MODELO ESTRUTURAL DE UM SISTEMA CRIPTOGRÁFICO EM JAVA
IMPLEMENTAÇÕES DOS ALGORITMOS DES e AES (Rijndael)

Leandro Batista de Almeida, Walter Godoy Jr.

CEFET-PR
Av. Sete de Setembro. 3165
80230-901 – Curitiba – PR – BRAZIL

RESUMO

O presente trabalho apresenta um modelo estrutural de um sistema criptográfico em Java que permite que quaisquer algoritmos de criptografia sejam codificados e utilizados de maneira simples e padronizada. Através do modelo proposto, é possível inserir criptografia de maneira transparente em canais de comunicação como sockets TCP/IP convencionais, acesso a arquivos, entre outros. Seguindo o modelo, foram implementados os algoritmos de criptografia simétrica DES e AES (Rijndael). Testes de desempenho e de segurança foram executados com seus resultados analisados. A linguagem utilizada foi Java, devido a sua interoperabilidade e portabilidade.

Palavras-chave: criptografia, DES, AES, Java.

ABSTRACT

The present work presents a structural model of a cryptographic system in Java, which allows that any encryption algorithm to be codified and used in a simple and standard way. Through this proposal structural model, it is possible to insert cryptography by a transparent way in communication channels like conventional TCP/IP sockets and access to files. According to the model, was implemented the DES and AES (Rijndael) alghorithms.. Performance and security tests were executed, with the results analized. The language used was Java, because of its interoperability and portability.

Keywords: cryptography, DES, AES, Java.

1. INTRODUÇÃO

O presente trabalho é uma estrutura de software construída utilizando-se a tecnologia Java, composto de uma coleção de classes, que permite a utilização de algoritmos de criptografia sobre *streams* de dados de maneira transparente, além de criptografia de blocos de texto convencionais. Os *streams* de dados podem ser acoplados a outros *streams* da linguagem Java, permitindo que *sockets* de comunicação TCP/IP, arquivos e outros *streams* possam utilizar criptografia transparentemente e sem maiores alterações no código.

A estrutura proposta serve de base para aplicações que necessitem de um maior grau de segurança, exigindo criptografia. A definição de um modelo (*framework*) reduz a complexidade inerente das tarefas de criptografia, fornecendo um método padronizado para a construção de programas, através de um modelo.

O modelo apresenta duas importantes características. A primeira é que os algoritmos de criptografia utilizados podem ser facilmente substituídos, ou atualizados, bastando para isso reescrever os métodos específicos do algoritmo, utilizando os conceitos de herança e de polimorfismo da tecnologia de orientação à objetos, presentes na plataforma Java. A segunda característica importante é que o uso da criptografia dento de programas, através do modelo, passa a ser simples, não apresentando dificuldades, mesmo para aqueles desenvolvedores que ignoram o funcionamento dos processos de criptografia. As funcionalidades de encriptação podem ser obtidas apenas utilizando as classes específicas do modelo em substituição ou adição às classes normalmente utilizadas nos processos envolvendo *streams* de dados, como processos de comunicação e outros.

Descrição do Problema

Existem dois principais problemas, no que diz respeito à implementação de criptografia em programas de computador. O primeiro está relacionado à complexidade natural dos algoritmos e processos de criptografia a serem implementados, que, em geral, exige um maior conhecimento da linguagem de programação utilizada e meticulosidade. Em sistemas de informação, não é comum se desejar somente encriptar um simples bloco de dados de forma isolada. Em geral, é necessário que a criptografia aconteça dentro de processos de comunicação, em gravações de arquivos, em processos de autenticação de usuários e serviços e outros. Os algoritmos e procedimentos de criptografia devem se inserir dentro do processamento dos dados da maneira mais transparente possível, não adaptando os processos a si, mas se adaptando ao que é necessário para que os sistemas possam cumprir seus objetivos.

O segundo problema diz respeito principalmente a soluções de criptografia externas. Atualmente, é possível implementar recursos de criptografia em programas de computador com a inclusão de tecnologias externas, como SSL e bibliotecas que contenham rotinas adequadas. Embora o nível de segurança dessas soluções seja geralmente bastante alto, e satisfatório para aplicações comerciais, alguns problemas podem aparecer.

