UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SISTEMA DE PROTEÇÃO PARA SERVIDORES DE JOGOS *ONLINE* CONTRA SOFTWARES CLIENTES NÃO OFICIAIS

THIAGO ALEXANDRE GESSER

BLUMENAU
2010

2010/2-27

THIAGO ALEXANDRE GESSER

SISTEMA DE PROTEÇÃO PARA SERVIDORES DE JOGOS *ONLINE* CONTRA SOFTWARES CLIENTES NÃO OFICIAIS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Paulo Fernando da Silva, Mestre - Orientador

BLUMENAU
2010

2010/2-27

SISTEMA DE PROTEÇÃO PARA SERVIDORES DE JOGOS *ONLINE* CONTRA SOFTWARES CLIENTES NÃO OFICIAIS

Por

THIAGO ALEXANDRE GESSER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo Fernando da Silva, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB

Blumenau, 07 de dezembro de 2010

RESUMO

Este trabalho apresenta o desenvolvimento de um sistema de proteção para softwares servidores contra softwares clientes não oficiais. A proteção oferecida faz com que um software servidor só se comunique com softwares cliente em que ele confia. Todo software cliente que deseja se conectar em um software servidor protegido passa por validações antes e depois de se conectar para provar que é confiável. Todas as validações feitas pelo sistema são automáticas, sendo abstraídas totalmente do usuário do software cliente. Este tipo de segurança é necessário em jogos *online*, pois softwares clientes adulterados podem possibilitar trapaças. Para prevenir a adulteração do próprio sistema, ele integra estudos atuais da área de segurança e de outras áreas, como a criptografia de caixa branca e atualização dinâmica de código. O sistema é extensível, permitindo a fácil implementação de novos tipos de validações.

Palavras-chave: Arquitetura cliente/servidor. Softwares clientes não oficiais. Proteção. Jogos *online*. Criptografia. *Hashing*. Atualização dinâmica de código.

ABSTRACT

This work presents the development of a protection system for server softwares against unofficial client softwares. The offered protection makes that a server software only communicates with client software that it trusts. All client software that wants to connect on the protected server software passes by validations before and after connecting to prove he is trustworthy. All validations done by the system are automatic, and fully abstracted from the client software user. This type of security is needed in online games, because tampered client softwares can enable cheating. To prevent the tampering of the system itself, it integrates current studies of the security area and other areas, such as white-box cryptography and dynamic code update. The system is extensible, allowing easy implementation of new types of validations.

Key-words: Client/server architecture. Unofficial client software. Protection. Online games. Cryptography. Hashing. Dynamic code update.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AES – *Advanced Encryption Standard*

SHA – *Secure Hash Algorithm* -1

MD5 – *Message Digest* 5

SSL – *Secure Socket Layer*

JVM – Java *Virtual Machine*

JBCO – Java *ByteCode Obfuscator*

DDoS – *Distributed Denial of Service*

UML – *Unified Model Language*

CBC – *Cipher Block Chaining*

API – *Application Programming Interface*

LZMA – Lempel Ziv Markov *Algorithm*

IDE – *Integrated Development Environment*

JDK – Java *Development Kit*

SUMÁRIO

1 INTRODUÇÃO

Atualmente há uma enorme variedade de jogos *online* disponíveis para pessoas do mundo inteiro. Eles podem variar muito em gênero, história e público alvo, mas uma coisa todos tem em comum: a necessidade da comunicação entre os jogadores. Esta comunicação é feita através da interface (cliente) desenvolvida pela empresa que mantém o jogo.

O jogador interage com a interface, que por sua vez troca mensagens com um servidor ou diretamente com outros clientes, dependendo da arquitetura de comunicação adotada pelo jogo. As mensagens servem para avisar sobre as ações de determinado jogador aos demais, permitindo que eles reajam com outras ações que também serão compartilhadas com todos. As mensagens não passam de pacotes que seguem um protocolo específico do jogo. Sendo assim, torna-se simples reproduzir estas mensagens através de uma aplicação própria. É assim que nasce um cliente não oficial.

Através de um cliente não oficial é possível trapacear o jogo de diversas formas. Uma delas é criando algoritmos que simulam um jogador de verdade ou até que vão além de um jogador normal. Assim quem utiliza esta aplicação pode deixar que ela jogue para si, poupando muito esforço que os demais jogadores terão. Este tipo de aplicação é conhecida como *bot*.

Ao longo dos anos surgiram varias soluções de prevenção aos clientes não oficiais. Algumas delas exigiam que o próprio jogador provasse que estava utilizando um cliente oficial através de, por exemplo, um Captcha[1]. Mas este tipo de verificação além de ser um grande incômodo, é suscetível a vários tipos de ataques. Outros tipos de soluções funcionam sem o conhecimento do jogador, exigindo que o cliente prove que é oficial de maneira automática. Mas independente do tipo de verificação utilizado, um agressor pode tentar descobrir como ela é feita e criar uma maneira de burlar esta verificação. Sendo que a maneira utilizada para quebrar a segurança pode ser facilmente transcrita para um algoritmo e assim colocado em uma aplicação que varias outras pessoas poderão usar para explorar a mesma falha.

Diante do exposto, foi desenvolvido um sistema que consegue bloquear de forma efetiva o uso de clientes não oficiais em jogos *online*. O sistema funciona de forma

[1] Captcha é um programa de proteção contra *bots* que funciona através da geração de testes que humanos conseguem passar, mas os computadores atuais não. Por exemplo, usando textos distorcidos, que seres humanos conseguem ler facilmente, ao contrário dos computadores (CARNEGIE MELLON UNIVERSITY, 2009).

automática, sem qualquer envolvimento do jogador. O bloqueio de clientes não oficiais é feito através da verificação da autenticidade do cliente. A verificação de autenticidade baseia-se na igualdade do arquivo executável que o jogador tenta utilizar como cliente com o arquivo executável do cliente oficial, armazenado no servidor. Além disso, o estado do software cliente em execução e o ambiente de execução em si também podem ser verificados. Ou seja, o cliente só pode se conectar no servidor de jogo *online* se ele provar que está usando um cliente idêntico ao oficial e que não há nenhuma alteração indevida no ambiente de execução. Depois de conectado, as verificações de autenticidade continuam acontecendo, fazendo com que clientes adulterados sejam desconectados do servidor. A comparação do conteúdo dos arquivos executáveis é feita utilizando um algoritmo de *hashing*[2]. Com este algoritmo, é possível calcular os valores de *hash* dos conteúdos dos arquivos que se deseja comparar e apenas verificar a igualdade desses valores ao invés de comparar todo o conteúdo dos arquivos em si.

