação sobre o valor do segredo será dada em cada uma das *shares*. Isso significa que se a *information rate* é maior do que 1, o esquema garantidamente não é um esquema de segredo compartilhado perfeito [3].

4 Esquemas de segredo compartilhado com Limiar

Como já citado anteriormente, os *threshold secret sharing schemes* foram os primeiros esquemas de compartilhamento de segredos propostos. No ano de 1979, Adi Shamir [17] e George Blakley [16] propuseram independentemente seus métodos. São definidos de forma que dentre n participantes, qualquer *threshold* (ou limiar) de $k \leq n$ possa obter o segredo (estes são chamados de esquemas (k, n) de compartilhamento). Esses esquemas são, na realidade, um caso especial de mecanismos de segredo compartilhado, pois não utilizam o conceito de estruturas de acesso da forma convencional.

Nesse tipo de esquema, não é possível a escolha de quais participantes estarão aptos a revelar o segredo em conjunto. Os grupos autorizados a reconstruir o segredo são todos aqueles que possuírem pelo menos k participantes, onde k é o limiar especificado para o esquema [24].

4.1 Esquema de Shamir

Duas observações muito simples formam a base do esquema de segredo compartilhado de Shamir:

- *Interpolação*: dados $n + 1$ pontos (x_i, y_i) diferentes, há um único polinômio $q(x)$ de grau n passando por todos os pontos. Este pode ser determinado usando o polinômio interpolador de Lagrange:

$$q(x) = \sum_{j=0}^{n} y_j q_j(x)$$

$$q_j(x) = \prod_{k=0;k\neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

- *Sigilo*: dado um conjunto contendo menos que $n + 1$ pontos, pode-se determinar infinitos polinômios passando por todos os pontos do conjunto [24].

4.1.1 Ideia Geral

O esquema de segredo compartilhado proposto por Shamir define uma forma para que um dado D (um segredo) possa ser dividido em n partes de tal forma que o segredo D pode ser facilmente recontruído a partir de k peças do segredo, mas mesmo o conhecimento

de k-1 partes do segredo não revela qualquer informação sobre o mesmo. Como já citado anteriormente, esta é a definição de um *(k,n)-threshold scheme*. Se $k = n$, então todas as partes do segredo são necessárias para a reconstrução do mesmo. Esta técnica permite a construção de esquemas robustos de gerenciamento de chaves para sistemas criptográficos que podem funcionar de forma segura e confiável mesmo quando k-1 partes do segredo forem expostas por falhas de segurança, por exemplo [17].

Como citado anteriormente, o esquema proposto por Shamir é baseado na interpolação polinomial de Lagrange (Figura 5): em um plano bidimensional, dados k pares ordenados $(x_1, y_1), \ldots, (x_k, x_k)$ com $x_i \neq x_j$ para todo $1 \leq i < j \leq k$, é garantido a existência de um único polinômio *q(x)* de grau k-1 de tal forma que $q(x_i) = y_i$ para todo 1 $\leq i \leq k$. A prova da unicidade deste polinômio obtida através dos polinômios de Lagrange [3].

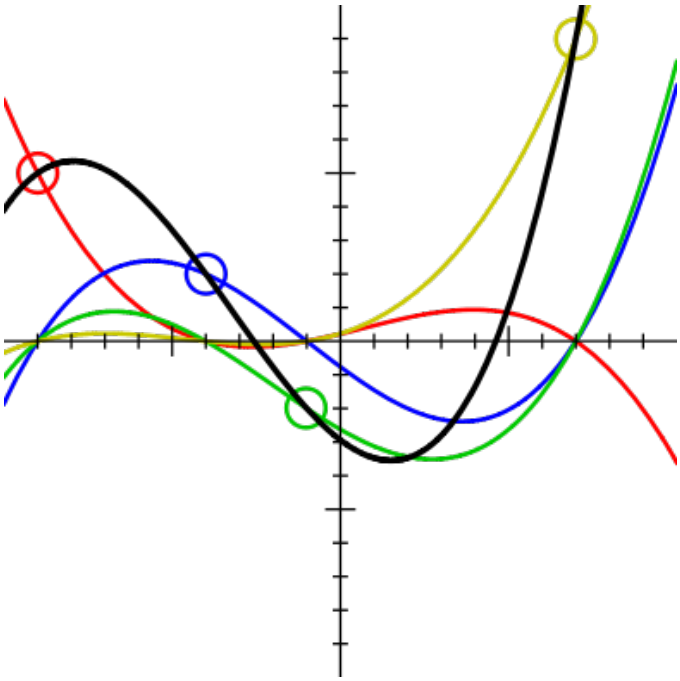

Figura 5 – Interpolação polinomial de Lagrange com k = 4

Fonte: http://en.wikipedia.org/wiki/File:Lagrange_polynomial.svg

Sem perda de generalidade, é possível assumir que o segredo D é (ou pode ser feito) um número. Para dividir o mesmo em D_i partes, o primeiro passo é a geração de um polinômio de grau k-1 no qual $a_0 = D$ cuja forma geral é:

$$q(x) = a_0 + a_1 x + ... + a_{k-1} x^{k-1} .$$

Afim de tornar o esquema mais preciso, faz-se uso de aritmética modular ao invés de aritmética real. Desta forma, todas as operações aritméticas são feitas dentro de um corpo finito F de tamanho $0 < k \leq n < P$, denotado por F_p, onde P é um número

primo que deve ser grande. O conjunto dos inteiros módulo um número primo P forma um corpo no qual a interpolação é possível [17]. É plausível que o tamanho desse primo seja determinado de acordo com as necessidades de segurança da aplicação. Um maior detalhamento sobre a teoria de corpos finitos é dado na seção 3.2.1.

4.1.2 Distribuição das *Shares*

A cada participante do esquema é atribuído por meio de um canal seguro uma *share*, que será um um ponto do polinômio $q(x)$ gerado, do tipo $(x, q(x))$. O primeiro passo para a geração desses pontos será a escolha de elementos únicos para os valores de x, valores estes que podem ser escolhidos randomicante ou sistematicamente dentro do corpo finito definido. No esquema original proposto por Shamir esses valores são sistemáticos, variando de 1 a n, onde n é o número de participantes do esquema. Os valores escolhidos, que podem ser públicos, são atribuídos a cada um dos participantes [25].

O próximo passo é a escolha de k-1 valores randomicamente a partir de uma distribuição uniforme dentro do corpo finito F para os coeficientes $a_1, \ldots, a_{k-1}$ do polinômio $q(x)$. Uma vez que todos esses valores tenham sido determinados, será dado a cada participante um ponto sobre o polinômio $q(x)$, que é sua respectiva *share*, isto é, sua parte do segredo D. Este ponto tem como coordernada x o valor previamente atribuído ao participante no primeiro passo, enquanto a coordenada y será dada por:

$$y = D + \sum_{j=1}^{k-1} a_j x^j \bmod P$$

4.1.3 Reconstrução do Segredo

Uma vez que todas as *shares* tenham sido atribuídas aos participantes do esquema, quando um grupo de pelo menos k participantes desejar reconstruir o segredo, eles utilizarão os valores de suas shares no polinômio $q(x)$. Cada participante irá criar uma equação com k valores de a desconhecidos, o que resultará em um sistema de equações que possui única solução. Uma vez resolvido o sistema, o segredo será simplesmente o valor de a_0. Afim de exemplicar este processo, damos abaixo um trivial exemplo de um (3,4)-*threshold scheme* baseado no esquema de Shamir sobre o corpo $\mathbb{Z}_{19}$.

$$x_1 = 1,\ x_2 = 2,\ x_3 = 3,\ x_4 = 4$$

$$\text{Randomicamente escolhidos: } D = a_0 = 12,\ a_1 = 14,\ a_2 = 3$$

Com esses valores, o polinômio pode ser gerado e sua saída para cada um dos valores de x_i pode ser determinada. Seguindo o exemplo dado, o polinômio do esquema será:

$$q(x) = 12 + 14x + 3x^2 \ mod\ 19$$

Através do polinômio gerado e dos valores de x_i, as *shares* podem ser obtidas e atribuídas aos participantes. No nosso exemplo, temos:

$$q(1) = 10,\ q(2) = 14,\ q(3) = 5,\ q(4) = 2$$

Para a reconstrução, cada participante gera uma equação utilizando os valores x e y de suas *shares*. Com os pontos dados acima, as equações obtidas para cada participante são, respectivamente:

$$10 = a_0 + a_1 + a_2$$
$$14 = a_0 + 2a_1 + 4a_2$$
$$5 = a_0 + 3a_1 + 9a_2$$
$$2 = a_0 + 4a_1 + 16a_2$$

Embora não mostrado aqui, quando quaisquer três das equações acima são simultaneamente resolvidas em $\mathbb{Z}_{19}$ como um sistema de equações lineares, os valores de a_i obtidos são 12, 14 e 3, respectivamente.

Na prática, é sempre utilizada uma segunda forma de efetuar a reconstrução do segredo derivada da, já citada, Interpolação de Lagrange. A razão é porque não estamos interessados em todos os coeficientes do sistema, mas sim somente no coeficiente livre a_0 que representa o segredo D. Para revelar o segredo, k participantes utilizam seus pontos (x_i, y_i) e calculam o valor do polinômio no zero usando interpolação:

$$D = \sum_{j=1}^{k} y_i l_j(x) \bmod P$$

$$l_j(x) = \prod_{1 \le t \le k.t \ne j} \frac{x_{i_t}}{x_{i_t} - x_i} \bmod P$$

4.1.4 Análise de Segurança

Agora, com o esquema de Shamir definido, uma análise sobre a segurança do esquema pode ser realizada. A primeira análise é determinar se este é ou não um esquema de segredo comapartilhado perfeito. A fim de responder essa questão, precisamos analisar as informações sobre o segredo a que k-1 participantes têm acesso combinando suas *shares*. De posse de apenas k-1 *shares*, os participantes nunca estarão aptos para determinar precisamente quaisquer dos coeficientes a_i do polinômio $q(x)$ utilizado. A única chance seria adivinhar esses valores, que podem ser qualquer valor dentro do corpo finito utilizado.

No entanto, esse processo para adivinhar os valores de a_i por força bruta é o mesmo e único modo que uma pessoa maliciosa poderia utilizar com nenhuma das *shares* do sistema em mãos. Por causa disto, o esquema de Shamir é um esquema de segredo compartilhado perfeito, pois k-1 *shares* não dão qualquer informação a respeito do segredo, ou seja, são tão úteis como não possuir qualquer uma delas.

4.1.5 Propriedades

Algumas propriedades dos (k,n)-*threshold schemes* de Shamir são:

1. **Mínimo:** O tamanho de cada *share* não excede o tamanho do segredo (esquema de shamir é um esquema *ideal*).
2. **Extensível:** quando o *threshold* (ou limiar) k é mantido fixo, novas *shares* podem ser geradas ou *shares* existentes serem excluídas sem afetar as já existentes.
3. **Dinâmico:** a segurança pode ser facilmente aumetada sem mudar o segredo, apenas gerando ocasionalmente um novo polinômio $q(x)$ (com o mesmo coeficiente livre a_0) e gerando novas *shares* aos participantes.
4. **Flexível:** em organizações onde uma hierarquia é importante, é possível fornecer a cada participante uma quantidade diferente de *shares* de acordo com sua importância dentro da organização. Por exemplo, suponha que sejam dadas três *shares* ao presidente da organização, duas ao vice-presidente e uma única *share* para cada um dos demais executivos. Em um (3,n)-*threshold scheme*, o segredo poderia ser recuperado pelo presidente individualmente, pelo vice-presidente e mais um executivo ou ainda por três executivos em conjunto.

4.2 Esquema de Blakley

No mesmo ano da publicação do esquema de Shamir, George Blakley publicou seu próprio esquema de segredo compartilhado baseado na geometria de hiperplanos sobre corpos finitos. De forma similar ao esquema de Shamir, o esquema de Blakley é também um *threshold scheme* baseado, no entanto, na intersecção de hiperplanos, ao invés de interpolação polinomial.

Assim como duas observações sobre polinômios fundamentam o funcionamento do esquema de Shamir, os seguintes fatos a respeito de hiperplanos são usados na construção do esquema de compartilhamento de segredos de Blakley:

- Em um espaço vetorial de n dimensões, n hiperplanos não paralelos se interceptam em um único ponto;

- Em um espaço vetorial de n dimensões, n - 1 hiperplanos não paralelos se interceptam em infinitos pontos [24];

4.2.1 Ideia Geral

Em um *(k,n)-threshold scheme* segundo o que foi proposto por Blakley, o segredo é codificado como um ponto em um espaço k-dimensional e n *shares* são definidas como hiperplanos que passam por este ponto. Os hiperplanos usados para este esquema serão todos de k dimensões, o que permite que k dos hiperplanos se interceptem em único ponto, dentro do corpo finito. O segredo pode ser codificado como qualquer coordenada individual deste ponto de intersecção dos hiperplanos.

Abaixo, temos a exemplificação do esquema de Blakley em 3 dimensões (Figura 6). Cada *share* (parte do segredo) é um plano, e o segredo é o ponto único no qual os três planos se interceptam. Apenas duas *shares* são insuficientes para determinar o segredo.

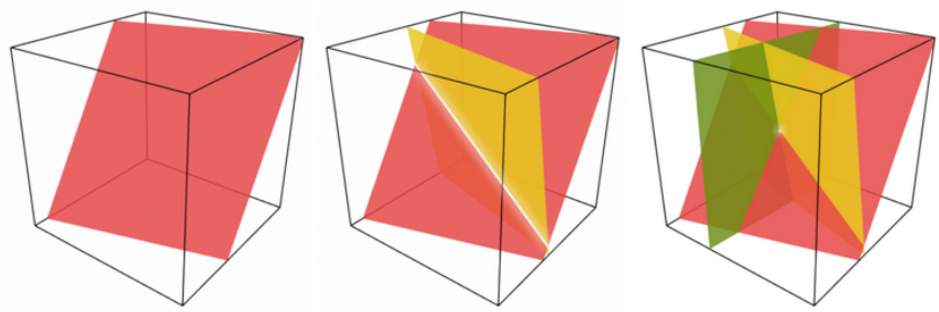

Figura 6 – Exemplo do esquema de Blakley em 3 dimensões.