Em relação à tecnologia SSL (*Secure Socket Layer*) [7], os principais problemas se relacionam a necessidade de um agente externo para regular a manutenção da segurança do sistema. Apesar da arquitetura do SSL ser extremamente benéfica e útil em muitas situações, em determinados casos, é necessário somente que

um canal encriptado seja estabelecido, sem a necessidade de complexidade extra. Outra questão é a manutenção dos certificados de segurança, que podem ser onerosos e complexos para algumas aplicações. O protocolo SET (*Secure Electronic Transaction*) [8], possui características semelhantes ao SSL no que diz respeito à autoridades de certificação externas, com os mesmos problemas de custos e de complexidade de implementação. Muitas soluções não precisam de tais recursos de segurança, ou não podem custear a complexidade ou mesmo o custo financeiro envolvido.

Outra questão em relação as soluções externas se refere ao nível de criptografia permitido para determinadas aplicações. Apesar de que o Acordo de Wassenaarr [9] já não ser mais um dos fatores de impedimento, e das regras de exportação de tecnologia de criptografia dos EUA já estarem sendo afrouxadas, ainda nos confrontamos com alguns limites. As empresas ainda vão demorar algum tempo para portar suas aplicações para essa nova realidade.

Juntos, esses problemas aumentam o custo das soluções que envolvem o uso da criptografia. Por um lado temos a dificuldade técnica, o que envolve um custo maior de desenvolvimento, e por outro temos o próprio custo financeiro de soluções e a maior complexidade de soluções envolvendo terceiros.

2. OBJETIVOS DO MODELO

Dadas as questões acima citadas, e considerando o atual estágio de desenvolvimento de sistemas de comunicação em rede, temos como principais objetivos do modelo:

a) Acesso de forma transparente e simples

Para permitir o uso do *framework* em aplicações distribuídas, o uso das classes e métodos de criptografia deve ser o mais transparente possível do ponto de vista do programador. O programador deve ser capaz de utilizar um canal *byte stream* – como um *socket* TCP/IP, um acesso a arquivos ou a uma porta de comunicação – da mesma forma que usaria normalmente ou com poucas modificações.

b) Criptografia forte

Alguns sistemas comerciais fornecem encriptação com chave simétrica com tamanho de chave até 40 bits. No atual estágio da tecnologia de criptoanálise, isto não é suficiente para muitas aplicações críticas [1][6][10]. Para assegurar o sigilo dos dados, os algoritmos de criptografia simétrica devem implementar tamanhos de chave de pelo menos 56 bits, ainda assim com certo risco de ataques bem sucedidos. Se possível, a implementação de criptografia simétrica com tamanhos de chave de 128 bits ou superior é desejável.

c) Capacidade de expansão

O modelo deve poder ser expandido, com a implementação de novos algoritmos, ou a atualização dos algoritmos já implementados. Utilizando os recursos de orientação à objetos, a atualização dos algoritmos deve ser transparente para os sistemas já desenvolvidos, seguindo a característica de encapsulamento. Através da herança, a adição de novos algoritmos deve ser simples, e sua utilização deve ser semelhante a dos outros algoritmos do modelo, seguindo o conceito de polimorfismo.

d) Baixo custo de implementação

A solução deve ter um baixo custo de implementação, significando que os recursos computacionais exigidos para seu funcionamento não irão exigir sistemas especializados e de alto custo. Os algoritmos implementados também devem ser de uso livre, sem o pagamento de direitos autorais. Não deve existir nenhum impedimento em relação à utilização do modelo em outros sistemas ou como suporte a soluções de terceiros.

e) Portabilidade e capacidade multiplataforma

O modelo deve permitir seu funcionamento em diversos ambientes operacionais, sem diferenças no funcionamento.

3. PLATAFORMA JAVA COMO FERRAMENTA

Na implementação do presente *framework*, foram consideradas diversas linguagens e plataformas, sendo que a plataforma Java foi escolhida por melhor se adaptar às necessidades do modelo e as funcionalidades desejadas.

Um dos pontos notáveis da tecnologia Java é a sua capacidade de ser executada em qualquer plataforma que implemente uma *Virtual Machine* sem nenhuma modificação – nem sequer recompilação – do código original, o que a torna também interessante para a implementação do presente modelo[2][3].