O sistema desenvolvido está dividido em dois componentes: o componente servidor e o componente cliente. O componente cliente deve ser consultado quando o jogador tenta abrir o cliente do jogo *online* para que a sua autenticidade seja avaliada. Este componente só irá obter as informações sobre o cliente e passá-las para o componente servidor, que fará a avaliação de fato. Se um cliente teve a sua autenticidade validada, poderá se conectar no servidor de jogo *online*. O servidor de jogo *online* deverá consultar o componente servidor quando um cliente tenta se conectar nele para verificar se o cliente realmente foi validado.

É imprescindível que a comunicação entre os componentes do sistema seja feita de maneira segura. Se a troca de informações feita entre estes componentes for escutada ou alterada por algum agressor, há chances de que o bloqueio proposto seja burlado. Por isso, são utilizadas maneiras de garantir a confidencialidade e a integridade dos dados durante as comunicações.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um sistema de proteção para jogos *online* que

[2] Um algoritmo de *hashing* é aquele que recebe como entrada um bloco de dados de qualquer tamanho e converte-o em um número aparentemente aleatório de tamanho fixo, chamado de *hash*. O mesmo bloco de dados sempre resultará no mesmo *hash*, desde que seja utilizado o mesmo algoritmo de *hashing* (GURTNER, 2007).

impeça a utilização de softwares clientes não oficiais.

Os objetivos específicos do trabalho são:

a) disponibilizar um software para o computador servidor (validador) que intermedie a comunicação entre o software servidor e os softwares clientes do jogo, permitindo apenas o acesso de clientes identificados como oficiais;
b) disponibilizar um software para o computador cliente (monitor) que intermedie a comunicação entre o software cliente e o software servidor do jogo e transmita ao servidor informações sobre o cliente, a fim de identificar sua oficialidade;
c) disponibilizar um mecanismo para que o validador possa garantir que as informações transmitidas sobre o software cliente sejam provenientes do monitor original e não de um outro software que imite seu comportamento e transmita informações falsas, a fim de permitir o acesso de um cliente não oficial.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em três capítulos, intitulados respectivamente como fundamentação teórica, desenvolvimento e conclusões.

O capítulo dois apresenta os aspectos teóricos estudados para o desenvolvimento do trabalho. São abordados os seguintes temas: jogos *online*, criptografia, algoritmos de *hashing*, assinatura digital, certificado digital, protocolos de comunicação segura, ataques contra a criptografia, criptografia de caixa branca, algoritmos de *hashing* gerados aleatoriamente, confiabilidade de software remoto, proteção de código Java, atualização dinâmica de código Java e por fim são apresentados os trabalhos correlatos.

No capítulo três é apresentada a especificação do trabalho, principais casos de uso, bem como os requisitos do sistema. Por último, no capítulo quatro tem-se a conclusão do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta inicialmente uma visão geral sobre jogos *online*, sua arquitetura e os problemas de segurança que eles possuem. A seguir são descritas as definições de criptografia e algoritmos de *hashing*. Logo após são apresentadas os conceitos tradicionais de área de segurança, como a assinatura digital, certificado digital e protocolos de comunicação segura. Posteriormente são apresentadas as inovações: a criptografia de caixa branca (precedida pela contextualização dos cenários de ataque contra a criptografia) e os algoritmos de *hashing* gerados aleatoriamente. A seguir é demonstrada uma maneira de como se garantir a confiabilidade de um software executado em um ambiente remoto não confiável. Posteriormente, são descritos os conceitos de proteção e atualização dinâmica de código Java. Por fim, nos trabalhos correlatos, são apresentadas três ferramentas de segurança para jogos *online* que são amplamente utilizadas atualmente.

2.1 JOGOS *ONLINE*

São chamados jogos *online* os jogos eletrônicos jogados em uma rede de computadores. Neles, um jogador conectado a determinada rede (como a internet, por exemplo), pode jogar com outros sem que ambos precisem estar no mesmo ambiente, sem sair de casa. O jogador pode desafiar adversários que estejam em outros lugares do país, ou até do mundo. Tudo em tempo real, como se o outro estivesse lado a lado. Desta forma, esta categoria de jogos abre novas perspectivas de diversão (JOGOS..., 2010). O número de jogadores de jogos deste tipo vem crescendo, o que faz com que eles tornem-se cada vez mais famosos. Por exemplo, estima-se que o jogo *online* conhecido como Ragnarök Online possua mais de 25 milhões de jogadores no mundo todo (RAGNARÖK..., 2010).

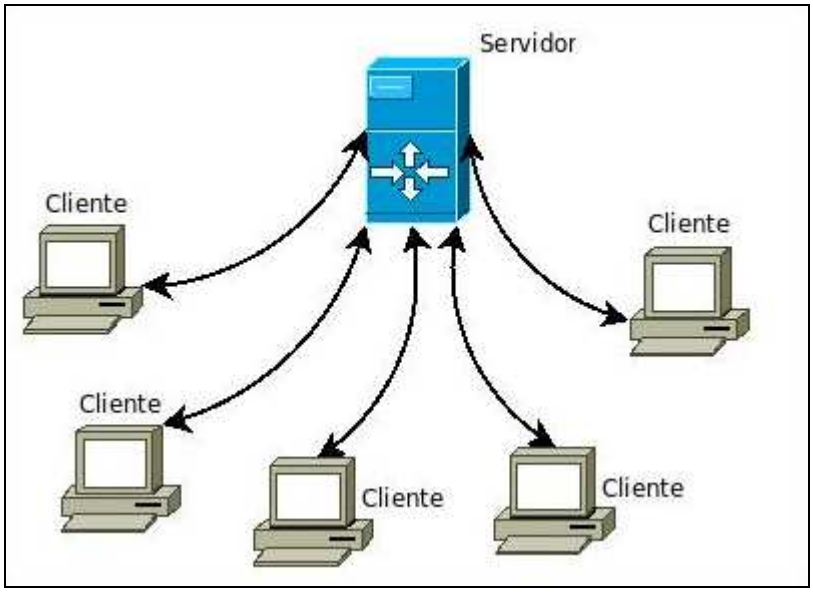
2.1.1 Arquitetura

O jogador interage com o jogo *online* através de um software que executa em seu computador. A maneira como os softwares dos jogadores se comunicam depende do tipo de

arquitetura adotado pelo jogo *online*, podendo ser ponto a ponto ou cliente/servidor. Como o tipo ponto a ponto é pouco utilizado, ele não é abordado neste trabalho.