Fonte: http://en.wikipedia.org/wiki/Secret_sharing

4.2.2 Distribuição das *Shares*

A distribuição das *shares* no esquema de Blakley começa com a geração de um ponto secreto k-dimensional em um corpo finito k-dimensional, F_p. Independente da forma como este ponto, rotulado como x, esteja localizado, a primeira coordenada do ponto é setada como o segredo sendo compartilhado. Uma vez que o ponto é determinado, a *share* de cada participante pode ser determinada: será a equação de um hiperplano k-dimensional que passa pelo ponto x gerado.

O primeiro passo para a geração da *share* de um participante é a geração de k valores de a, necessários para a determinação da equação do hiperplano. Usando o ponto secreto e os valores de a, a *share* de cada participante é gerada por por:

$$\{(x_1, ..., x_k) \in F_p | a_1.x_1 + ... + a_k.x_k = y\} \tag{4.1}$$

Uma vez geradas as *shares*, que correspondem ao termo y da equação do hiperplano como calculado acima, estas podem ser atribuídas aos participantes. Os diferentes valores de a gerados podem ser públicos, pois não são sensíveis para a segurança do esquema [26].

4.2.3 Reconstrução do Segredo

No esquema de Blakley, o segredo é recuperado por meio do cálculo do ponto de intersecção dos hiperplanos de k participantes, onde k é o *threshold* especificado. O segredo é então recuperado simplesmente tomando uma coordernada especificada do respectivo ponto.

Para a reconstrução do segredo neste esquema, os k participantes que desejam reconstruir o segredo irão combinar suas *shares*. Cada um deles obterá a equação de um hiperplano com seus correspondentes valores de a e y, mas com os valores de x desconhecidos. Cada equação será da mesma forma como apresentado em (4.1). Com os k participantes, uma matriz de equações será gerada combinando todas as *shares*:

$$Ax = y \tag{4.2}$$

Solucionando o sistema de equações, são obtidos k valores de x, os quais correspondem às coordenadas do ponto secreto. O segredo pode então ser recuperado simplesmente tomando a primeira coordenada do ponto, tendo em vista que o segredo foi assim codificado no ponto de intersecção dos planos.

É possível notar similaridades entre esta forma de reconstrução do segredo e o modo como a mesma é realizada no esquema de Shamir. No esquema de Blakley, no entanto, os participantes obtém os valores de x diretamente, ao invés dos coeficientes a do polinômio no esquema de Shamir. Para o caso do esquema de Blakley, não são conhecidos métodos mais simples para a reconstrução do segredo, além de métodos alternativos para resolver o sistema de equações [25].

4.2.4 Análise de Segurança

Como cada uma das *shares* é um valor na mesma faixa que as coordenadas do corpo finito, e como uma das coordenadas é o segredo, este esquema tem um *information rate* igual a 1, assim como o esquema de Shamir. No entanto, este esquema não é *ideal* devido ao fato de que nao é um esquema de segredo compartilhado perfeito. No caso do esquema de Blakley, isso decorre do fato de que à medida que o número de *shares* combinadas aumenta, o número de possibilidades para o ponto secreto diminui. Por exemplo, cada

dos participantes já sabe de antemão que o ponto secreto está em seu hiperplano, o que reduz a quantidade de pontos possíveis.

Além disso, embora não seja possível determinar o ponto secreto com apenas k-1 *shares*, é possível determinar a reta sobre a qual o ponto secreto está localizado, o que facilitaria um ataque de força bruta para descobrir o ponto secreto. Isso faz de Blakley um esquema não perfeito, porque a segurança do esquema diminui com k-1 *shares*. Como já dito, o ideal para um esquema de segredo compartilhado é que k-1 *shares* não forneçam absolutamente qualquer informação sobre o segredo [27].

5 Tecnologias Utilizadas

As principais tecnologias e recursos que foram utilizados na realização deste trabalho são apresentados nas subseções abaixo.

5.1 Linguagem de Programação Java

Java é uma linguagem de programação de alto nível que utiliza o paradigma de programação orientado a objetos. Essa linguagem proporciona portabilidade e independência de plataforma às suas aplicações. Isso acontece, devido ao fato da compilação de um código Java gerar código intermediário, conhecido como *bytecodes*. Esse código intermediário é executado pela Máquina Virtual Java (JVM). Sendo assim, é possível executar uma aplicação implementada em Java em qualquer plataforma que tenha uma JVM instalada.

Outro ponto a favor da linguagem Java é a manutenção de uma biblioteca de classe (APIs) extensa e bem documentada, gerando um excelente suporte aos usuários [28].

5.2 Biblioteca JLinAlg

A JLinAlg[1] é uma biblioteca de funções matemáticas desenvolvida na linguagem Java pelo projeto *JLinAlg-project*. A biblioteca tem código aberto e é licenciada pela *GNU General Public License* (GPL)[2].

A JLingAlg foi projetada para ser capaz de realizar diversas operações da álgebra linear, entre elas:

- Operações básicas sobre matrizes e vetores, como produto escalar e multiplicação de matrizes.
- Calcular o inverso ou determinante uma matriz.
- Calcular uma solução ou o espaço de soluções de um sistema de equações lineares.

5.3 Subversion (SVN)

O *Subversion* (SVN) é um sistema de controle de versões que permite aos usuários verificar mudanças feitas em arquivos no formato digital. O SVN foi projetado para

[1] Para maiores informações, consulte http://jlinalg.sourceforge.net/
[2] Para maiores informações, consulte http://www.gnu.org/licenses/gpl.html

substituir o *Concurrent Version System* (CVS)[3] [29].

Com o SVN, é possível que várias pessoas trabalhem simultaneamente em arquivos do projeto sem comprometer uma o trabalho da outra, pois os arquivos ficam armazenados num servidor. Neste sistema, um log registra as modificações feitas no projeto mesclando as alterações feitas. O controle de versões permite que, em caso de ocorrência de alguma falha em alguma parte do projeto, seja possível retomar versões anteriores dos arquivos.

5.4 Visual Paradigm for UML

A *Visual Paradigm for UML* (VP-UML)[4] é uma ferramenta CASE e de design UML projetada para auxiliar no desenvolvimento de software. A VP-UML suporta os principais padrões de modelagem, tais como *Unified Modeling Language* (UML) 2.4, SoaML, SysML, ERD, DFD, BPMN 2.0, ArchiMate 2.0, entre outros. A ferramenta é projetada para dar suporte em várias etapas no desenvolvimento de software, como captura de requisitos, planejamento (análise de caso de uso), engenharia de código, modelagem de classes, modelagem de dados, etc.

Além do apoio na modelagem, proporciona a geração de relatórios e recursos de engenharia de código, incluindo a geração de código. Ele pode efetuar a criação de diagramas a partir de código fonte implementado, e fornece engenharia *round-trip* para várias linguagens de programação.

A ferramenta foi utilizada para geração do diagrama de classes da aplicação desenvolvida, assim como para a geração de todos os demais diagramas UML apresentados ao longo deste trabalho.

[3] Para maiores informações, consulte http://pt.wikipedia.org/wiki/CVS
[4] Versão *trial* pode ser obtida em http://www.visual-paradigm.com/download/vpuml.jsp

6 Implementação da Biblioteca

6.1 Introdução

Como foi comentado na seção 1.3, o objetivo deste trabalho consiste na implementação de uma biblioteca que implementasse diferentes mecanismos de segredo compartilhado. Alguns mecanismos de segredo compartilhado foram estudados, sendo que dois mecanismos foram escolhidos para implementação, ambos explicados no Capítulo 4. O primeiro deles é o esquema de segredo compartilhado proposto por Shamir, baseado na interpolação polinomial de Lagrange; o segundo esquema implementado que integra a biblioteca final é o esquema de segredo compartilhado proposto por George Blakley, baseado na geometria de hiperplanos.

A escolha desses dois mecanismos de segredo compartilhado deu-se pelo fato de que ambos são os mecanismos de segredo compartilhado mais tradicionais e amplamente conhecidos. Como já descrito nas seções anteriores, cada um dos métodos tem seu modo de funcionamento em particular, empregando diferentes meios para divisão e posterior reconstrução de um segredo. Porém, de uma forma geral, o objetivo de cada um dos métodos é o mesmo: dividir um determinado segredo D em n partes, sendo que quaisquer k partes são suficientes para reconstruí-lo, onde $k \leq n$. Por isso, ambos são classificados como esquemas (k, n) de compartilhamento. São estes os dois mecanismos de segredo compartilhado implementados que integram a biblioteca desenvolvida neste trabalho.

Uma biblioteca de software pode ter sua importância exemplificada com o cenário dado a seguir: ao escrever um programa usando uma linguagem de programação, existe a possibilidade de reutilização de um conjunto de funções pré-escritas por outros programadores que já resolveram determinados problemas, fazendo com que quem está programando não precise criar uma nova solução, mas sim utilizar uma já existente. A esse conjunto de funções damos o nome de biblioteca, do inglês, *library*. É justamente essa a ideia desse trabalho: prover uma biblioteca de software que poderá ser utilizada por outros usuários e aplicações que necessitem da implementação de mecanismos de segredo compartilhado, mais especificamente por softwares desenvolvidos pelo LabSEC que fazem uso de mecanismos desse tipo.

A linguagem de programação escolhida para implementação da biblioteca foi a linguagem Java. A escolha pode ser justificada pela fluência com a linguagem, pela possibilidade de desenvolver a aplicação baseado no paradigma de orientação a objetos e pelo fato de a linguagem Java fornecer bibliotecas matemáticas necessárias para a implementação dos mecanismos de segredo compartilhado, tornando a implementação

dos mesmos mais simples e eficiente.

Nas seções seguintes, temos dois importantes diagramas gerados como parte da modelagem estrutural da biblioteca de software implementada, assim como a descrição completa da aplicação desenvolvida, com detalhes de funcionamento e implementação, bem como exemplos de uso das funções da biblioteca.

6.2 Modelagem

Como parte da modelagem estrutural da biblioteca de software desenvolvida, temos o diagrama de classes de projeto e um diagrama de pacotes da aplicação. Ambos podem ser consultados a seguir.

6.2.1 Modelagem Estrutural

O diagrama de classes de projeto da biblioteca desenvolvida pode ser consultado na Figura 7.

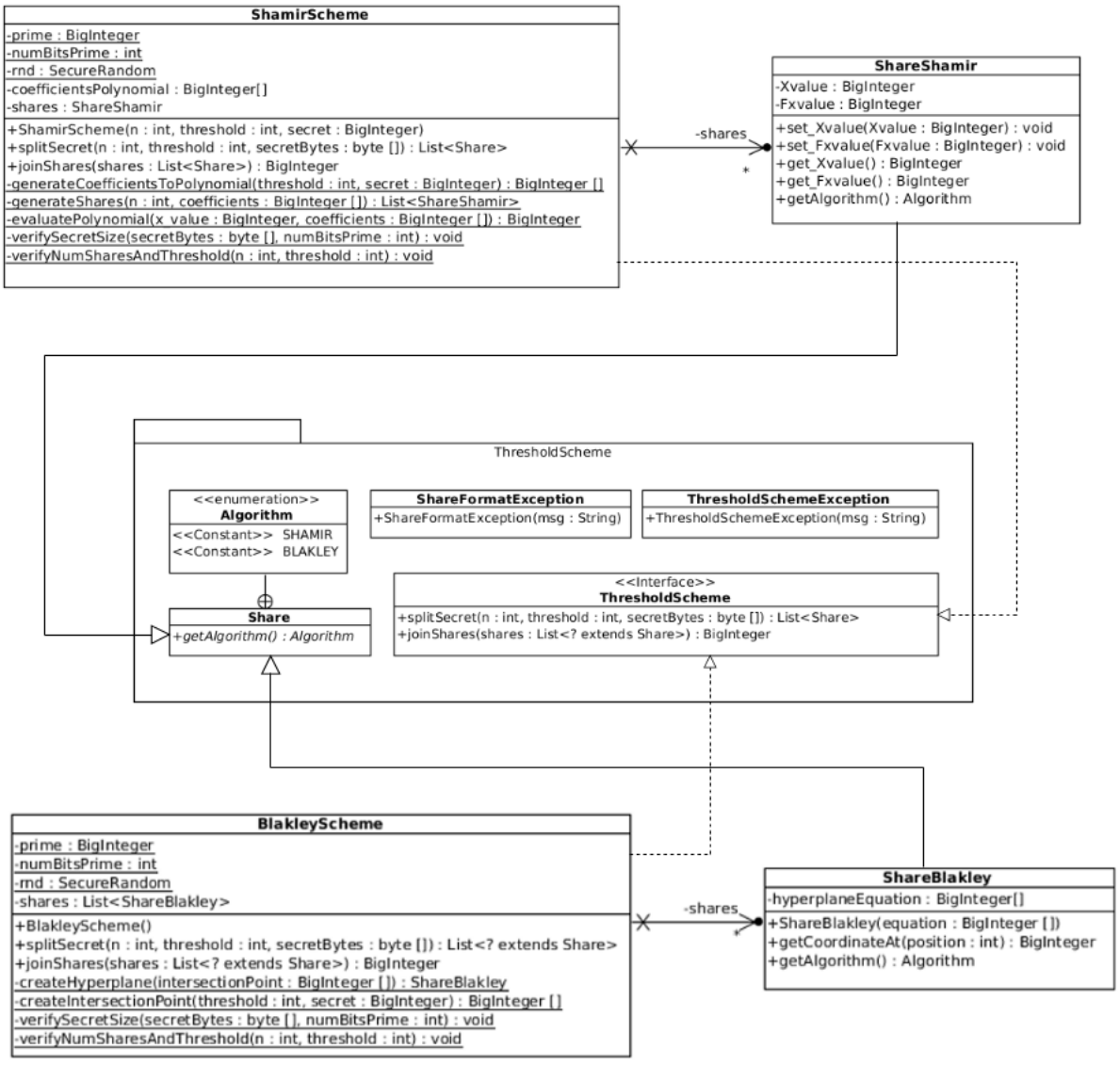

Figura 7 – Diagrama de classes de projeto da biblioteca

Com relação à estrutura em pacotes, a aplicação compreende três pacotes, conforme ilustrado na Figura 8. A seguir, explicaremos de forma breve cada uma das classes ilustradas no diagrama de pacotes que segue logo abaixo. Mais detalhes podem ser obtidos na seção 6.3.