Um outro fator auxiliador na escolha é a capacidade de integração da Java. Ela consegue interagir com objetos distribuídos, como Corba ou DCom e outros sistemas distribuídos através das bibliotecas de classes. Se integra com a grande maioria dos sistemas de bancos de dados através da interface JDBC. Pode até mesmo se integrar com outras linguagens de programação ou o próprio sistema operacional via chamadas nativas, através do JNI (*Java Native Interface*). Isso pode ser utilizado para integrar o modelo com aplicações já existentes desenvolvidas em outras linguagens[4][5].

Com todos esses fatores, a plataforma Java foi uma escolha que permitiu que o modelo se integrasse a grande maioria das arquiteturas e sistemas operacionais existentes, além de permitir uma grande integração com sistemas distribuídos e corporativos.

4. O MODELO

O presente modelo é uma coleção de classes Java que permitem que recursos de criptografia possam ser implementados em programas escondendo a complexidade da implementação e permitindo uma utilização simples. O modelo foi desenvolvido através da forte utilização dos recursos de herança, polimorfismo e encapsulamento, da tecnologia de orientação à objetos. Através desses recursos, o modelo pode ser estendido e utilizado de maneira simples e padronizada.

O modelo é composto por uma classe abstrata que determina o comportamento e características dos algoritmos, classes que implementam um fluxo de dados (*stream*) que herdam de `java.io.FilterOutputStream` e `java.io.FilterInputStream` e classes auxiliares, além das classes que implementam de fato os algoritmos. As descrições das classes e seu relacionamento seguem abaixo.

Classe Cripto

Esta é a classe principal do modelo, e determina, através de herança, o comportamento de todas as classes que implementam de fato os algoritmos de criptografia. A classe `Cripto` é definida como abstrata para impedir que a criação de objetos diretamente a partir dela. A idéia é que se instanciem objetos de classes que herdam de `Cripto`. A chave do algoritmo deve ser definida preferencialmente como uma *array* de bytes, mesmo que alguns algoritmos tenham chaves que possam ser definidas dentro de tipos primitivos de Java.

A classe `Cripto` contém todos os métodos que realizam a encriptação e decriptação, além de métodos auxiliares para tratar o tamanho do bloco e o tamanho da chave do algoritmo. Os métodos de `Cripto` apresentam uma interface padronizada de acesso aos algoritmos de criptografia. Seja qual for o algoritmo, a chamada do método é a mesma, e o acesso à criptografia também.

Classes CriptoInputStream e CriptoOutputStream

A classe `Cripto` mantém uma interface padronizada para os algoritmos de criptografia de bloco, e para utilizar os recursos de criptografia, seus métodos devem ser invocados. Porém, para embutir os recursos de criptografia em um fluxo de dados padronizado de Java, mesmo a classe `Cripto` deixará os detalhes de criptografia evidentes.

A característica dos algoritmos a serem implementados através de `Cripto` é de tratarem blocos de bits e realizarem a criptografia baseada nesses blocos. Os fluxos de dados comumente encontrados em acessos a arquivos, redes de computadores e outros geralmente tratam a informação octeto a octeto, quando não bit a bit. Sendo assim, o tratamento da criptografia em um fluxo de dados, deve ser encapsulado de alguma maneira, de forma a permitir que o programador não tenha que tratar manualmente os processos de encriptação e decriptação, enquanto envia os dados através do canal.

Na plataforma Java, o tratamento de fluxos de dados é simplificado pela hierarquia de herança das classes `InputStream` e `OutputStream`. Estas classes implementam as características comuns aos fluxos de dados, e são herdadas por outras classes que realmente implementam esses fluxos e que estão ligadas a recursos do mundo real, como arquivos, conexões de rede, portas de comunicação e outros.

Dessa forma, recursos como criptografia podem ser implementados de forma transparente dentro de fluxos de dados, e o presente modelo utiliza esta técnica para permitir que quaisquer fluxos de dados possam ser encriptados sem que o programador tenha que tratar as idiossincrasias resultantes da colocação de um fluxo de bytes em um algoritmo de criptografia em blocos de bits.