> No modelo de arquitetura Cliente/Servidor, existem dois processos envolvidos, um no host cliente e um outro no host servidor. A comunicação acontece quando um cliente envia uma solicitação pela rede ao processo servidor, e então o processo servidor recebe a mensagem, e executa o trabalho solicitado ou procura pelos dados requisitados e envia uma resposta de volta ao cliente, que estava aguardando. Nesta arquitetura o servidor tem uma aplicação que fornece um determinado serviço e os clientes tem aplicações que utilizam este serviço. Uma característica desta arquitetura, é que um cliente não se comunica com outro cliente, e o servidor, que tem um endereço fixo, esta sempre em funcionamento. (INTRODUÇÃO..., 2010).

O modelo de organização da arquitetura cliente/servidor pode ser visto na Figura 1.



Fonte: Introdução... (2010).

Figura 1 – Modelo cliente/servidor

Através da arquitetura cliente/servidor, os softwares que executam nos computadores dos jogadores tornam-se softwares clientes. Estes softwares se comunicam apenas com o software servidor que executa no computador servidor disponibilizado pela empresa que mantém o jogo *online*.

2.1.2 Problemas de segurança

O surgimento de jogos *online* mudou fundamentalmente a exigência de segurança para jogos de computador. Neste novo contexto, a proteção contra cópias não é mais o único problema de segurança. Apesar dos jogos *online*, por outro lado, serem comumente considerados como aplicações distribuídas de comércio eletrônico, eles tem seus próprios desafios de segurança (CHOI; YAN, 2002, p. 12). Estes problemas de segurança, quando

explorados, possibilitam o uso de diversos tipos de trapaças nos jogos *online*.

Um dos problemas de segurança é a possibilidade de modificação do software cliente do jogo, que pode ser feito na verdade em qualquer aplicação cliente/servidor ou ponto a ponto. Segundo Choi e Yan (2002, p. 7), a alteração de softwares de jogos tem sido uma prática comum desde o início da era dos jogos de computador. Existem diversas ferramentas que auxiliam a modificar os arquivos, a memória e até a fazer engenharia reversa nestes softwares. Desta forma, uma pessoa pode moldar o software do jogo como desejar e obter vantagens em relação aos demais jogadores.

De acordo com Choi e Yan (2002, p. 7), para evitar este tipo de problema é possível projetar protocolos de validação para o software cliente com o auxílio da criptografia. Estes protocolos podem fazer a validação quando o software cliente inicializa a conexão com o software servidor e depois periodicamente enquanto o jogo estiver rodando. Quando a validação falhar, o software servidor pode tomar as ações apropriadas, como por exemplo, desconectar o software cliente.

2.2 CRIPTOGRAFIA

A criptografia ou cifragem consiste na aplicação de um algoritmo aos dados de forma que eles se tornem ilegíveis. Para recuperar os dados originais será necessário conhecer o algoritmo de decriptação ou decifragem. As aplicações básicas da criptografia são a confidencialidade (garantir que apenas quem foi autorizado pode ler os dados), a autenticação (garantir que os dados têm a origem correta) e a integridade (garantir que os dados não foram alterados entre origem e destino) (MOREIRA, 2002).

Segundo Silva (2004, p. 44), o segredo da criptografia não está no algoritmo empregado, mas sim na chave de criptografia utilizada. Existem dois tipos de chave: a chave simétrica e a assimétrica.

2.2.1 Chave simétrica

A chave simétrica também é conhecida como chave secreta. Segundo Silva (2004, p. 44), ela é compartilhada pelos dois pontos, ou seja, tanto o emissor quanto o receptor sabem

como encriptar e decriptar as mensagens. O problema deste tipo de chave é fazer com que somente os envolvidos na conversa a conheçam. Para isto, existe o estudo do gerenciamento de chave simétrica.

2.2.1.1 Gerenciamento de chave simétrica

> A criptografia de chave simétrica pode manter seguro seus segredos, mas pelo fato de você precisar das suas chaves para recuperar os dados criptografados, você também deve mantê-las seguras. O processo pra manter suas chaves seguras e disponíveis para utilização é conhecido como gerenciamento de chave. (BURNETT; PAINE, 2002, p. 45).

De acordo com Burnett e Paine (2002, p. 45-65), uma chave simétrica pode ser protegida de duas formas: através de uma senha que apenas o dono da chave conhece ou através de um hardware (chamado de *token*) que apenas o dono da chave possui. Com a senha é possível criptografar a chave e armazená-la de certa forma em segurança. Com o *token,* a chave fica armazenada em nele mesmo, fazendo com que os dados só possam ser decriptados se o *token* for conectado ao computador.

Estas duas formas de proteção de chaves requerem uma interação com o dono da chave, o que geralmente é um ser humano. Mas e quando o dono da chave for um software? Certamente ele precisaria armazenar a chave em algum local seguro para prevenir possíveis invasões para descobri-la. Se não qualquer um que obtiver a chave poderá se passar pelo software.

2.2.1.2 *Advanced Encryption Standard* (AES)

Segundo Faleiros e Rosa (2003, p. 1-6), o AES é o algoritmo de criptografia de chave simétrica oficial do governo dos Estados Unidos. Ele foi definido a partir de um concurso que iniciou em 1997 e terminou em 2000. O algoritmo escolhido para representar o AES foi o Rijndael. O Rijndael foi desenvolvido por Vincent Rijmen e Joan Daemen.