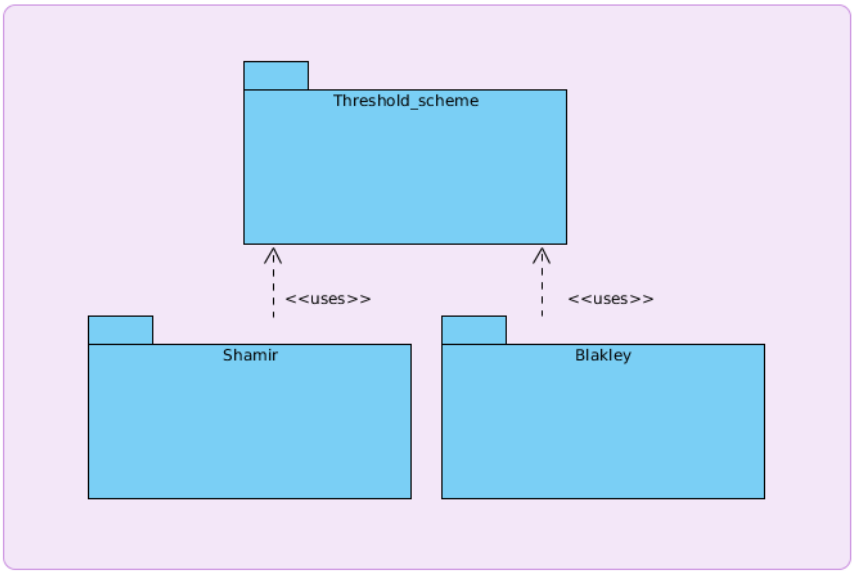

Figura 8 – Diagrama de pacotes da biblioteca

No pacote rotulado como *Threshold_scheme*, temos as seguintes classes:

- **ThresholdScheme**: interface que define um método para divisão do segredo e um método para reconstrução do segredo. Essa interface é implementada pelos dois mecanismos de segredo compartilhado que fazem parte da biblioteca e deverá ser implementada por qualquer outra implementação de mecanismos de segredo compartilhado desse tipo que venha ser adicionada a biblioteca.
- **ThresholdSchemeException**: classe utilizada para que seja possível o lançamento de exceções que precisam ser consideradas na implementação de mecanismos de segredo compartilhado baseados em *threshold*.
- **Share**: classe abstrata para representação de uma *share*, ou parte de um dado segredo. Cada um dos esquemas de segredo compartilhado implementados utilizam classes próprias para a representação de suas *shares*. Estas classes, que serão posteriormente citadas, extendem à classe *Share*.
- **ShareFormatException**: classe utilizada para o tratamento de exceções relacionadas a problemas com o formato de representação de uma dada *share*.

No pacote rotulado como *Shamir*, estão todas as classes relacionadas à implementação do esquema de segredo compartilhado de Shamir. Temos as seguintes classes:

- **ShamirScheme**: classe responsável pela implementação do esquema de Shamir como um todo. Nesta classe estão os métodos que efetuam todas as principais operações deste mecanismo de segredo compartilhado, como a divisão do segredo, sua reconstrução, assim como outros métodos auxiliares. Mais detalhes de implementação e funcionamento podem ser obtidos na subseção 6.3.2.

- **ShareShamir**: classe para a representação de uma *share* em formato adequado para o esquema de Shamir.

Por fim, no pacote rotulado como *Blakley*, estão todas as classes relacionadas à implementação do esquema de segredo compartilhado de Blakley. Temos as seguintes classes:

- **BlakleyScheme**: de forma simplificada, é a classe responsável pela implementação de todas as funcionalidades para o esquema de segredo compartilhado de Blakley. Nesta classe estão os métodos que efetuam todas as principais operações deste mecanismo de segredo compartilhado, como a divisão do segredo, sua reconstrução, assim como outros métodos auxiliares. Mais detalhes de implementação e funcionamento podem ser obtidos na subseção 6.3.3.

- **ShareBlakley**: classe para a representação de uma *share* em formato adequado para o esquema de Blakley.

6.3 Descrição da Implementação

Nesta seção temos uma completa descrição da parte prática do trabalho, onde apresentamos e detalhamos a implementação da biblioteca de segredo compartilhado produzida. Apresentaremos todos os detalhes de implementação de cada um dos mecanismos de segredo compartilhado que compõem a biblioteca, os quais já foram explicados e detalhados no Capítulo 4, bem como outros detalhes e classes que foram implementadas para tornar a biblioteca padronizada para a adição de outros mecanismos para compartilhamento de segredos que possam a vir ser adicionados na mesma. As soluções que, eventualmente, foram aqui omitidas podem ser encontradas no código fonte anexo ao projeto no Apêndice A.

Nas seções seguintes, descrevemos separadamente: a implementação de cada um dos mecanismos de segredo compartilhado implementados, a implementação de uma interface genérica para mecanismos de segredo compartilhado, bem como outros aspectos relevantes da implementação da biblioteca em questão.

6.3.1 Interface para Threshold Secret Sharing Schemes

Antes de explicarmos efetivamente a interface Java desenvolvida aqui e qual sua importância desenvolvida dentro da biblioteca implementada, vale relembrar alguns conceitos importantes relativos à interfaces na linguagem de programação Java.

Interface é um recurso da orientação a objeto utilizado em Java que define ações que devem ser obrigatoriamente executadas, mas que cada classe pode executar de forma diferente. Interfaces contém valores constantes ou assinaturas de métodos que devem ser implementados dentro de uma classe. Isso se deve ao fato que muitos objetos (classes) podem possuir a mesma ação (método), porém, podem executá-la de maneira diferente. Uma interface estabelece uma espécie de contrato que é obedecido por uma classe. Quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

Relembrado o conceito de interface, um dos primeiros aspectos que considerou-se importante no desenvolvimento de uma biblioteca de mecanismos de segredo compartilhado foi a existência de uma interface Java genérica para mecanismos de segredo compartilhado baseados em um *threshold*. Isso deu-se pelo fato de que, embora cada esquema de segredo compartilhado baseado em um *threshold* utilize diferentes formas para dividir e posteriormente reconstruir um segredo, todo esquema deste tipo deve obrigatoriamente possuir rotinas que implementem cada uma destas funcionalidades, independentemente do modo como isso seja feito.

O fato acima citado justifica a criação dessa interface Java implementada para mecanismos de segredo compartilhado baseados em um *threshold*, a qual define: um método para divisão do segredo e um método para reconstrução do segredo. Essa interface é implementada pelos dois mecanismos de segredo compartilhado que fazem parte da biblioteca e deverá ser implementada por qualquer outra implementação de mecanismos de segredo compartilhado desse tipo que venha ser adicionada a biblioteca.

Em termos de implementação, em ambos os mecanismos de segredo compartilhado baseados em *threshold*, os métodos que fazem a divisão e reconstrução do segredo terão a mesma assinatura definida pela interface que implementam. O método que efetua a divisão do segredo sempre recebe os seguintes parâmetros:

- **segredo**: o segredo a ser dividido, recebido como um array de *bytes*.
- **n**: um número inteiro, que representa a quantidade de *shares* que serão geradas para o dado segredo.
- **threshold ou k**: um número inteiro, que representa a quantidade de *shares* que serão necessárias para efetuar a reconstrução do segredo em um esquema (k, n) de compartilhamento.

O método de divisão do segredo tem como retorno n partes do segredo, ou *shares*, retornadas em uma lista da classe *ArrayList*[1] do Java, implementação da interface *List*[2].

O método de reconstrução do segredo, por sua vez, deve receber como parâmetro uma lista com k partes do segredo. A partir destas *shares*, o segredo é reconstruído por diferentes métodos em cada um dos esquema de segredo compartilhado implementados. O segredo é representado com um objeto da classe *BigInteger*[3] do Java, utilizada para representar números inteiros arbitrariamente grandes.

O diagrama de classes com a interface acima mencionada e demais classes do mesmo pacote são mostrados na Figura 9.

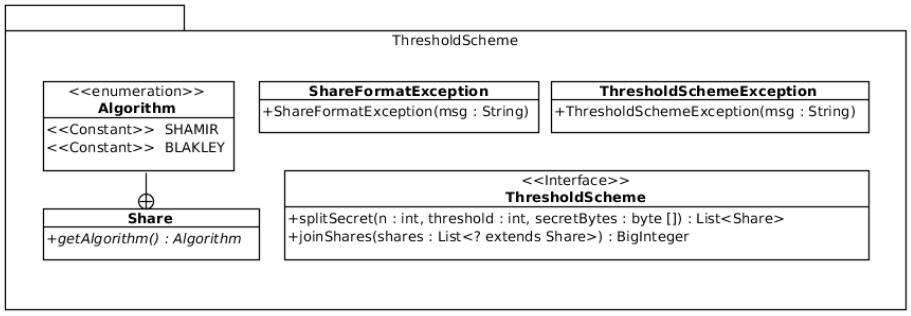

Figura 9 – Diagrama de classes para um *Threshold Scheme* genérico

Nas próximas seções, apresentamos a descrição detalhada da implementação dos dois mecanismos de segredo compartilhado implementados.

6.3.2 Esquema de Shamir

A implementação do esquema de Shamir é realizada por uma classe específica, que, como já comentado, implementa a interface para mecanismos de segredo compartilhado baseados em *threshold* que foi implementada.

O primeiro passo da implementação é a definição de alguns aspectos necessários para a implementação do mecanismo, definidos no construtor da classe. Dentre os aspectos podemos citar a definição do número primo q sobre o qual serão realizados todas as operações de aritmética modular, conforme já explicado na seção 4.1, assim como a definição de um gerador de números aleatórios que será necessário nos passos posteriores da implementação. Para a geração dos números aleatórios são utilizadas duas classes em

[1] Para maiores informações, consulte http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

[2] Para maiores informações, consulte http://docs.oracle.com/javase/6/docs/api/java/util/List.html

[3] Para maiores informações, consulte http://docs.oracle.com/javase/1.4.2/docs/api/java/math/BigInteger.html

conjunto: a classe *SecureRandom*[4] e a classe *BigInteger* do Java. A classe *SecureRandom* fornece um gerador de números aleatórios criptograficamente forte.

Tendo sido definidos esses aspectos iniciais, temos as duas funcionalidades básicas implementadas no mecanismo de Shamir que poderão ser utilizadas pelos usuários da biblioteca: a divisão e posterior reconstrução do segredo.

6.3.2.1 Divisão do segredo

A sequência das etapas realizadas para a divisão de um segredo são ilustradas pelo diagrama de atividade na Figura 10 e são mais detalhadamente explicadas a seguir.

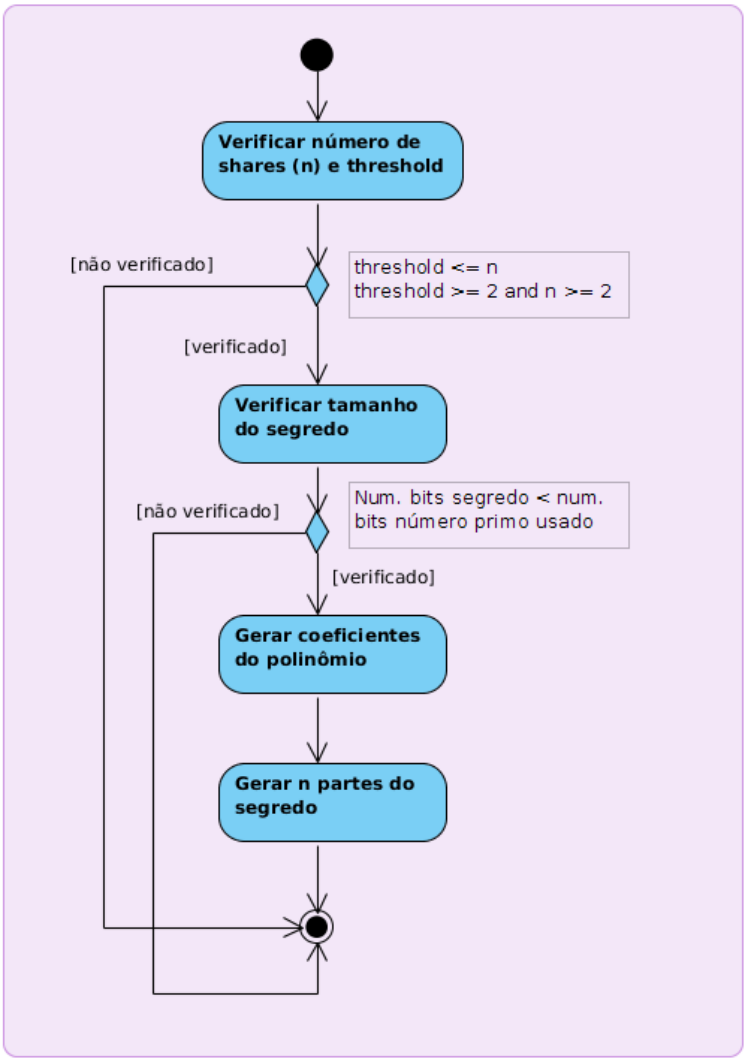

Figura 10 – Diagrama de atividades da divisão do segredo pelo Esquema de Shamir

Quando a rotina para divisão de um segredo for invocada, três parâmetros são necessários: n, ou seja, o número de partes em que deseja-se dividir o segredo; k,

[4] Para maiores informações, consulte http://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html

representando o número de mínimo de partes do segredo que será necessário para reconstruí-lo; e por fim, o segredo D a ser dividido, que deve ser passado como um array de *bytes*.