As classes herdam de FilterInputStream e FilterOutputStream, implementando recursos de criptografia nos fluxos de dados de forma transparente, desde que sejam inicializadas com uma instância da classe `Cripto` – na realidade de uma classe que herda de `Cripto` – para realizar as funções de criptografia.

Os métodos das classes FilterOutputStream e FilterInputStream são sobrepostos para tratar das operações de criptografia. A cada operação de *write* ou *read*, os processos de criptografia embutidos, mostram as informações em seu estado puro, escondendo o processo do usuário da classe.

Os *streams* encriptados deverão ser conectados a um *stream* de base para permitir seu funcionamento com criptografia. Uma instância da classe `Cripto` será utilizada para cuidar da criptografia, sendo que a mesma chave e o mesmo algoritmo deverão ser utilizados no processo de decriptação, para podermos ter um resultado satisfatório. Como as classes podem se conectar a qualquer *stream*, temos a flexibilidade de transmitir em rede, gravar em um arquivo, ou utilizar qualquer outro *stream* acoplado ao próprio *stream* encriptado, como mostrado na Figura 1.

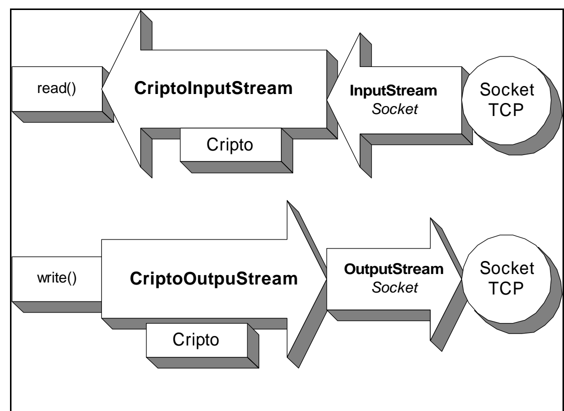

Figura 1: Encriptação de canais utilizando o modelo.

5. ALGORITMOS IMPLEMENTADOS

Para a validação do modelo, foram implementados dois algoritmos de criptografia simétrica, o DES e o AES. O algoritmo DES também foi implementado na sua versão tripla, mantendo a segurança aceitável. As implementações herdam da classe `Cripto`, mantendo a interface dos métodos padronizada, porém, os construtores de cada classe levam em conta as características únicas de cada algoritmo, e aceitam parâmetros de acordo com a necessidade.

Algoritmo DES

Apesar de atualmente o algoritmo DES já não ser mais indicado para aplicações críticas, mas seu histórico faz dele uma opção razoável para aplicações não-críticas, além da sua confiabilidade já corroborada por muitos anos de mercado. Por sua importância na padronização de algoritmos e por ter sido um padrão por vários anos, o DES costuma ser apresentado como solução em diversos modelos e bibliotecas de criptografia.

A implementação utilizada segue os padrões do DES, utilizando chaves de 64 bits (56 bits válidos) e tamanho de bloco também de 64 bits. Foi utilizada uma otimização desenvolvida por David A. Barret, onde as S-Box e as P-Box são combinadas em uma única matriz [1][10].

O seu único construtor recebe a chave como uma *array* de bytes, que será utilizada por toda a existência da instância. Como o tamanho da chave é de 64 bits, o tamanho da *array* apontada por byte[] chave deve ser 8, porém, se o tamanho for maior ou menor, o construtor irá fazer ajustes para tornar a chave compatível.

Algoritmo AES (Rijndael)

O algoritmo Rijndael será colocado como substituto do DES como padrão para criptografia de escala comercial. É mais seguro, mais rápido, foi projetado para funcionar em software, e terá uma vida útil de algumas décadas. Devido a isso, é interessante para um modelo de criptografia oferecer uma implementação dele, permitindo que os desenvolvedores possam adicionar segurança às suas aplicações [12].

A implementação seguida foi da Cryptix Development Team, que é, atualmente, a implementação oficial do AES (Rijndael). O algoritmo AES não tem restrições quanto à utilização e implementação, apesar disso, algumas implementações podem utilizar ferramentas de desenvolvimento e bibliotecas que possuam marcas registradas. No caso desta implementação, as porções de código que dependiam de bibliotecas comerciais da própria Cryptix foram reescritas para evitar a quebra de direitos autorais.