O AES foi projetado para usar somente simples operações de bytes completos. Ele funciona criptografando blocos de 128 bits de dados. A chave por sua vez pode ter 128, 192 ou 256 bits. Para criptografar um bloco de dados, o AES opera sobre ele como matriz de bytes com 4 x 4 posições (4 x 4 bytes = 16 bytes = 128 bits). Esta matriz de bytes é denominada de

estado. O processo de criptografia consiste em várias rodadas regulares compostas pelas mesmas operações, sendo que ao final é executada uma rodada extra com uma das operações omitidas. O número de rodadas depende do tamanho da chave, sendo:

a) 9 se a chave for de 128 bits;
b) 11 se a chave for de 192 bits;
c) 13 se a chave for de 256 bits.

Na inicialização do algoritmo, são calculadas sub-chaves derivadas da chave principal, utilizando o algoritmo de agendamento de chaves. Cada sub-chave calculada corresponde a uma rodada do AES, sendo que uma chave adicional é calculada para ser utilizada no início da criptografia do bloco. As sub-chaves possuem o mesmo tamanho do estado.

Na encriptação de dados pelo AES, cada rodada regular é composta por quatro operações: `SubBytes`, `ShiftRows`, `MixColumns` e `AddRoundKey`. Antes da primeira rodada, um `AddRoundKey` adicional é executado. A rodada extra é composta apenas por três operações: `SubBytes`, `ShiftRows` e `AddRoundKey`. As etapas da encriptação do AES podem ser vistos no Quadro 1.

```
AddRoundKey(0)
para rodada = 1 até NUMERO_DE_RODADAS-1 {
      SubBytes()
      ShiftRows()
      MixColumns()
      AddRoundKey(rodada)
}
SubBytes()
ShiftRows()
AddRoundKey(NUMERO_DE_RODADAS)
```

Quadro 1 – Passos da encriptação do AES

A seguir, as operações de encriptação do AES são detalhadas. Ao final, é apresentado o funcionamento da decriptação de dados pelo AES.

Na etapa `SubBytes`, cada byte do estado é substituído por um byte correspondente ao seu valor como índice de um *array* de bytes de substituição, denominado `S-Box`. O Quadro 2 mostra a `S-Box` do AES. Por exemplo, o byte `1` seria substituído pelo byte `124`, pois é o byte localizado no índice `1` da `S-Box`.

99	124	119	123	242	107	111	197	48	1	103	43	254	215	171	118
202	130	201	125	250	89	71	240	173	212	162	175	156	164	114	192
183	253	147	38	54	63	247	204	52	165	229	241	113	216	49	21
4	199	35	195	24	150	5	154	7	18	128	226	235	39	178	117
9	131	44	26	27	110	90	160	82	59	214	179	41	227	47	132
83	209	0	237	32	252	177	91	106	203	190	57	74	76	88	207
208	239	170	251	67	77	51	133	69	249	2	127	80	60	159	168
81	163	64	143	146	157	56	245	188	182	218	33	16	255	243	210
205	12	19	236	95	151	68	23	196	167	126	61	100	93	25	115
96	129	79	220	34	42	144	136	70	238	184	20	222	94	11	219
224	50	58	10	73	6	36	92	194	211	172	98	145	149	228	121
231	200	55	109	141	213	78	169	108	86	244	234	101	122	174	8
186	120	37	46	28	166	180	198	232	221	116	31	75	189	139	138
112	62	181	102	72	3	246	14	97	53	87	185	134	193	29	158
225	248	152	17	105	217	142	148	155	30	135	233	206	85	40	223
140	161	137	13	191	230	66	104	65	153	45	15	176	84	187	22

Fonte: Faleiros e Rosa (2003, p. 3-4).

Quadro 2 – A `S-Box` do AES

Na etapa `ShiftRows`, Cada linha da matriz de estado (exceto a primeira) tem seus bytes deslocados. Um exemplo da maneira como os bytes são deslocados é mostrada no Quadro 3.

De:	**Para:**
1 5 9 13	1 5 9 13
2 6 10 14	6 10 14 2
3 7 11 15	11 15 3 7
4 8 12 16	16 4 8 12

Fonte: Faleiros e Rosa (2003, p. 4).

Quadro 3 – Exemplo de funcionamento do `ShiftRows` do AES

Na etapa `MixColumns`, cada coluna da matriz de estado é multiplicada pela matriz mostrada no Quadro 4. Porém, nesta multiplicação, os bytes são tratados como polinômios em lugar de números.

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Fonte: Faleiros e Rosa (2003, p. 3-4).

Quadro 4 – A matriz utilizada na etapa `MixColumns`

Na etapa `AddRoundKey`, cada byte do estado é somado ao byte correspondente da sub-chave gerada para a rodada corrente. A soma é feita através da operação `XOR` (ou exclusivo).

2.2.1.2.1 Decriptação dos dados

A decriptação dos dados é feita executando as etapas inversas da encriptação. Cada operação de encriptação do AES possui uma operação inversa correspondente, a não ser o `AddRoundKey` , pois o inverso é ele mesmo. Sendo assim, o inverso do `SubBytes` é o `InvSubBytes`, que substitui os bytes do estado utilizando uma `S-Box` invertida, fazendo com que os bytes substituídos pela `S-Box` normal voltem a ser o que eram. O inverso do `ShiftRows` é o `InvShiftRows`, que desloca os bytes das linhas do estado ao contrário. Por fim, o inverso do `MixColumns` é o `InvMixColumns`, que multiplica cada coluna do estado por uma matriz inversa correspondente (mostrada no Quadro 5).

```
14 11 13 9
9 14 11 13
13 9 14 11
11 13 9 14
```

Fonte: Faleiros e Rosa (2003, p. 6).