Conforme ilustrado no diagrama de atividades da Figura 10, antes da divisão do segredo ser realizada, é necessário que sejam realizadas algumas verificações sobre os parâmetros utilizados ao invocar a função de divisão do segredo. Se qualquer uma das verificações falhar, a divisão do segredo não pode ser realizada.

A primeira verificação consiste na validação dos valores de n (número total de *shares* desejado) e k (*threshold*): não é permitido, por exemplo, que o valor de k seja maior do que n, podendo ser no máximo igual a n, no caso onde todas as partes do segredo serão necessárias para reconstruí-lo; uma segundo verificação é atestar-se que ambos os valores de n e k sejam no mínimo iguais a dois, pois não faz sentido a divisão de segredo entre um número de participantes menor do que este. A segunda verificação consite em validar o segredo que deseja ser dividido. O segredo não pode ser maior do que o número primo utilizado pela aplicação, para que seja possível representá-lo dentro do corpo finito determinado pelo número primo que está sendo utilizado.

Validados todos os dados passados como parâmetro para a rotina de divisão do segredo, esta pode efetivamente ser executada. Quando invocada e após a validação de todos os parâmetros, o próximo passo é a escolha de k-1 valores randomicamente a partir de uma distribuição uniforme dentro do corpo finito determinanado por q para os coeficientes $a_1, \ldots, a_{k-1}$ de um polinômio $q(x)$. Lembrando que o coeficiente livre a_0 do polinômio $q(x)$ é setado como sendo o segredo D. A geração desses valores aleatórios é realizada através da, já citada, classe *BigInteger* do Java. Esta fornece um método construtor, para o qual podemos passar um gerador de números aleatórios da classe *SecureRandom* e um número de bits x, e o mesmo retorna um valor aleatório uniformemente distribuído na faixa entre 0 e 2^x - 1, inclusive.

Gerados os coeficientes do polinômio $q(x)$ para o segredo D sendo compartilhado, é possível efetuar a geração das diferentes partes do segredo, isto é, das diferentes *shares* que serão atribuídas a cada um dos participantes do esquema de segredo compartilhado. Aqui não há muito o que ser comentado, haja vista que a implementação segue examente o processo de distribuição das *shares* segundo o esquema de Shamir, como explicado na subseção 4.1.2. Cada *share* é um ponto do polinômio $q(x)$ gerado, do tipo $(x, q(x))$. Cada uma das *shares* têm um valor de x único, escolhido randomicamente dentro do corpo finito definido. Já o valor de $q(x)$ da *share*, é calculado segundo a implementação da avaliação do polinômio gerado no valor de x em questão, conforme explicado na subseção anteriormente citada. Por fim, seguindo o mesmo processo, n partes para o segredo são calculadas e retornadas como uma lista do tipo *ArrayList*.

6.3.2.2 Reconstrução do segredo

Explicado a divisão do segredo e geração das *shares*, a segunda funcionalidade disponível é a reconstrução do segredo, realizada a partir de um conjunto de tamanho k de *shares*, onde k é o *threshold* estabelecido pelo esquema. Conforme explicado na subseção 4.1.3, há duas formas possíveis para a reconstrução do segredo segundo o esquema de Shamir: a primeira seria através da solução de um sistema de equações lineares; a segunda, e mais eficiente, através de um método derivado da Interpolação de Lagrange. Por questões de eficiência computacional, a biblioteca implementada utilizada esta segunda forma para reconstruir o segredo D.

A sequência das etapas realizadas para a reconstrução de um segredo são ilustradas de forma simplificada pelo diagrama de atividade na Figura 11 e são mais detalhadamente explicadas a seguir.

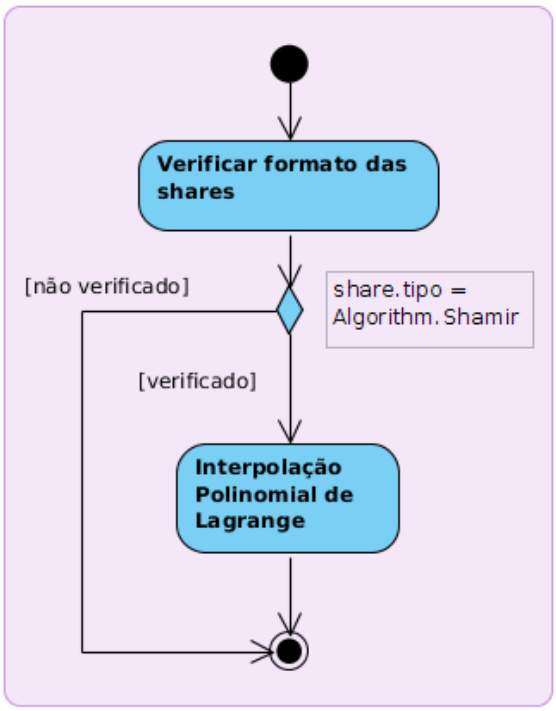

Figura 11 – Diagrama de atividades da reconstrução do segredo pelo Esquema de Shamir

A rotina que efetua a reconstrução do segredo precisa receber uma lista do tipo *ArrayList* contendo k *shares*, previamente geradas, para que o segredo D original possa ser recuperado. Não ocorrerá qualquer tipo de erro caso a lista recebida contenha uma quantidade inferior de *shares*, no entanto a informação retornada pela rotina de reconstrução do segredo não corresponderá ao segredo original inicialmente compartilhado.

O primeiro passo na rotina de reconstrução é verificar se as *shares* recebidas como parâmetro estão no formato exigido pela biblioteca. As partes do segredo precisam, a nível de funcionamento da biblioteca, corresponder a objetos da classe criada especificamente

para representá-las, para que a chamada de métodos e funcionamento correto da rotina possa ser garantido. Caso as *shares* recebidas não estejam no formato apropriado, uma exceção é lancada e a rotina é encerrada.

Após cada uma das *shares* ter sido verificada, a Interpolação de Lagrange pode ser iniciada para a reconstrução do segredo. Como já explicado anteriormente, no caso do esquema de segredo compartilhado de Shamir, o segredo é codificado como sendo o coeficiente livre do polinômio $q(x)$ gerado. Para a reconstrução do segredo, as k *shares*, que são pontos do tipo (x_i, y_i) sobre o polinômio $q(x)$ gerado, serão utilizadas para calcular o valor do polinômio no zero usando interpolação, obtendo assim o coeficiente livre a_0 correspondente ao segredo.

Todas as operações matemáticas necessárias para a implementação da interpolação polinomial foram realizadas utilizando a classe *BigInteger* do Java, que fornece métodos para adição, subtração, multiplicação, exponenciação e divisão, entre outros. Lembrando que como estamos trabalhando com todas as operações em aritmética modular, a operação de divisão é realizada de uma forma particular: quando deseja-se dividir um número a por um número b, temos que multiplicar a pelo inverso multiplicativo modular de b. A classe *BigInteger* oferece um método que calcula o inverso multiplicativo modular de um número de forma eficiente.

O diagrama de classes de todo estema esquema é apresentado na Figura 12 abaixo.

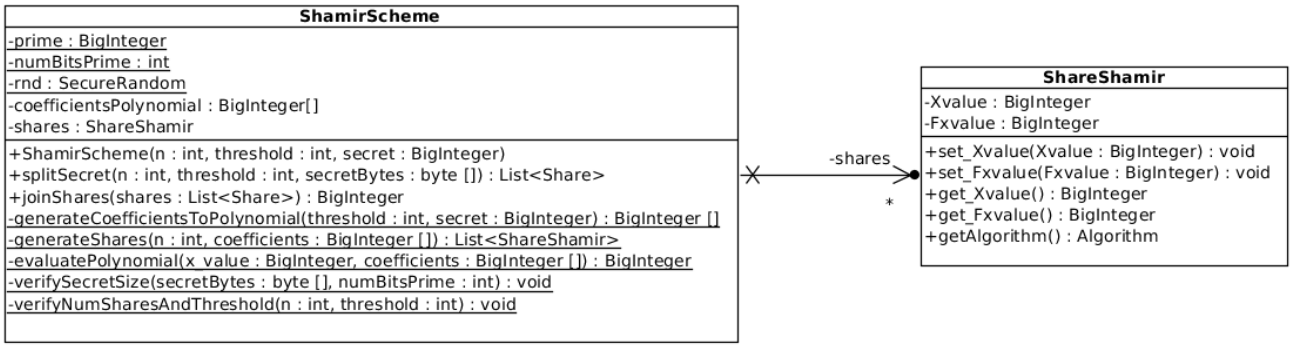

Figura 12 – Diagrama de classes da implementação do esquema de segredo compartilhado de Shamir

6.3.2.3 Exemplo de Utilização

Afim de exemplificar como as funções fornecidas pela biblioteca implementada devem ser utilizadas, damos a seguir um exemplo de utilização na qual efetua-se a divisão e reconstrução de um dado segredo pelo esquema de segredo compartilhado de Shamir.

```
1  public class ShamirTest {
2
3    final static int n = 5; // nro de usuarios participantes do esquema
4    final static int threshold = 3; // threshold do esquema
5
6    public static void main (String[] args) {
7
8      String secret = "Secret Sharing";
9      List<? extends Share> shares;
10     List<ShareShamir> sharesReconstrucao = new ArrayList<>(threshold);
11     ShamirScheme schemeTest = new ShamirScheme();
12
13     try {
14     shares = schemeTest.splitSecret (n, threshold, secret.getBytes());
15     } catch (ThresholdSchemeException ex) {
16        System.out.println(ex.toString());
17        return;
18     }
19     for (int i = 0; i < threshold; i++ ) {
20         ShareShamir aux = (ShareShamir) shares.get(i);
21         sharesReconstrucao.add(aux);
22     }
23     byte[] bytes = null;
24     try {
25     bytes = schemeTest.joinShares(sharesReconstrucao).toByteArray();
26
27     } catch (ShareFormatException ex) {
28         System.out.println(ex.toString());
29     }
30     secret = new String(bytes); // segredo obtido novamente
31   }
32 }
```

Para a utilização do esquema de segredo compartilhado de Shamir, os métodos fornecidos pela biblioteca são: um método construtor de classe, o método *splitSecret()* e *joinShares()*, utilizados no código mostrado nas linhas 11, 14 e 25, respectivamente.

Na próxima seção, temos o detalhamento da implementação do esquema de segredo compartilhado de Blakley.

6.3.3 Esquema de Blakley

A implementação do esquema de Blakley é realizada por uma classe específica, que, assim como no caso do esquema de Shamir, implementa a interface para mecanismos de segredo compartilhado baseados em *threshold* que foi implementada. Em termos de programação, a implementação do esquema de Blakley tem algumas semelhanças com a implementação do esquema de Shamir descrita na seção anterior, embora utilize mecanismos completamente diferentes para a divisão e posterior reconstrução do segredo.

Assim como na implementação do esquema de Shamir, o primeiro passo da implementação do esquema de Blakley é a definição de alguns aspectos necessários para a implementação do mesmo, que são definidos no construtor da classe. No construtor são definidos o número primo que determina o corpo finito sobre o qual são realizadas as operações, assim como a definição de um gerador de números aleatórios que será necessário nos passos posteriores da implementação. De forma similar ao esquema de Shamir, para a geração dos números aleatórios escolheu-se utilizar a classe *SecureRandom* do Java.

Conforme explicado na seção 4.2, onde o esquema de segredo compartilhado de Blakley foi explicado em detalhes, este é também um esquema de segredo compartilhado baseado em um *threshold*, assim como o esquema de Shamir. Contudo, este baseia-se na intersecção de hiperplanos, ao invés de interpolação polinomial. Em um *(k,n)-threshold scheme*, segundo o que foi proposto por Blakley, o segredo é codificado como um ponto em um espaço k-dimensional e n *shares* são definidas como hiperplanos que passam por este ponto. Os hiperplanos usados para este esquema serão todos de k dimensões, o que permite que k dos hiperplanos se interceptem em único ponto, dentro do corpo finito. O segredo pode ser codificado como qualquer coordenada individual deste ponto de intersecção dos hiperplanos. Para recuperar o segredo, basta efetuar a intersecção de k nos n hiperplanos gerados e tomar a coordenada especificada em que o segredo foi codificado.

6.3.3.1 Divisão do segredo

Relembrada a ideia básica de funcionamento do esquema de segredo compartilhado de Blakley, explicaremos a seguir a sequência dos passos realizados pela biblioteca implementada no momento da divisão de um segredo e geração das *shares*. Assim como no caso do esquema de Shamir explicado na seção anterior, antes que a divisão do segredo possa ser realizada, é necessário que sejam realizadas verificações sobre os parâmetros utilizados ao invocar a função de divisão do segredo. Se qualquer uma das verificações falhar, a divisão do segredo não pode ser realizada. Não vamos aqui abordar cada uma das verificações, tendo em vista que são as mesmas realizadas, e já explicadas, para o caso do esquema de Shamir.

Validados todos os dados passados como parâmetro para a rotina de divisão do

segredo, esta pode efetivamente ser executada. O primeiro passo é então a geração de um ponto em um espaço k-dimensional, ponto este em que o segredo a ser compartilhado será codificado e que será a intersecção dos n hiperplanos. O valor de k é o *threshold* do esquema, valor este que é passado como parâmetro para a rotina de divisão do segredo, assim como o valor de n (número total de *shares* que serão geradas), e o segredo propriamente dito, que deve ser passado como um array de *bytes*. A geração deste ponto, que é representado como um array de números da classe *BigInteger*, é simples: a primeira coordenada do ponto é setada como sendo o segredo a ser compartilhado, enquanto as demais são valores aleatórios gerados dentro do corpo finito definido pelo número primo utilizado pela aplicação.