O AES permite tamanhos de bloco e de chave de 128, 192 e 256 bits. A implementação utilizada está otimizada para tamanho de bloco e de chave de 128 bits, que tende a ser o mais comum pelos próximos anos[11].

Como os tamanhos de chave e bloco são variáveis, os construtores precisam refletir isso. Os tamanhos de bloco e chave são dados em número de octetos, ou seja, para 128 bits, temos 16 octetos, para 192 bits, temos 24 octetos e para 256 bits temos 32 octetos. Se os tamanhos de bloco ou chave não forem um dos valores válidos (16, 24 e 32) o valor padrão de 16 (128 bits) é seguido. Se a chave não for do tamanho apropriado, ela será cortada ou será preenchida com caracteres padrão.

6. IMPLEMENTAÇÃO BASEADA NO MODELO

A implementação de sistemas baseados no presente modelo segue regras suficientes para garantir que os processos de criptografia aconteçam de forma adequada e que a arquitetura original do sistema não precise ser alterada drasticamente.

Para se utilizar os algoritmos de criptografia, é necessário antes instanciar apropriadamente uma classe que herde de Cripto, como a classe DES ou a classe AES, e após isso, utilizar os métodos que encriptam os blocos de texto claro. Na realidade, o modelo é construído para que o desenvolvedor não utilize diretamente os métodos de criptografia, mas sim, utilize os *streams* criptografados, necessitando apenas indicar uma chave no construtor da classe do algoritmo. Esse processo evita que o programador tenha que tratar diretamente das questões de *padding*, adequação do algoritmo de bloco ao *stream* e outras.

Para se iniciar um *stream* criptografado, deve-se instanciar uma classe que herde de Cripto, em seguida, passá-la como parâmetro para as classes CriptoInputStream e CriptoOutputStream, que deverão por sua vez se conectar a um *stream* real que será o transporte dos dados.

7. CONCLUSÕES E RESULTADOS OBTIDOS

O presente modelo foi testado em relação a sua segurança e seu desempenho, criando parâmetros de comparação com outras implementações e modelos. Os parâmetros dos testes, bem como o ambiente utilizado será descrito em seguida. Procurou-se utilizar os ambientes mais comuns do mercado e com maior representatividade junto à base instalada de máquinas virtuais Java.

Para demonstrar a funcionalidade do modelo, foram implementados programas que utilizam diferentes aspectos da arquitetura do modelo, mostrando de que forma diferentes tipos de *streams* podem ser tratados pelo modelo.

Análise de segurança

A segurança do modelo reside em dois aspectos principais. Como primeiro aspecto, temos a segurança dos algoritmos em si, com a aderência aos padrões especificados e a garantia da comunidade acadêmica em relação à arquitetura dos algoritmos.

O segundo aspeto diz respeito em relação ao modo como as informações são transmitidas através do *stream* de dados. Quando as classes CriptoInputStream e CriptoOutputStream detectam um *stream* de dados para arquivos, são suprimidos os tamanhos de blocos, impedindo uma análise que perceba pela freqüência a presença dos tamanhos dos blocos. O único tamanho transmitido é o do último bloco, que sofreu preenchimento para se ajustar ao tamanho do algoritmo. Como o tamanho do preenchimento é estatisticamente improvável de se repetir, as análises não poderão utilizar esta informação para tentar quebrar o protocolo.

Em relação à transmissão por *streams* interativos, como conexões de rede, os tamanhos de blocos são transmitidos quando o *stream* de base gera uma chamada ao método *flush*, ou o limite de tamanho é atingido. Apesar desses valores ainda poderem ser analisados, a probabilidade de repetição dos tamanhos é pequena, em sistemas responsivos. Porém, quando se transmitem grandes blocos de informação, um ataque por texto claro conhecido é possível. No caso desse tipo de transmissões, se um algoritmo fraco for utilizado, o programa deverá utilizar chaves de sessões para proteger a informação. Se algoritmos fortes – como por exemplo, o AES – forem utilizados, as probabilidades de comprometimento de segurança são mínimas.