Quadro 5 – A matriz utilizada na etapa `InvMixColumns`

As etapas da decriptação do AES podem ser vistas no Quadro 6.

```
AddRoundKey(NUMERO_DE_RODADAS)
para rodada = NUMERO_DE_RODADAS-1 até 1 {
      InvShiftRows()
      InvSubBytes()
      AddRoundKey(rodada)
      InvMixColumns()
}
InvShiftRows()
InvSubBytes()
AddRoundKey(0)
```

Quadro 6 – Passos da decriptação do AES

2.2.2 Chave assimétrica

A chave assimétrica também é conhecida como chave pública. De acordo com Silva (2004, p. 44), este conceito de chave é bem mais amplo que o de chave simétrica. Na criptografia assimétrica a chave é dividida em duas partes relacionadas matematicamente. Uma parte é distribuída livremente como uma chave pública para possíveis emissores e a

outra fica guardada para o receptor como uma chave privada. Desta forma, quem desejar comunicar-se com o receptor tem que criptografar a mensagem utilizando a chave pública. Quando o receptor receber esta mensagem, utilizará sua chave privada para decritpá-la. Assim, qualquer um pode conversar com o receptor, mas apenas ele vai conseguir entender o conteúdo das mensagens, pois não é viável decriptar uma mensagem a partir da chave publica.

2.3 ALGORITMOS DE *HASHING*

Segundo Silva (2004, p. 65), um algoritmo de *hashing* é aquele que gera um valor pequeno, de tamanho fixo, chamado de *digest* ou valor *hash*, derivado de uma mensagem de entrada, de qualquer tamanho. O *hashing* também é denominado de *message digest.* O *digest* pode ser comparado a um dígito verificador de uma conta corrente ou o *cheksum* de valores. Serve, portanto, para garantir a integridade do conteúdo da mensagem que está sendo representada. Assim, após o valor *hash* de uma mensagem ter sido calculado mediante o emprego de um algoritmo de *hashing*, qualquer modificação em seu conteúdo, mesmo que apenas um bit, será detectada, pois um novo cálculo do valor *hash* sobre o conteúdo modificado resultará em um valor *hash* diferente. Existe um número razoável de algoritmos de *hashing* disponíveis, mas os mais seguros no momento são os seguintes:

a) *Secure Hash Algorithm* (SHA-1): o SHA-1 é considerado seguro porque é computacionalmente inviável encontrar uma mensagem correspondente a determinado *hash* gerado, ou encontrar duas mensagens que produzem *hashes* idênticos. O *hash* que ele gera é composto por 160 bits;
b) *Message Digest* 5 (MD5): assim como o SHA-1, o MD5 também é considerado seguro porque gera *hashes* únicos e invioláveis. O *hash* que ele gera é composto por 128 bits.

2.4 ASSINATURA DIGITAL

No mundo real, vários tipos de documentos só se tornam legalmente válidos depois que são assinados. As nossas assinaturas pessoais, em teoria, possuem algumas características

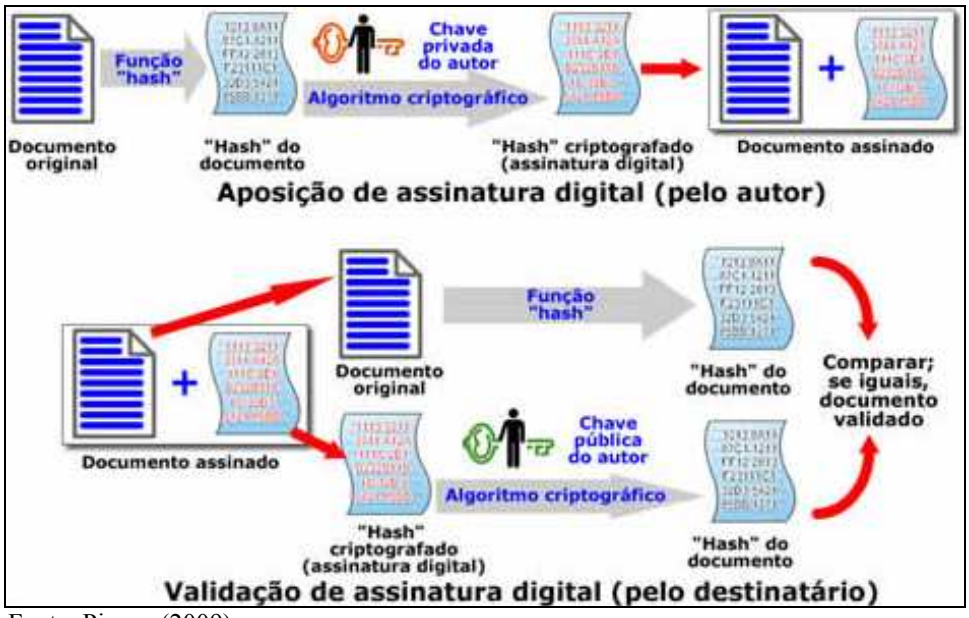
básicas:

a) a assinatura é autêntica. Ela pode convencer o recipiente do documento de que o signatário realmente assinou o nome dele;
b) a assinatura não é reutilizável. É escrita no documento e não pode ser movida para um documento diferente;
c) o documento assinado é inalterável. Depois que foi assinado, não pode ser alterado;
d) a assinatura não pode ser repudiada. O signatário não pode contestar a autoria do que assinou.

No universo virtual, também surge a necessidade de assinar documentos na forma digital, por meio da assim chamada assinatura digital. A assinatura digital nada mais é que um valor que somente pode ser gerado pelo emitente do documento, incorporado no corpo deste e verificado pelo receptor por meio de um processo específico (SILVA, 2004, p. 182-183). A assinatura digital funciona a partir de dois recursos que são amplamente utilizados individualmente na área de segurança: a criptografia de chave assimétrica e os algoritmos de *hashing*.

Segundo Burnett e Paine (2002, p. 116-120), normalmente quando se utiliza a criptografia de chave assimétrica, uma mensagem é encriptada com a chave pública e apenas quem possuir a chave privada pode decriptá-la. Mas também é possível fazer o contrário: encriptar a mensagem com a chave privada decriptá-la com a chave pública. Desta forma, apenas quem possuir a chave privada poderá encriptar a mensagem, fazendo com que a chave privada atue como uma espécie de assinatura. Como a chave de decriptação é pública, qualquer um que possuí-la poderá ler a mensagem. Mas criptografar mensagens inteiras pode gerar um grave problema de desempenho, devido à lentidão dos algoritmos de criptografia de chave assimétrica. Portanto, em vez de encriptar a mensagem inteira com a chave privada, o melhor método é encriptar apenas um representante dela, ou seja, um valor *hash*.