Uma vez gerado o ponto de intersecção, as *shares* podem ser geradas, sendo que cada *share* é um hiperplano que contém o ponto de intersecção x. Cada hiperplano é gerado conforme a equação de um hiperplano apresentada na subseção 4.2.2, onde a distribuição das *shares* no esquema de Blakley foi explicada. A sequência das etapas realizadas para a divisão do segredo são ilustradas pelo diagrama de atividade na Figura 13.

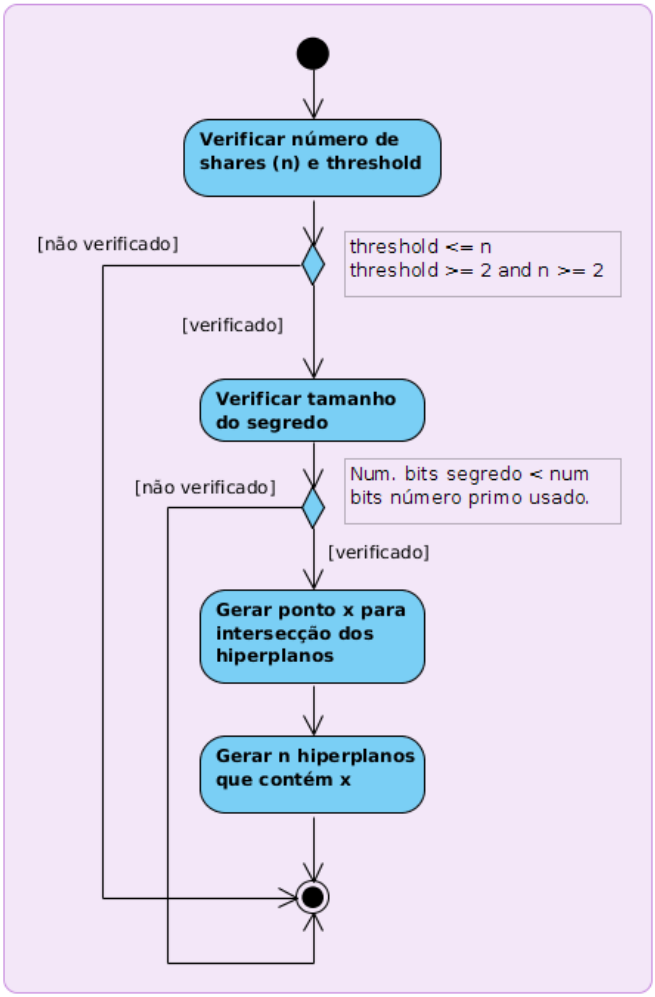

Figura 13 – Diagrama de atividades da divisão do segredo pelo Esquema de Blakley

Cada hiperplano é representado como array de números do tipo *BigInteger*. Cada hiperplano possui k+1 coordenadas, onde k é o *threshold* do esquema: são gerados k coordenadas aleatórias dentro corpo finito sendo utilizado; e a última coodernada corresponde ao valor obtido multiplicando-se, uma a uma, as coordenadas geradas do plano pelas coodernadas do ponto de interseçção, conforme o que é demonstrado na Equação 4.1. Estas coordenadas são o que constituem uma *share*, ou parte do segredo, a partir das quais o ponto de intersecção dos hiperplanos poderá ser posteriormente calculado, obtendo assim o segredo original. As *shares* são retornadas como uma lista de objetos de uma classe específica para a representação das mesmas segundo o esquema de segredo compartilhado de Blakley.

6.3.3.2 Reconstrução do segredo

A rotina que efetua a reconstrução do segredo acontece seguindo o que é mostrado, de forma simplificada, no diagrama de atividades da Figura 14. A mesma é explicada com mais detalhes a seguir.

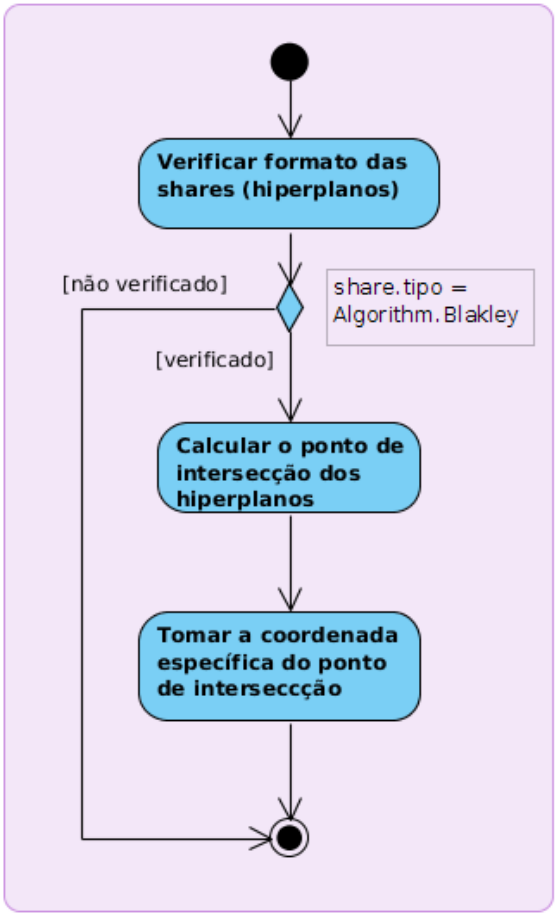

Figura 14 – Diagrama de atividades da reconstrução do segredo pelo Esquema de Blakley

Para que o método que efetua a reconstrução do segredo possa recuperá-lo de forma correta, é necessário que o mesmo receba como parâmetro k das n *shares* geradas

no momento da divisão do segredo. Da mesma forma como acontece na implementação do esquema de Shamir já descrita, o primeiro passo na rotina de reconstrução é verificar se as *shares* recebidas como parâmetro estão no formato exigido pela biblioteca, isto é, se estão representadas como objetos da classe criada especificamente para representá-las. Isso é necessário afim de que a chamada de métodos e o funcionamento correto da rotina possa ser garantida. Caso as *shares* recebidas não estejam no formato apropriado, uma exceção é lancada e a rotina é encerrada.

Uma vez que todas as *shares* são verificadas, o procedimento que efetivamente irá recuperar o segredo original pode ser realizado. Como já dito, a recuperação do segredo no caso do esquema de segredo compartilhado de Blakley é realizada através da solução de um sistema de equações lineares. A biblioteca desenvolvida como parte deste trabalho não implementa rotinas para a solução de sistemas desse tipo. Utilizamos a biblioteca *JLinAlg*, descrita na seção 5.2, para realizar esta funcionalidade. Sendo assim, vale ressaltar a dependência da biblioteca aqui proposta pela biblioteca *JlingAlg* para o caso do esquema de segredo compartilhado de Blakley.

Para a reconstrução do segredo, o procedimento é realizado conforme o explicado na subseção 4.2.3. Com as k *shares* que precisam ser recebidas como parâmetro, tem-se k equações de hiperplanos com seus correspondentes valores de a e y, mas com os valores de x desconhecidos. Cada equação será da mesma forma como apresentado em (4.1). Com os k *shares*, tem-se um sistema de equações combinando todas elas.

Solucionando o sistema de equações, são obtidos k valores de x, os quais correspondem às coordenadas do ponto secreto. O segredo pode então ser recuperado simplesmente tomando a primeira coordenada do ponto, tendo em vista que o segredo foi assim codificado no ponto de interseccção dos planos.

O diagrama de classes de todo estema esquema é apresentado na Figura 15 abaixo.

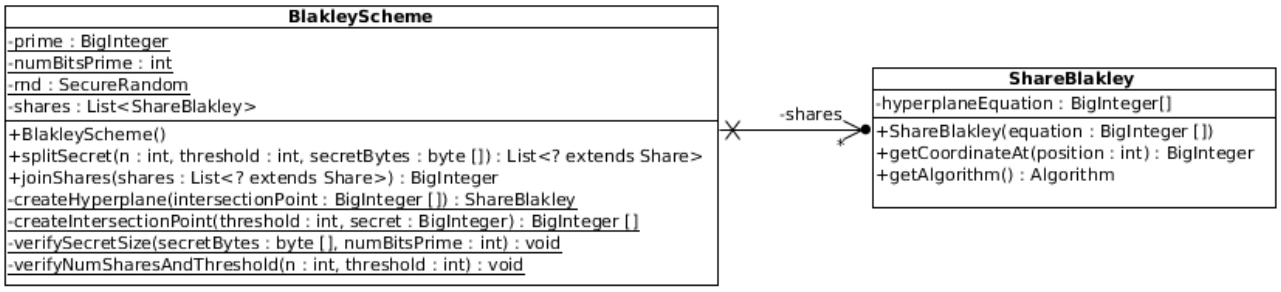

Figura 15 – Diagrama de classes da implementação do esquema de segredo compartilhado de Blakley

O diagrama de classes de projeto completo da biblioteca desenvolvida pode ser consultado no Figura 7.

6.3.3.3 Exemplo de Utilização

Assim como fizemos para o esquema de Shamir, damos a seguir um exemplo de utilização das funções fornecidas pela biblioteca implementada para a divisão e reconstrução de um dado segredo, agora utilizando o esquema de segredo compartilhado de Blakley.

```
1  public class BlakleyTest {
2
3    final static int n = 5;   // nro de usuarios participantes do esquema
4    final static int threshold = 3; // threshold do esquema
5
6    public static void main (String args[]){
7
8      List<? extends Share> shares;
9      List<ShareBlakley> sharesToReconstruct;
10     String secret = "Secret Sharing";
11
12     BlakleyScheme schemeTest = new BlakleyScheme();
13
14     try {
15       shares = schemeTest.splitSecret (n, threshold, secret.getBytes());
16
17     } catch (Exception ex) {
18       System.out.println(ex.toString());
19       return;
20     }
21     sharesToReconstruct = new ArrayList<>(threshold);
22
23     for (int i = 0; i < threshold; i++ ) {
24       ShareBlakley aux = (ShareBlakley) shares.get(i);
25       sharesToReconstruct.add(aux);
26     }
27
28     byte[] bytes = null;
29       try {
30       bytes = schemeTest.joinShares(sharesToReconstruct).toByteArray();
31
32     } catch (ShareFormatException ex) {
33       System.out.println(ex.toString());
34     }
35     String text = new String(bytes);  // segredo obtido novamente
36     }
37 }
```

Para a utilização do esquema de segredo compartilhado de Blakley, os métodos fornecidos pela biblioteca são: um método construtor de classe, o método *splitSecret()* e

joinShares(), utilizados no código mostrado nas linhas 12, 15 e 32, respectivamente.

7 Considerações Finais

Este trabalho teve como objetivo geral o desenvolvimento de uma biblioteca de software que implementasse diferentes mecanismos de segredo compartilhado de forma reconfigurável e extensível, e uma série de outros objetivos específicos que possibilitariam que o objetivo geral fosse alcançado. O objetivo geral do trabalho foi alcançado com a implementação de uma biblioteca na linguagem de programação *Java* que suporta dois mecanismos de segredo compartilhado baseados em *threshold*: o esquema de segredo compartilhado de Shamir e o esquema de segredo compartilhado de Blakley.

A partir da revisão da literatura realizada no início deste trabalho foi possível construir toda a fundamentação teórica necessária para a realização deste projeto. Nos capítulos 3 e 4, temos todos os principais conceitos relativos ao conceito de segredo compartilhado, bem como a ideia de funcionamento, características e peculiaridades dos esquemas de segredo compartilhado de Shamir e Blakley, os quais constituíram o assunto principal deste trabalho. Esta foi uma parte fundamental para que esses mecanismos fossem bem compreendidos e, posteriormente, implementados.

A biblioteca aqui proposta ainda foi projetada de forma que seja fácil e simples a adição de novos mecanismos de segredo compartilhado baseados em *threshold* junto aos já presentes. Isso é garantido pela existência de uma interface comum que define uma assinatura padrão para os métodos de divisão e reconstrução de um dado segredo, a qual deve ser implementada por todo mecanismo de segredo compartilhado baseado em *threshold* presente na biblioteca.

A biblioteca implementada como parte deste projeto fornece uma interface pública, com métodos para a divisão e reconstrução de um dado segredo em cada um dos mecanismos de segredo compartilhado implementados. Sendo assim, a mesma está pronta para ser utilizada por outras aplicações que necessitem de mecanismos de segredo compartilhado, ou mesmo em outros contextos onde mecanismos deste tipo sejam necessários.

No capítulo 5, temos a descrição de todo o processo realizado para a implementação da biblioteca de segredo compartilhado desenvolvida neste trabalho. Apresentamos a modelagem realizada utilizando diferentes diagramas UML, como diagramas de atividades para os principais métodos implementados, diagrama de pacotes que ilustra a estrutura da biblioteca, e um diagrama de classes de projeto completo da aplicação desenvolvida.

Por fim, ainda no capítulo 5 temos a descrição completa da implementação da biblioteca desenvolvida. Explicamos com detalhes a implementação de cada um dos mecanismos de segredo compartilhado escolhidos e damos exemplos de uso das funcionalidades de cada um deles. Detalhamos ainda a interface genérica desenvolvida

para mecanismos de segredo compartilhado baseados em *threshold*.

7.1 Trabalhos Futuros

Após a conclusão deste trabalho, conseguimos observar claramente a existência de dois possíveis trabalhos futuros para o presente projeto.