Análises de Desempenho

Os testes de desempenho do modelo se baseiam no funcionamento dos *streams* criptografados, já que os algoritmos de criptografia em si já possuem numerosos testes de desempenho e já é de conhecimento público. Ainda assim, os testes de desempenho dos *streams* refletem as diferenças que os próprios algoritmos possuem em termos de taxa de blocos ou bytes criptografados por segundo.

O ambiente de testes é composto por dois microcomputadores, um baseado em um processador Intel Pentium III de 550 Mhz, com 256 Mb de RAM e outro baseado em um processador AMD K6-2 450 Mhz. Para executar os programas de testes, foram utilizados os sistemas operacionais MS-Windows NT Server 4.0 e MS-Windows 2000 Professional.

A máquina virtual Java utilizada foi a J2SDK (Java 2 Software Development Kit) 1.3 Standard Edition da Sun Microsystems [5]. Os dois computadores estão interligados por uma rede Ethernet a 100 Mbps diretamente, sem a interferência de outros computadores. Como nessa situação, a rede Ethernet passa a ter um volume muito baixo de tráfego, o número de colisões passa a ser desprezível, com toda a banda disponível dedicada à transmissão.

Os programas de teste são um servidor simples, que escuta na porta TCP 9000, e um programa cliente que se conecta ao servidor. É transmitido então um bloco de 1 Mb de texto, através dos *streams* criptografados pelas classes CriptoInputStream e CriptoOutpuStream. O tempo decorrido somente da transmissão é contado em milissegundos por funções internas da linguagem Java. Foram testadas as transmissões sem encriptação, com encriptação utilizando DES e utilizando AES com seus três tamanhos de chave e de bloco. Foram coletadas 15 amostras de tempo em períodos diferentes. A Figura 2 mostra um gráfico comparativo do desempenho dos algoritmos, comparados com uma transmissão em texto claro.

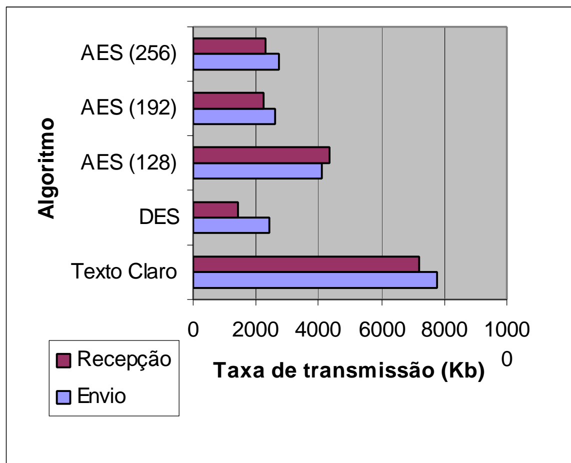

Figura 2: Comparativo de desempenho.

REFERÊNCIAS

[1] B. Schneier, "Applied Cryptography – Protocols, Algorithms and Source Code in C", second edition, John Wiley & Sons Inc., 1996

[2] C. Horstmann & G. Cornell, "Core JAVA 2 – Volume I – Fundamentals", Prentice Hall, 1999

[3] C. Horstmann & G. Cornell, "Core JAVA 2 – Volume II – Advanced Features", Prentice Hall, 2000

[4] R. E. Harols, "Java Networking Programming 2nd ed.", O'Reilly, 2000

[5] Sun Inc. , JDK 1.3 Documentation

[6] D. R. Stinson, "Cryptography : Theory and Practice (Discrete Mathematics and Its Applications)" – CRC Pr 1995

[7] Netscape Inc, "Secure Socket Layer Documentation", http://www.netscape.com, 1998

[8] SET, "SET – Secure Electronic Transaction 1.0 Specification", http://www.setco.org/set_specifications.html, 1998

[9] WASSENAAR, "The Wasssenaar Arrangement, 'Export Controls for Conventional Arms and Dual-Use Goods and Technologies'", http://www.wassenaar.org, 1999

[10] R. Terada, "Segurança de Dados, Criptografia em Redes de Computador", Editora Edgard Blücher, 2000

[11] J. Daemen e V. Rijmen, "The Rijndael Block Cipher", AES Proposal – NIST, 2000

[12] NIST, "AES Suvey", http://www.nist.gov/aes, 2000