Uma assinatura digital, portanto, é o valor *hash* de uma mensagem encriptado com a chave privada de seu autor. Esta assinatura pode então ser anexada a mensagem para que, quem recebê-la, consiga validar sua procedência. Esta validação é feita decriptando a assinatura com a chave pública do autor e validando se o valor *hash* da mensagem é o mesmo valor *hash* que foi assinado. Este processo é ilustrado na Figura 2.



Fonte: Piropo (2009).

Figura 2 – Funcionamento da assinatura digital

2.4.1 Assinatura de código

Assinatura de código é uma técnica utilizada para assinar programas executáveis com uma assinatura digital. Ela é projetada para melhorar a confiabilidade dos softwares distribuídos pela internet através da detecção de alterações em seus códigos executáveis. A assinatura de código também é utilizada pra combater o problema de programas maliciosos, incluindo vírus de computador (GARFINKEL; SPAFFORD, 2002, p. 560). A assinatura de código funciona da mesma forma que uma assinatura digital, mas ao invés de se assinar uma mensagem de texto, se assina o código executável de um software.

Com a assinatura de código, um usuário de um software pode garantir que o código executável que ele irá rodar é o mesmo código que o autor do software disponibilizou. Ou seja, é uma garantia para o computador cliente de que o software que baixou de um computador servidor é autêntico.

2.5 CERTIFICADO DIGITAL

A assinatura digital funciona através de criptografia de chave assimétrica, obrigando o

autor de uma mensagem a possuir uma chave privada e o receptor a chave pública correspondente. Existem várias maneiras de o autor transmitir sua chave pública, mas como uma pessoa qualquer pode verdadeiramente saber se uma chave pública pertence a uma determinada pessoa? Sem esta certeza sobre a chave pública, não é possível garantir que uma mensagem realmente foi assinada por quem diz ser assinada.

Segundo Burnett e Paine (2002, p. 141), a maneira mais comum de saber se uma chave pública pertence ou não à entidade de destino é por meio de um certificado digital. Um certificado digital associa um nome a uma chave pública. Uma analogia para isso seria um passaporte, que associa uma foto a um nome e a um número. Supostamente, um passaporte é feito de tal maneira que o torna perceptível se alguém pegar um passaporte existente e substituir a foto verdadeira com uma foto de um impostor. Isto pode ser um passaporte válido, mas não é válido para a pessoa na foto. As autoridades não honrarão esse passaporte.

Um certificado digital é produzido de tal maneira que o torna perceptível se um impostor pegou um certificado existente e substitui a chave pública ou o nome. Qualquer pessoa ao examinar esse certificado saberá que algo está errado. Talvez o nome ou a chave pública esteja errado, portanto, você não confiará na combinação desse par de chaves e do nome.

O certificado digital é construído a partir de um nome e uma chave pública. Considere essas duas coisas como uma mensagem e assine a mensagem. O certificado é composto pelo nome, pela chave pública e pela assinatura. A única coisa que ainda deve ser determinada é quem assinará o certificado. A assinatura é quase sempre feita por meio de uma autoridade certificadora.

2.5.1 Autoridade certificadora

Segundo Silva (2004, p. 187-189), uma autoridade certificadora é a entidade responsável por emitir certificados digitais. Esses certificados podem ser emitidos para diversos tipos de entidades como: pessoas, computadores e empresas. É possível manter uma autoridade certificadora para uso mais restrito, como em uma empresa, por exemplo, ou contratar os serviços de uma autoridade certificadora externa. Manter sua própria autoridade certificadora traz uma redução de custos, mas faz com que as garantias que ela provê se apliquem apenas na empresa em que é mantida. Outra empresa pode não confiar nos certificados emitidos por ela. Neste caso, o melhor é utilizar uma autoridade certificadora

externa conhecida por ambas as empresas.

De acordo com Silva (2004, p. 142), uma autoridade certificadora possui as seguintes atribuições:

a) emissão de certificados;
b) revogação de certificados;
c) renovação de certificados;
d) publicação da lista de certificados revogados;
e) disponibilizar a situação do certificado quando requerida;
f) gerência de chaves criptográficas;
g) publicação de suas regras operacionais;
h) fiscalização do cumprimento dessa política pelos usuários.

2.6 PROTOCOLOS DE COMUNICAÇÃO SEGURA

Para garantir que a comunicação entre dois softwares está sendo feita de forma segura, é necessária a utilização de um protocolo de segurança. Muitos destes protocolos utilizam a criptografia e o *hashing*, pois eles garantem três aspectos essenciais em uma comunicação segura: a autenticidade, a confidencialidade e a integridade. Um exemplo deste tipo de protocolo é o *Secure Socket Layer* (SSL).

2.6.1 SSL

SSL é uma camada do protocolo de rede, situada exatamente abaixo da camada de aplicação, com a responsabilidade de gerenciar um canal de comunicação seguro entre o cliente e o servidor (BRANDÃO, 2003).

Segundo Garfinkel e Spafford (2002, p. 107-108), SSL adiciona diversas funcionalidades a comunicação, incluindo:

a) autenticação do servidor, utilizando assinaturas digitais;
b) autenticação do cliente, utilizando assinaturas digitais;
c) confidencialidade dos dados através do uso de criptografia;
d) integridade ao código através do uso de códigos de autenticação de mensagens.

A criptografia só funciona se ambas as partes da comunicação conhecerem os mesmos algoritmos. Por isso, SSL é um protocolo extensível e adaptável. Quando um programa usando SSL tenta contatar outro, os dois programas eletronicamente comparam notas, determinando qual é o protocolo criptográfico mais forte que ambos conhecem.