O primeiro deles, conforme já citamos ao longo deste trabalho, é a tradução da biblioteca de software aqui desenvolvida, na linguagem de programação *Java*, para uma outra linguagem de programação, como C ou C++, afim de que esta seja compatível com os projetos desenvolvidos pelo LabSEC que fazem uso de mecanismos de segredo compartilhado. Futuramente, espera-se que esta última possa ser a biblioteca de mecanismos de segredo compartilhado sendo utilizada nos projetos do laboratório.

O segundo possível e esperado trabalho futuro é a adição de novos mecanismos de segredo compartilhado à biblioteca de software desenvolvida. No caso de um módulo de segurança criptográfico, por exemplo, a existência de mecanismos de segredo compartilhado com capacidade de veto, ou ainda mecanismos de segredo compartilhado hierárquicos ou compartimentados, é de grande importância na medida em que abrem uma série de novas possibidades dentro deste contexto.

Referências

1. R. M. Capocelli, A. De Santis, L. Gargano, and U. Vaccaro. *On the size of shares for secret sharing schemes*. Journal of Cryptology, 6(3):157-168, 1993.

2. SUTIL, J. M. *Gestão Segura de Múltiplas Instâncias de uma Mesma Chave de Assinatura em Autoridades Certificadoras*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2011.

3. D. R. Stinson. *Cryptography Theory and Practice*, 2nd ed. CRC Press, Inc., Boca Raton, 2006.

4. Wikipedia. *Criptografia*. Disponível em: http://pt.wikipedia.org/wiki/Criptografia. Acesso em: 17.7.2013.

5. SCHNEIER, B. *Applied Cryptography*. John Wiley & Sons, 1996.

6. NIST. FIPS PUB 46 - *Data Encryption Strandard*. 1977.

7. NIST. FIPS PUB 197 - *Advanced Encryption Strandard*. 2001.

8. NIST. FIPS PUB 81 - *DES Modes of Operation*. 1980.

9. STALLINGS,W. *Cryptography and Network Security Principles and Practices*. [S.l.]: Prentice Hall, 2005.

10. NIST. FIPS PUB 180 - *Secure Hash Standard*. 1993.

11. FERGUSON, N.; SCHNEIER, B.; KOHNO, T. *Cryptography Engineering: Design Principles and Practical Applications*. [S.l.]: Wiley Publishing, 2010.

12. DIFFIE, W.; HELLMAN, M. *New directions on cryptographic techniques*. Proceedings of the AFIPS National Computer Conference, 1976.

13. HARRIS, S. *CISSP All-in-One Exam Guide*. 5. ed. [S.l.]: Mc Graw Hill, 2010.

14. RIVEST, R.; SHAMIR, A.; ADLEMAN, L. *A Method for Obtainig Digital Signatures and Public-Key Cryptosystems*. [S.l.], 1977. 15 p.

15. SOUZA, T. C. S. de. *Aspectos Técnicos e Teóricos da Gestão do Ciclo de Vida de Chaves Criptográficas no OpenHSM*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2008.

16. BLAKLEY, G. R. *Safeguarding cryptographic keys*. In: Proceedings of the National Computer Conference. [S.l.: s.n.], 1979. p. 313–317.

17. SHAMIR, A. *How to share a secret*. Communications of the ACM, v. 22, n. 11, p. 612–613, 1979.

18. Guo C, Chang CC. *A Construction for secret sharing scheme with general access structure*. Journal of Information Hiding and Multimedia Signal Processing 2013; 4(1):1–8.

19. Iftene, S.: *Secret sharing schemes with applications in security protocols*. Technical report, Alexandru Ioan Cuza University, Faculty of Computer Science (2007)

20. Wikipedia, *Finite field arithmetic*. Disponível em: http://en.wikipedia.org/wiki/Finite_field_arithmetic. Acesso em: 20.9.2013.

21. MASUDA, A. *Tópicos de Corpos Finitos com Aplicações em Criptografia e Teoria dos Códigos*. Rio de Janeiro: IMPA, 2007.

22. J. Benaloh and J. Leichter. *Generalized secret sharing and monotone functions*. In S. Goldwasser, editor, Advanced in Cryptology-CRYPTO' 88, volume 403 of Lecture Notes in Computer Science, pages 27–35. Springer-Verlag, 1989.

23. L. Csirmaz. *The size of a share must be large*. J. Cryptology 10 (1997) 223–231.

24. Pellegrini, J. C. *Introdução à Criptografia e seus fundamentos - Notas de aula*, versão 33, 2011.

25. Martin, Russ. *Introduction to Secret Sharing Schemes*.

26. K. Bozkurt and G. Selcuk, *Threshold Cryptography Based on Blakely Secret Sharing*, Information Sciences, no. x, 2008.

27. RSA Laboratories. *What are some secret sharing schemes?* [Online]. Disponível em: http://www.rsa.com/rsalabs/node.asp?id=2259. Acesso em: 19.6.2013.

28. Oracle Technology Network. *Moving Java Forward*. 2011a. Disponível em: http://www.oracle.com/br/technologies/java/index.html. Acesso em: 13.10.2013.

29. Wikipedia. *SUBVERSION*. Disponível em: http://pt.wikipedia.org/wiki/Subversion. Acesso em: 13.10.2013.

Anexos

ANEXO A – Código Fonte

Conforme apresentamos na Figura 8, a biblioteca desenvolvida está estruturada em três diferentes pacotes. A seguir, temos o código fonte desenvolvido conforme a estrutura em pacotes realizada. Iniciamos pela apresentação das classes do pacote *ThresholdScheme*.

```
1  package ThresholdScheme.ThresholdScheme;
2
3  import java.math.BigInteger;
4  import java.util.List;
5
6   public interface ThresholdScheme {
7
8       public List<? extends Share> splitSecret(int n, int threshold, byte
           [] secretBytes) throws ThresholdSchemeException;
9
10      public BigInteger joinShares (List<? extends Share> shares) throws
           ShareFormatException ;
11  }
```

```
1  package ThresholdScheme.ThresholdScheme;
2
3  public class ThresholdSchemeException extends Exception {
4
5      public ThresholdSchemeException (String msg) {
6
7          super(msg);
8      }
9  }
```

```
1  package ThresholdScheme.ThresholdScheme;
2
3  public abstract class Share {
4
5      public abstract Algorithm getAlgorithm();
6
7      public enum Algorithm {
8
9          SHAMIR, BLAKLEY
10     }
11 }
```

```java
package ThresholdScheme.ThresholdScheme;

public class ShareFormatException extends Exception {

    public ShareFormatException (String msg) {

        super(msg);

    }
}
```

A seguir, temos as classes referentes a implementação do esquema de segredo compartilhado de Shamir.

```java
package ThresholdScheme.Shamir.Main;

import ThresholdScheme.ThresholdScheme.Share;
import ThresholdScheme.ThresholdScheme.ShareFormatException;
import ThresholdScheme.ThresholdScheme.ThresholdScheme;
import ThresholdScheme.ThresholdScheme.ThresholdSchemeException;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.List;

public class ShamirScheme implements ThresholdScheme {

  private static BigInteger prime;
  private static int numBitsPrime;
  private static SecureRandom rnd;
  private BigInteger[] coefficientsPolynomial;
  private List<ShareShamir> shares;

  public ShamirScheme () {

    numBitsPrime = 128;
    rnd = new SecureRandom();
    prime = new BigInteger("243137693727224902841980180233862604941");

  }
  @Override
  public List<? extends Share> splitSecret (int n, int threshold, byte[]
      secretBytes) throws ThresholdSchemeException {

    ShamirScheme.verifyNumSharesAndThreshold(n, threshold);
    ShamirScheme.verifySecretSize(secretBytes, numBitsPrime);
```

```
32
33      this.coefficientsPolynomial = ShamirScheme.
            generateCoefficientsToPolynomial (threshold, new BigInteger(
            secretBytes));
34      this.shares = ShamirScheme.generateShares(n, coefficientsPolynomial)
            ;
35
36      return shares;
37    }
38
39    private static BigInteger[] generateCoefficientsToPolynomial (int
        threshold, BigInteger secret) {
40
41      BigInteger[] coefficients = new BigInteger[threshold];
42
43      coefficients[0] = secret;
44
45      for ( int i = 1; i < threshold; i++ ) {
46
47        coefficients[i] = new BigInteger(numBitsPrime, rnd);
48      }
49
50      return coefficients;
51
52    }
53
54    private static List<ShareShamir> generateShares (int n, BigInteger[]
        coefficients) {
55
56      ArrayList<ShareShamir> shares;
57      shares = new ArrayList<>(n);
58
59      BigInteger polyEvaluate, x_value;
60
61      for (int i = 1; i <= n; i++ ) {
62
63        do {
64          x_value = new BigInteger(numBitsPrime, rnd);
65
66        } while (!verifyIfXvalueIsUnique(x_value, shares));
67
68          shares.add(new ShareShamir());
69          shares.get(i-1).set_Xvalue(x_value);
70      }
71
72      for( int i = 1; i <= n; i++) {
73
```

```
74        polyEvaluate = evaluatePolynomial(shares.get(i-1).get_Xvalue(),
              coefficients);
75        shares.get(i-1).set_Fxvalue(polyEvaluate);
76      }
77      return shares;
78    }
79
80    private static BigInteger evaluatePolynomial(BigInteger x_value,
          BigInteger coefficients[]) {
81
82      BigInteger polyEvaluate = BigInteger.ZERO;
83
84      for(int i = 0; i < coefficients.length; i++) {
85
86        BigInteger aux = x_value.pow(i);
87        polyEvaluate = polyEvaluate.add(coefficients[i].multiply(aux));
88
89      }
90      polyEvaluate = polyEvaluate.mod(prime);
91
92      return polyEvaluate;
93    }
94
95    @Override
96    public BigInteger joinShares (List<? extends Share> shares) throws
          ShareFormatException {
97
98      List<ShareShamir> sharesReconstrucao = new ArrayList<>();
99
100     for (int i = 0; i < shares.size(); i++) {
101
102       if (shares.get(i).getAlgorithm() != Share.Algorithm.SHAMIR) {
103         throw new ShareFormatException("Erro de formato das shares.");
104       }
105       else {
106         ShareShamir aux = (ShareShamir) shares.get(i);
107         sharesReconstrucao.add(aux);
108       }
109     }
110
111     BigInteger numeratorAccum, denominatorAccum, partialResult;
112
113     BigInteger secret = BigInteger.ZERO;
114
115     BigInteger aux;
116
117     for (int i = 0; i < sharesReconstrucao.size(); i++) {
```

```
118
119         numeratorAccum = BigInteger.ONE;
120         denominatorAccum = BigInteger.ONE;
121
122         for (int j = 0; j < sharesReconstrucao.size(); j++) {
123
124           if( j != i ) {
125
126           aux = sharesReconstrucao.get(j).get_Xvalue();
127           numeratorAccum = numeratorAccum.multiply(aux);
128
129           aux = sharesReconstrucao.get(j).get_Xvalue().subtract(
                  sharesReconstrucao.get(i).get_Xvalue());
130           denominatorAccum = denominatorAccum.multiply(aux);
131
132           }
133         }
134         aux = denominatorAccum.modInverse(prime);
135         partialResult = numeratorAccum.multiply(aux).mod(prime);
136
137         aux = partialResult.multiply(sharesReconstrucao.get(i).get_Fxvalue
                ());
138         secret = secret.add(aux).mod(prime);
139       }
140       return secret;
141     }
142
143     private static boolean verifyIfXvalueIsUnique (BigInteger x, ArrayList
          <ShareShamir> shares) {
144
145       for (int i = 0; i < shares.size(); i++){
146
147         if( shares.get(i) == null) {
148           break;
149
150         } else if ( shares.get(i).get_Xvalue().compareTo(x) == 0) {
151
152           return false;
153         }
154       }
155       return true;
156     }
157
158     private List<? extends Share> getSharesShamir () {
159
160       return this.shares;
161     }
```

```
162
163   public List<ShareShamir> updateAllShares(List<ShareShamir> shares, int
          threshold) throws ShareFormatException {
164
165     for (int i = 0; i < shares.size(); i++) {
166
167       if (shares.get(i).getAlgorithm() != Share.Algorithm.SHAMIR) {
168
169         throw new ShareFormatException("Erro de formato das shares.");
170       }
171     }
172
173     byte[] bytesSecret = joinShares(shares).toByteArray();
174
175     BigInteger[] coefficients = new BigInteger[threshold];
176
177     coefficients = ShamirScheme.generateCoefficientsToPolynomial(
           threshold, new BigInteger(bytesSecret));
178
179     List<BigInteger> deltas = new ArrayList();
180     BigInteger x_value, newFxValue, deltaValue;
181
182     for (int i = 0; i < shares.size(); i++ ) {
183
184       x_value = shares.get(i).get_Xvalue();
185
186       newFxValue = ShamirScheme.evaluatePolynomial(x_value, coefficients
              );
187
188       deltaValue = newFxValue.subtract(shares.get(i).get_Fxvalue());
189
190       deltas.add(deltaValue);
191
192     }
193
194     List<ShareShamir> sharesUpdated = new ArrayList<>();
195
196     for (int i = 0; i < shares.size(); i++ ) {
197
198       ShareShamir shareI = shares.get(i);
199       BigInteger oldFxValue = shareI.get_Fxvalue();
200
201       shares.get(i).set_Fxvalue(oldFxValue.add(deltas.get(i)));
202
203       sharesUpdated.add(shares.get(i));
204     }
205     return sharesUpdated;
```

```
206   }
207
208   public List<ShareShamir> revokeShareAtPosition(List<ShareShamir>
          shares, int threshold, int position) throws ShareFormatException,
          ThresholdSchemeException {
209
210     if (position < 0 || position > shares.size()-1) {
211       throw new ThresholdSchemeException("Posicao informada eh invalida!
              ");
212
213     }
214
215     for (int i = 0; i < shares.size(); i++) {
216
217       if (shares.get(i).getAlgorithm() != Share.Algorithm.SHAMIR) {
218         throw new ShareFormatException("Erro de formato das shares.");
219       }
220     }
221
222     byte[] bytesSecret = joinShares(shares).toByteArray();
223
224     BigInteger[] coefficients = new BigInteger[threshold];
225
226     coefficients = ShamirScheme.generateCoefficientsToPolynomial(
          threshold, new BigInteger(bytesSecret));
227
228     List<BigInteger> deltas = new ArrayList();
229     BigInteger x_value, newFxValue, deltaValue;
230
231     for (int i = 0; i < shares.size(); i++ ) {
232
233       if (i == position)
234         deltas.add(new BigInteger("0"));
235
236       x_value = shares.get(i).get_Xvalue();
237
238       newFxValue = ShamirScheme.evaluatePolynomial(x_value, coefficients
              );
239
240       deltaValue = newFxValue.subtract(shares.get(i).get_Fxvalue());
241
242       deltas.add(deltaValue);
243
244     }
245
246     List<ShareShamir> sharesUpdated = new ArrayList<>();
247
```

```
248     for (int i = 0; i < shares.size(); i++ ) {
249
250       ShareShamir shareI = shares.get(i);
251       BigInteger oldFxValue = shareI.get_Fxvalue();
252
253       shares.get(i).set_Fxvalue(oldFxValue.add(deltas.get(i)));
254
255       sharesUpdated.add(shares.get(i));
256     }
257     return sharesUpdated;
258   }
259
260   private static void verifySecretSize(byte[] secretBytes, int
        numBitsPrime) throws ThresholdSchemeException {
261
262     BigInteger secret = new BigInteger(secretBytes);
263
264     if(secret.bitLength() >= numBitsPrime) {
265
266       throw new ThresholdSchemeException( "Numero de bits do segredo eh
            maior do que o numero de bits do nro. primo utilizado!");
267     }
268   }
269
270   private static void verifyNumSharesAndThreshold(int n, int threshold)
        throws ThresholdSchemeException {
271
272     if ( threshold > n) {
273
274       throw new ThresholdSchemeException("O Threshold nao pode ser maior
            que o n mero total de shares.");
275     }
276
277     else if ( n < 2 | threshold < 2 ) {
278       throw new ThresholdSchemeException("O nro. de shares e o threshold
            devem ser no minimo igual a 2.");
279
280     }
281   }
282 }
```

```
1  package ThresholdScheme.Shamir.Main;
2
3  import ThresholdScheme.ThresholdScheme.Share;
4  import java.math.BigInteger;
5
6  public class ShareShamir extends Share {
```

```
7
8     private BigInteger Xvalue;
9     private BigInteger Fxvalue;
10
11    public void set_Xvalue (BigInteger Xvalue) {
12      this.Xvalue = Xvalue;
13    }
14
15    public void set_Fxvalue(BigInteger Fxvalue) {
16      this.Fxvalue = Fxvalue;
17    }
18
19    public BigInteger get_Xvalue () {
20      return Xvalue;
21    }
22
23    public BigInteger get_Fxvalue() {
24      return Fxvalue;
25    }
26
27    @Override
28    public Algorithm getAlgorithm() {
29      return Algorithm.SHAMIR;
30    }
31
32  }
```

A seguir, temos as classes referentes a implementação do esquema de segredo compartilhado de Blakley.

```
1  package ThresholdScheme.Blakley.Main;
2
3  import ThresholdScheme.ThresholdScheme.Share;
4  import ThresholdScheme.ThresholdScheme.ShareFormatException;
5  import ThresholdScheme.ThresholdScheme.ThresholdScheme;
6  import ThresholdScheme.ThresholdScheme.ThresholdSchemeException;
7  import java.math.BigInteger;
8  import java.security.SecureRandom;
9  import java.util.ArrayList;
10 import java.util.List;
11 import org.jlinalg.LinSysSolver;
12 import org.jlinalg.Matrix;
13 import org.jlinalg.FieldElement;
14 import org.jlinalg.rational.Rational;
15 import org.jlinalg.Vector;
16
```

```
17 public class BlakleyScheme implements ThresholdScheme {
18
19   private static BigInteger prime;
20   private static int numBitsPrime;
21   private static SecureRandom rnd;
22   private List<ShareBlakley> shares;
23
24   public BlakleyScheme ()  {
25
26     BlakleyScheme.numBitsPrime = 128;
27     rnd = new SecureRandom();
28     prime = new BigInteger("2431376937272249028419801802338626049 41");
29   }
30
31   private static ShareBlakley createHyperplane(BigInteger
        intersectionPoint[]){
32
33     BigInteger hyperplaneEquation[] = new BigInteger[intersectionPoint.
          length+1];
34     BigInteger sum = new BigInteger("0");
35
36     for (int i = 0; i < intersectionPoint.length; i++){
37
38       hyperplaneEquation[i] = new BigInteger(numBitsPrime, rnd);
39       sum = sum.add(hyperplaneEquation[i].multiply(intersectionPoint[i])
             );
40     }
41
42     hyperplaneEquation[intersectionPoint.length] = sum;
43     return new ShareBlakley(hyperplaneEquation);
44   }
45
46   @Override
47   public List<? extends Share> splitSecret (int n, int threshold, byte[]
        secretBytes) throws ThresholdSchemeException {
48
49     BlakleyScheme.verifyNumSharesAndThreshold(n, threshold);
50     BlakleyScheme.verifySecretSize(secretBytes, numBitsPrime);
51
52     BigInteger secret = new BigInteger(secretBytes);
53
54     BigInteger[] intersectionPoint = BlakleyScheme.
          createIntersectionPoint(threshold, secret);
55     shares = new ArrayList<>(n);
56
57     //Generate n keys
58     for (int i = 0; i < n; i++) {
```

```
59        shares.add( BlakleyScheme.createHyperplane(intersectionPoint) );
60      }
61      return shares;
62    }
63
64    @Override
65    public BigInteger joinShares (List<? extends Share> shares) throws
          ShareFormatException {
66
67      List<ShareBlakley> hyperplanesEquations = new ArrayList<>();
68
69      for (int i = 0; i < shares.size(); i++) {
70
71        if (shares.get(i).getAlgorithm() != Share.Algorithm.BLAKLEY) {
72          throw new ShareFormatException("Erro de formato das shares.");
73        }
74        else {
75          ShareBlakley aux = (ShareBlakley) shares.get(i);
76          hyperplanesEquations.add(aux);
77        }
78      }
79
80      FieldElement[][] fields = new FieldElement[hyperplanesEquations.size
            ()][hyperplanesEquations.get(0).getHyperplaneEquation().length
            -1];
81
82      Vector <Rational> vector = new Vector <Rational> (
            hyperplanesEquations.size(), Rational.FACTORY);
83
84      for(int i = 0; i < hyperplanesEquations.size(); i++){
85
86        for(int j = 0; j < hyperplanesEquations.get(0).
              getHyperplaneEquation().length-1; j++){
87
88          fields[i][j] = Rational.FACTORY.get(hyperplanesEquations.get(i).
                getCoordinateAt(j));
89        }
90
91        vector.set(i+1, Rational.FACTORY.get(hyperplanesEquations.get(i).
              getCoordinateAt(hyperplanesEquations.get(0).
              getHyperplaneEquation().length-1)));
92      }
93
94      Matrix matrix = new Matrix(fields);
95
96      Vector solve = new LinSysSolver().solve(matrix, vector);
97
```

```
98      BigInteger solution = new BigInteger(solve.getEntry(1).toString());
99
100     return solution;
101   }
102
103   private static BigInteger[] createIntersectionPoint (int threshold,
          BigInteger secret){
104
105     BigInteger[] intersectionPoint = new BigInteger[threshold];
106
107     byte[] scrt = secret.toByteArray();
108
109     intersectionPoint[0] = new BigInteger(scrt);
110
111     for (int i = 1; i < threshold; i++) {
112       intersectionPoint[i] = new BigInteger(numBitsPrime, rnd);
113     }
114
115     for (int i = 0; i < threshold; i++) {
116       System.out.println(intersectionPoint[i]);
117     }
118     return intersectionPoint;
119
120   }
121
122   private static void verifySecretSize(byte[] secretBytes, int
          numBitsPrime) throws ThresholdSchemeException {
123
124     BigInteger secret = new BigInteger(secretBytes);
125
126     if(secret.bitLength() >= numBitsPrime) {
127
128       throw new ThresholdSchemeException( "Numero de bits do segredo eh
              maior do que o numero de bits do nro. primo utilizado!");
129     }
130   }
131
132   private static void verifyNumSharesAndThreshold(int n, int threshold)
          throws ThresholdSchemeException {
133
134     if ( threshold > n) {
135       throw new ThresholdSchemeException("O Threshold nao pode ser maior
              que o numero total de shares.");
136     }
137     else if ( n < 2 | threshold < 2 ) {
138       throw new ThresholdSchemeException("O nro. de shares e o threshold
              devem ser no minimo igual a 2.");
```

```
139     }
140   }
141 }
```

```
1  package ThresholdScheme.Blakley.Main;
2
3  import ThresholdScheme.ThresholdScheme.Share;
4  import java.math.BigInteger;
5
6  public class ShareBlakley extends Share {
7
8    private BigInteger[] hyperplaneEquation;
9
10   public ShareBlakley (BigInteger[] equation) {
11     this.hyperplaneEquation = equation;
12   }
13
14   public BigInteger[] getHyperplaneEquation() {
15     return this.hyperplaneEquation;
16   }
17
18   public BigInteger getCoordinateAt(int position) {
19     return hyperplaneEquation[position];
20   }
21
22   @Override
23   public Algorithm getAlgorithm() {
24     return Algorithm.BLAKLEY;
25   }
26
27 }
```

ANEXO B – Documentação com *Javadoc*

A seguir, segue a documentação *Javadoc* das classes referentes ao pacote *Threshold_scheme*, ilustrado na Figura 8.

ThresholdScheme.ThresholdScheme

Interface ThresholdScheme

All Known Implementing Classes:

BlakleyScheme, ShamirScheme

```
public interface ThresholdScheme
```

Interface Java genérica para mecanismos de segredo compartilhado baseados em um threshold. Esta classe define um método para divisão do segredo e um método para reconstrução do segredo. Essa interface é implementada pelos dois mecanismos de segredo compartilhado que fazem parte da biblioteca e deverá ser implementada por qualquer outra implementação de mecanismos de segredo compartilhado desse tipo que venha ser adicionada a biblioteca.

Method Summary

Methods

Modifier and Type	Method and Description
`java.math.BigInteger`	`joinShares(java.util.List<? extends Share> shares)` Efetua a reconstrução do segredo.
`java.util.List<? extends Share>`	`splitSecret(int n, int threshold, byte[] secretBytes)` Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontruí-lo.

Method Detail

splitSecret

```
java.util.List<? extends Share> splitSecret(int n,
                                             int threshold,
                                             byte[] secretBytes)
                                               throws ThresholdSchemeException
```

Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontruí-lo.

Parameters:

n - Número de partes em que o segredo será dividido.

`threshold` - Número de partes necessárias para recuperar o segredo.

`secretBytes` - O segredo a ser dividido.

Returns:

lista com as n shares (partes do segredo). Cada share é um objeto de qualquer classe que extenda a classe `Share`.

Throws:

`ThresholdSchemeException` - Será lançada uma exceção nos seguinte casos: o número de shares e o threshold não sejam, no mínimo, igual a 2; o threshold seja maior do que o número total de shares; ou se o número de bits do segredo for maior do que o número de bits do número primo utilizado.

joinShares

```
java.math.BigInteger joinShares(java.util.List<? extends Share> shares)
                                 throws ShareFormatException
```

Efetua a reconstrução do segredo. Deve receber como parâmetro um número de shares igual ao threshold do esquema. As shares são objetos de qualquer classe que extenda a classe `Share`, podendo ser das classes `ShareShamir` ou `ShareBlakley`.

Parameters:

`shares` - shares para reconstrução do segredo.

Returns:

o segredo reconstruído, representado como um `BigInteger`.

Throws:

`ShareFormatException`

See Also:

`Share`, `BigInteger`

ThresholdScheme.ThresholdScheme

Class ThresholdSchemeException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      ThresholdScheme.ThresholdScheme.ThresholdSchemeException

All Implemented Interfaces:

java.io.Serializable

```
public class ThresholdSchemeException
extends java.lang.Exception
```

Classe criada para o tratamento de exceções, que nesse caso são situações que precisam ser considerados casos de erros em mecanismos de segredo compartilhado baseados em threshold. Será lançada uma exceção nos seguinte casos: o número de shares e o threshold não sejam, no mínimo, igual a 2; o threshold seja maior do que o número total de shares; ou se o número de bits do segredo for maior do que o número de bits do número primo utilizado.

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
`ThresholdSchemeException(java.lang.String msg)` Construtor uma exceção do tipo ThresholdSchemeException.

Constructor Detail

ThresholdSchemeException

```
public ThresholdSchemeException(java.lang.String msg)
```

Construtor uma exceção do tipo ThresholdSchemeException.

Parameters:

`msg` - Descrição da exceção.

ThresholdScheme.ThresholdScheme

Class Share

java.lang.Object
    ThresholdScheme.ThresholdScheme.Share

Direct Known Subclasses:

ShareBlakley, ShareShamir

```
public abstract class Share
extends java.lang.Object
```

Classe abstrata para representação de uma share. Em cada um dos mecanismos de segredo compartilhados implementados, existem classes apropriadas para a representação das shares naquele mecanismo, sendo que estas extendem a esta classe Share.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**Share.Algorithm** Método que retorna qual o tipo de uma determinada share, ou seja, se esta é uma share do mecanismo de Shamir ou Blakley.

Method Detail

getAlgorithm

```
public abstract Share.Algorithm getAlgorithm()
```

ThresholdScheme.ThresholdScheme

Class ShareFormatException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      ThresholdScheme.ThresholdScheme.ShareFormatException

All Implemented Interfaces:

java.io.Serializable

```
public class ShareFormatException
extends java.lang.Exception
```

Classe criada para o tratamento de exceções referentes a inadequações no formato de uma determinada share cujo formato não corresponde ao esperado para o correto funcionamento da aplicação.