2.7 ATAQUES CONTRA A CRIPTOGRAFIA

De acordo com Chow et al. (2002, p. 4-5) e Wyseur (2009, p. 33-35), é possível distinguir três cenários principais de ataque contra sistemas criptográficos: o cenário de caixa preta, caixa cinza e caixa branca. Os ataques têm como objetivo descobrir a chave criptográfica utilizada.

O cenário de caixa preta é o mais tradicional, em que a comunicação criptografada é feita entre entidades mutuamente confiáveis. Desta forma é possível fazer a distribuição de chaves para que as mensagens possam ser criptografadas e assim tornem-se confidencias. Descobrir a chave neste cenário é uma tarefa muito complexa, pois o invasor possui poucos insumos, como observar a troca de mensagens criptografadas entre as entidades ou outros comportamentos externos. Sendo assim, um ataque de caixa preta funciona de maneira genérica, não dependendo do conhecimento de detalhes sobre o funcionamento do algoritmo de criptografia utilizado.

Ataques de caixa cinza operam em um cenário similar ao dos de caixa preta, mas eles vão além de apenas observar comportamentos externos, buscando informações sobre o funcionamento interno do algoritmo de criptografia utilizado. Através destas informações, torna-se mais simples descobrir a chave criptográfica. Isto mostra que apenas a visibilidade parcial dos comportamentos internos de um algoritmo de criptografia já enfraquece a segurança que ele provê.

Em um cenário de caixa branca, ao contrário dos demais cenários, o invasor possui visibilidade total do código executável do algoritmo de criptografia e controle total do ambiente de execução. Por exemplo, um software servidor e um software cliente de jogo *online* que utilizam criptografia para garantir que estão trocando mensagens de forma segura. Como o software cliente é distribuído para os jogadores, um invasor também terá acesso a ele. Desta forma, ele terá visibilidade da implementação do algoritmo de criptografia utilizado (contido no software cliente) e controle sobre o ambiente de execução (seu próprio

computador), o que configura um cenário de caixa branca.

Segundo Chow et al. (2002, p. 4-5) e Wyseur (2009, p. 33-35), um invasor em um cenário de caixa branca pode extrair a chave criptográfica diretamente do código executável do software protegido se ela estiver armazenada literalmente nele. No caso de chaves que são geradas em tempo de execução, o invasor pode simplesmente observar a memória utilizada pelo software e obtê-la assim que for gerada. Também é possível alterar o código executável do software para fazer o que o invasor desejar, desde que não se altere a parte que criptografa as mensagens. Independente do tipo de invasão utilizado, existem diversas ferramentas disponíveis para auxiliar no ataque. Alguns exemplos de suas funcionalidades são: análise de memória e *debug* do código executável, interceptação das chamadas ao processador ou bibliotecas externas, engenharia reversa do código executável, adulteração da implementação do software, injeção de falhas em tempo de execução.

Como a comunicação em um cenário de caixa branca é feita entre entidades que não são mutuamente confiáveis, a simples distribuição de chaves não basta para garantir a segurança. Para resolver este problema, há uma área de estudo da criptografia específica para cenários de caixa branca, denominada criptografia de caixa branca.

2.8 CRIPTOGRAFIA DE CAIXA BRANCA

Criptografia de caixa branca é uma técnica de ofuscação de código que se destina a implementar algoritmos de criptografia de tal forma que, mesmo se um invasor tiver acesso total a sua implementação e a plataforma de execução, seja incapaz de extrair informações sobre a chave (CHOW et al., 2002). Seguindo esta teoria, é possível garantir a autenticidade da comunicação feita entre softwares que criptografam suas mensagens utilizando este tipo de algoritmo.

Segundo Wyseur (2009, p. 37-39), quando detalhes internos de um algoritmo são expostos para um invasor, a última linha de defesa é a maneira como ele é implementado. A estratégia geral de um algoritmo de criptografia de caixa branca deve ser a seguinte:

a) as informações sobre a chave precisam ser espalhadas sobre toda a implementação, forçando um invasor a ter que analisar tudo ao invés de apenas partes específicas;
b) as rotinas que compõem o algoritmo de criptografia devem ser construídos de maneira aleatória.

2.8.1 AES de caixa branca

Em seu artigo, Chow et al (2002) propõem a implementação de um gerador de algoritmos de AES para cenários de caixa branca. A entrada para este gerador é uma chave qualquer normalmente utilizada no AES e a saída é um algoritmo de AES customizado especificamente para a chave de entrada. O algoritmo é montado de maneira aleatória e a chave é espalhada sobre toda a implementação. Ao fim, o algoritmo gerado executa as mesmas funções do AES comum. Desta forma, torna-se muito mais difícil para um invasor conseguir descobrir a chave utilizada na criptografia de dados.

A idéia básica do AES de caixa branca é de que as transformações que o AES faz nos blocos de dados (através das operações de criptografia) podem ser convertidas em tabelas de consulta (PLASMANS, 2005, p. 28). Uma tabela de consulta é aquela que mapeia todos os possíveis dados de entrada de um algoritmo para os resultados correspondentes. Desta forma, para receber o resultado, basta consultar na tabela através dos dados de entrada, não precisando que o algoritmo calcule-o novamente. Por exemplo, dado um algoritmo cuja entrada é um byte e a saída é o byte de entrada multiplicado por ele mesmo. Uma tabela de consulta para este algoritmo teria 256 posições, pois a entrada pode variar de 0 até 255. Para gerar esta tabela, o algoritmo teria que ser executado 256 vezes, ligando cada resultado obtido com a entrada correspondente na tabela.

De acordo com Chow et al (2002, p. 5-10) e Plasmans (2005, p. 28-35), o AES de caixa branca é composto por uma rede de tabelas de consulta (representando as operações de criptografia), que são ofuscadas através de codificações geradas aleatoriamente (o que torna a montagem do algoritmo aleatória). A única operação de criptografia que continua sendo executada da mesma forma é a `ShiftRows`. A seguir são apresentados os passos para adaptar um algoritmo de AES para um AES de caixa branca (apenas para chaves de 128 bits).