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
`ShareFormatException(java.lang.String msg)` Construtor uma exceção do tipo ShareFormatException.

Constructor Detail

ShareFormatException

```
public ShareFormatException(java.lang.String msg)
```

Construtor uma exceção do tipo ShareFormatException.

Parameters:

`msg` - Descrição da exceção.

A seguir, segue a documentação *Javadoc* das classes referentes à implementação do esquema de segredo compartilhado de Shamir.

ThresholdScheme.Shamir.Main

Class ShamirScheme

java.lang.Object
    ThresholdScheme.Shamir.Main.ShamirScheme

All Implemented Interfaces:

ThresholdScheme

```
public class ShamirScheme
extends java.lang.Object
implements ThresholdScheme
```

Classe que implementa todas as operações necessárias para o esquema de segredo compartilhado de Shamir, o qual é baseado na interpolação de Lagrange. Além das operações de usuais de divisão e reconstrução do segredo, há dois métodos adicionais: no primeiro deles, um conjunto de shares pode ser atualizado/modificado sem que o segredo original seja alterado; no segundo, pode-se revogar uma share existente.

Você precisa de k de n shares para resolver o segredo.

Constructor Summary

Constructors

Constructor and Description
ShamirScheme() Construtor para o esquema de Shamir.

Method Summary

Methods

Modifier and Type	Method and Description
java.math.BigInteger	**joinShares**(java.util.List<? extends **Share**> shares) Efetua a reconstrução do segredo utilizando a Interpolação de Lagrange.
java.util.List<**ShareShamir**>	**revokeShareAtPosition**(java.util.List<**ShareShamir**> shares, int threshold, int position) Método que atualiza um conjunto de shares recebido como parâmetro mantendo o segredo original, e revoga a share cuja posição na lista de shares foi passado como parâmetro.
java.util.List<? extends **Share**>	**splitSecret**(int n, int threshold, byte[] secretBytes) Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontrui-lo.
java.util.List<**ShareShamir**>	**updateAllShares**(java.util.List<**ShareShamir**> shares, int threshold) Método que atualiza um conjunto de shares recebido como parâmetro mantendo o segredo original.

Constructor Detail

ShamirScheme

```
public ShamirScheme()
```

Construtor para o esquema de Shamir.
São definidos: um número primo, característica do corpo primo; o número de bits do primo; e um gerador seguro de números aleatórios.

Method Detail

splitSecret

```
public java.util.List<? extends Share> splitSecret(int n,
                                                    int threshold,
                                                    byte[] secretBytes)
                                                      throws ThresholdSchemeException
```

Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontruí-lo.

Specified by:

`splitSecret` in interface `ThresholdScheme`

Parameters:

n - Número de partes em que o segredo será divido.

`threshold` - Número de partes necessárias para recuperar o segredo.

`secretBytes` - O segredo a ser dividido.

Returns:

array com as n shares (partes do segredo). Cada share é um objeto da classe `ShareShamir` que extende a classe `Share`.

Throws:

`ThresholdSchemeException` - Será lançada uma exceção nos seguinte casos: o número de shares e o threshold não sejam, no mínimo, igual a 2; o treshold seja maior do que o número total de shares; ou se o número de bits do segredo for maior do que o número de bits do número primo utilizado.

joinShares

```
public java.math.BigInteger joinShares(java.util.List<? extends Share> shares)
                                throws ShareFormatException
```

Efetua a reconstrução do segredo utilizando a Interpolação de Lagrange. Deve receber como parâmetro um número de shares igual ao threshold do esquema. As shares são objetos da classe `ShareShamir` que extende a classe `Share`.

Specified by:

`joinShares` in interface `ThresholdScheme`

Parameters:

`shares` - shares para reconstrução do segredo. São objetos da classe `ShareShamir`.

Returns:

o segredo reconstruído, representado como um `BigInteger`.

Throws:

`ShareFormatException` - Caso alguma das shares não esteja no formato que é esperado pela aplicação. Espera-se que todas sejam da classe `ShareShamir`.

See Also:

`Share`, `ShareShamir`, `BigInteger`

updateAllShares

```
public java.util.List<ShareShamir> updateAllShares(java.util.List<ShareShamir> shares,
                                        int threshold)
                                          throws ShareFormatException
```

Método que atualiza um conjunto de shares recebido como parâmetro mantendo o segredo original.

Parameters:

`shares` - lista com shares da classe `ShareShamir`.

`threshold` - Número de partes necessárias para recuperar o segredo.

Returns:

lista com as shares (partes do segredo) atualizadas.

Throws:

`ShareFormatException` - Caso alguma das shares não esteja no formato que é esperado pela aplicação. Espera-se que todas sejam da classe `ShareShamir`.

revokeShareAtPosition

```
public java.util.List<ShareShamir> revokeShareAtPosition(java.util.List<ShareShamir> shares,
                                                         int threshold,
                                                         int position)
                                                           throws ShareFormatException,
                                                                  ThresholdSchemeException
```

Método que atualiza um conjunto de shares recebido como parâmetro mantendo o segredo original, e revoga a share cuja posição na lista de shares foi passado como parâmetro.

Parameters:

`shares` - lista com shares da classe `ShareShamir`.

`threshold` - Número de partes necessárias para recuperar o segredo.

`position` - Posição da share que deseja-se revogar na lista de shares.

Returns:

lista com as shares (partes do segredo) atualizadas e a share em questão revogada.

Throws:

`ShareFormatException` - Caso alguma das shares não esteja no formato que é esperado pela aplicação. Espera-se que todas sejam da classe `ShareShamir`.

`ThresholdSchemeException`

ThresholdScheme.Shamir.Main

Class ShareShamir

java.lang.Object
- ThresholdScheme.ThresholdScheme.Share
  - ThresholdScheme.Shamir.Main.ShareShamir

```
public class ShareShamir
extends Share
```

Classe que representa um share no esquema de Shamir.

No esquema de Shamir, a share é um ponto sobre o polinômio utilizado, do tipo (x, f(x)).

Nested Class Summary

Nested classes/interfaces inherited from class ThresholdScheme.ThresholdScheme.Share

`Share.Algorithm`

Constructor Summary

Constructors

Constructor and Description
`ShareShamir()`

Method Summary

Methods

Modifier and Type	Method and Description
java.math.BigInteger	**get_Fxvalue()** Retorna o valor de f(x) da share, ou seja, o valor do polinômio para o valor de x (primeira coordenada da share).
java.math.BigInteger	**get_Xvalue()** Retorna o valor de x da share, ou seja, o valor para o qual o polinômio é avaliado.
Share.Algorithm	**getAlgorithm()**
void	**set_Fxvalue**(java.math.BigInteger Fxvalue) Altera o valor de f(x) da share, ou seja, o valor do polinômio para o valor de x (primeira coordenada da share).
void	**set_Xvalue**(java.math.BigInteger Xvalue) Altera o valor de x da share, ou seja, o valor em que o polinômio é avaliado.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ShareShamir

```
public ShareShamir()
```

Method Detail

set_Xvalue

```
public void set_Xvalue(java.math.BigInteger Xvalue)
```

Altera o valor de x da share, ou seja, o valor em que o polinômio é avaliado.

Parameters:

Xvalue - valor de x a ser setado.

set_Fxvalue

```
public void set_Fxvalue(java.math.BigInteger Fxvalue)
```

Altera o valor de f(x) da share, ou seja, o valor do polinômio para o valor de x (primeira coordenada da share).

Parameters:

`Fxvalue` - valor de f(x) a ser setado.

get_Xvalue

```
public java.math.BigInteger get_Xvalue()
```

Retorna o valor de x da share, ou seja, o valor para o qual o polinômio é avaliado.

Returns:

a primeira coordenada da share, que é o valor de x para o qual o polinômio é avaliado.

get_Fxvalue

```
public java.math.BigInteger get_Fxvalue()
```

Retorna o valor de f(x) da share, ou seja, o valor do polinômio para o valor de x (primeira coordenada da share).

Returns:

f(x) da share, ou seja, o valor do polinômio para o valor de x que corresponde à primeira coordenada da share.

getAlgorithm

```
public Share.Algorithm getAlgorithm()
```

Specified by:

`getAlgorithm` in class `Share`

A seguir, segue a documentação *Javadoc* das classes referentes à implementação do esquema de segredo compartilhado de Blakley.

ThresholdScheme.Blakley.Main

Class BlakleyScheme

java.lang.Object
    ThresholdScheme.Blakley.Main.BlakleyScheme

All Implemented Interfaces:

ThresholdScheme

```
public class BlakleyScheme
extends java.lang.Object
implements ThresholdScheme
```

Classe que implementa todas as operações necessárias para o esquema de segredo compartilhado de Blakley, o qual é baseado na intersecção de hiplerplanos.

Há três métodos públicos que podem ser chamados externamente à classe: o construtor; um efetuando a divisão do segredo em n shares, de tal modo que k delas sejam suficientes para reconstruir o segredo; e outro que efetua a reconstrução do segredo baseado nas k shares recebidas.

Você precisa de k de n shares para resolver o segredo.

Esta classe utiliza a biblioteca JLinAlg, disponível em http://jlinalg.sourceforge.net. A mesma já se encontra empacotado junto ao .jar desta aplicação.

Constructor Summary

Constructors

Constructor and Description
`BlakleyScheme()` Construtor para o esquema de Blakley.

Method Summary

Methods

Modifier and Type	Method and Description
`java.math.BigInteger`	**`joinShares`**`(java.util.List<? extends Share> shares)` Efetua a reconstrução do segredo utilizando a resolução de um sistema linear.
`java.util.List<? extends Share>`	**`splitSecret`**`(int n, int threshold, byte[] secretBytes)` Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontruí-lo.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

BlakleyScheme

```
public BlakleyScheme()
```

Construtor para o esquema de Blakley.
São definidos: um número primo, característica do corpo primo; o número de bits do primo; e um gerador seguro de números aleatórios.

Method Detail

splitSecret

```
public java.util.List<? extends Share> splitSecret(int n,
                                                    int threshold,
                                                    byte[] secretBytes)
                                                      throws ThresholdSchemeException
```

Efetua a divisão do segredo em n partes, de tal forma que *threshold* partes serão suficientes para recontruí-lo.

Specified by:

`splitSecret` in interface `ThresholdScheme`

Parameters:

n - Número de partes em que o segredo será dividido.

`threshold` - Número de partes necessárias para recuperar o segredo.

`secretBytes` - O segredo a ser dividido.

Returns:

array com as n shares (partes do segredo). Cada share é um objeto da classe `ShareBlakley` que extende a classe `Share`.

Throws:

`ThresholdSchemeException` - Será lançada uma exceção nos seguinte casos: o número de shares e o threshold não sejam, no mínimo, igual a 2; o threshold seja maior do que o número total de shares; ou se o número de bits do segredo for maior do que o número de bits do número primo utilizado.

joinShares

```
public java.math.BigInteger joinShares(java.util.List<? extends Share> shares)
                                 throws ShareFormatException
```

Efetua a reconstrução do segredo utilizando a resolução de um sistema linear. Deve receber como parâmetro um número de shares igual ao threshold do esquema. As shares são objetos da classe `ShareBlakley` que extende a classe `Share`. Cada share representa a equação do hiperplano do participante.

Specified by:

`joinShares` in interface `ThresholdScheme`

Parameters:

`shares` - shares para reconstrução do segredo. São objetos da classe `ShareBlakley`.

Returns:

o segredo reconstruído, representado como um `BigInteger`.

Throws:

`ShareFormatException`

ThresholdScheme.Blakley.Main

Class ShareBlakley

java.lang.Object
  ThresholdScheme.ThresholdScheme.Share
    ThresholdScheme.Blakley.Main.ShareBlakley

```
public class ShareBlakley
extends Share
```

Classe criada para a representação de shares em formato adequado para o esquema de segredo compartilhado de Blakley. O único atributo desta classe é um array de números da classe `BigInteger`, que são todas os valores da equação de um dado hiperplano.

Nested Class Summary

Nested classes/interfaces inherited from class ThresholdScheme.ThresholdScheme.Share

`Share.Algorithm`

Constructor Summary

Constructors

Constructor and Description
`ShareBlakley``(java.math.BigInteger[] equation)` Construtor de uma share para o esquema de Blakley.

Method Summary

Methods

Modifier and Type	Method and Description
`Share.Algorithm`	**`getAlgorithm`**`()` Retorna o tipo da share, nesse caso uma share do tipo Algorithm.BLAKLEY.
`java.math.BigInteger`	**`getCoordinateAt`**`(int position)` Retorna uma coordenada específica da equação do hiperplano.
`java.math.BigInteger[]`	**`getHyperplaneEquation`**`()` Retorna a equação do hiperplano.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ShareBlakley

```
public ShareBlakley(java.math.BigInteger[] equation)
```

Construtor de uma share para o esquema de Blakley.

Parameters:

`equation` - array de números da classe `BigInteger`, contendo todos os valores da equação de um dado hiperplano, que é uma share segundo este esquema de segredo compartilhado.

Method Detail

getHyperplaneEquation

```
public java.math.BigInteger[] getHyperplaneEquation()
```

Retorna a equação do hiperplano.

Returns:

array contendo todas os valores da equação do dado hiperplano.

getCoordinateAt

```
public java.math.BigInteger getCoordinateAt(int position)
```

Retorna uma coordenada específica da equação do hiperplano.

Parameters:

`position` - posição da coordenada do plano que deseja-se obter.

Returns:

a coordenada especificada como um número do tipo `BigInteger`

getAlgorithm

```
public Share.Algorithm getAlgorithm()
```

Retorna o tipo da share, nesse caso uma share do tipo Algorithm.BLAKLEY.

Specified by:

`getAlgorithm` in class `Share`

Returns:

o tipo da share.