2.8.1.1 Escondendo a chave com as `S-Boxes`

Segundo com Chow et al (2002, p. 5-10) e Plasmans (2005, p. 28-35), o único passo do AES que utiliza a chave é o `AddRoundKey`. Por isso para esconder a chave juntamente com as `S-Boxes`, é necessário compor tabelas de consulta misturando os passos e `AddRoundKey` e `SubBytes`. Com uma chave fixa de 128 bits, o algoritmo terá 10 rodadas. Como a operação

AddRoundKey varia para cada rodada (devido as sub-chaves) será preciso compor tabelas diferentes para cada rodada. Em cada rodada, um bloco de dados de 16 bytes (matriz de estado) sofre estas operações e cada byte possui 256 estados diferentes. Por isso é necessário 10 x 16 = 160 tabelas de consulta com 256 posições cada. O cálculo que deve ser utilizado para compor as tabelas de consulta pode ser visto no Quadro 7.

Para a rodada 1 até 9: $T^r_{i,j}(x) = S(x \oplus k^{r-1}_{i,j})$	Onde: • ⊕ é ou exclusivo (XOR) • T é a tabela de consulta; • r é a rodada; • i é a linha da matriz de estado; • j é a coluna da matriz de estado; • x é a posição da tabela de consulta (varia de 0 até 255); • S é a operação SubByte; • k é a sub-chave; • sr refere-se a nova posição da célula (linha, coluna) depois da etapa de ShiftRows.
Para a rodada 10: $T^{10}_{i,j}(x) = S(x \oplus k^{9}_{i,j}) \oplus k^{10}_{sr(ij)}$	

Fonte: Plasmans (2005, p. 30).

Quadro 7 – Cálculo das operações SubBytes e AddRoundKey em tabelas de consulta

Depois desta etapa, a rotina de encriptação do AES de caixa branca fica da forma mostrada no Quadro 8.

```
para rodada = 1 até NUMERO_DE_RODADAS-1 {
      ShiftRows()
      aplicaTabelaConsultaSubBytesERoundKey(rodada)
      MixColumns()
}
ShiftRows()
aplicaTabelaConsultaSubBytesERoundKey(10)
```

Quadro 8 – Passos da encriptação do AES de caixa branca na primeira etapa da adaptação

2.8.1.2 Convertendo a operação MixColumns em tabelas de consulta

De acordo com Chow et al (2002, p. 5-10) e Plasmans (2005, p. 28-35), a operação MixColumns opera sobre uma coluna da matriz de estado de cada vez. Cada coluna é

composta por 4 bytes e o resultado são os novos 4 bytes da coluna. Se isto for representado diretamente em uma tabela de consulta, haveriam 256 x 256 x 256 x 256 = 4.294.967.296 possibilidades de entrada. Como a saída seriam 4 bytes, a tabela de consulta teria 4 x 4.294.967.296 = 17.179.869.184 bytes de tamanho, ou seja, 16 GB. Manter uma tabela deste tamanho é completamente inviável. Para contornar este problema, pode-se separar esta etapa em 4 tabelas de consulta, uma para cada coluna da matriz utilizada na operação `MixColumns` (vide Quadro 4). Cada uma das 4 tabelas é composta por outras 4 tabelas, pois cada coluna da matriz possui 4 bytes. Desta forma, as tabelas ocuparão apenas 4 x 4 x 256 = 4.096 bytes de tamanho, ou seja, 4 KB.

A nova operação de `MixColumns` não se difere muito da original. Ainda são feitas as operações de soma para gerar cada novo byte de cada coluna, como se estivessem multiplicando matrizes. A diferença é os dados vem diretamente das tabelas de consultas. Depois desta etapa, a rotina de encriptação do AES de caixa branca fica da forma mostrada no Quadro 9.

```
para rodada = 1 até NUMERO_DE_RODADAS-1 {
      ShiftRows()
      aplicaTabelaConsultaSubBytesERoundKey(rodada)
      aplicaTabelaConsultaMixColumns()
}
ShiftRows()
aplicaTabelaConsultaSubBytesERoundKey(10)
```

Quadro 9 – Passos da encriptação do AES de caixa branca na segunda etapa da adaptação

2.8.1.3 Integrando as tabelas de consulta

De acordo com Chow et al (2002, p. 5-10) e Plasmans (2005, p. 28-35), as operações de `AddRoundKey` e `SubBytes` são compostas em uma rede de tabelas de consulta e a operação `MixColumns` em outra. Integrar estas duas redes em uma só fará com chave torne-se ainda mais oculta.

Para realizar a integração, cria-se uma nova rede de tabelas de consulta, fazendo com que as operações de `AddRoundKey` e `SubBytes` façam parte da operação `MixColumns`. Isto significa que cada byte de cada rodada precisará ter os 4 resultados correspondentes que são necessários para calcular a operação `MixColumns`. Como a última rodada não sofre o `MixColumns`, não faz sentido manter as tabelas de consulta dessa rodada na nova rede. Por

isso, ainda deverá existir uma rede de tabelas de consulta com as operações `AddRoundKey` e `SubBytes` específica para a última rodada. Como a nova rede vai mapear apenas 9 rodadas, ela terá 9 x 16 x 4 = 576 tabelas de consulta. Depois desta etapa, a rotina de encriptação do AES de caixa branca fica da forma mostrada no Quadro 10.

```
para rodada = 1 até NUMERO_DE_RODADAS-1 {
      ShiftRows()
      aplicaTabelaConsultaSubBytesERoundKeyEMixColumns(rodada)
}
ShiftRows()
aplicaTabelaConsultaUltimaRodada()
```

Quadro 10 – Passos da encriptação do AES de caixa branca na terceira etapa da adaptação

2.8.1.4 Inserindo codificações geradas aleatoriamente

Segundo com Chow et al (2002, p. 5-10) e Plasmans (2005, p. 28-35), neste ponto, apesar da chave estar aparentemente bem escondida no meio das tabelas de consulta, um invasor ainda pode ler a memória do algoritmo durante sua execução para tentar descobrir a chave. Para dificultar ainda mais este tipo de invasão, as tabelas de consultas podem ser codificadas logo depois que são geradas. Desta forma, não basta um invasor simplesmente ler a memória, pois a codificação fará com que as tabelas fiquem totalmente aleatórias. Isto porque cada versão do AES de caixa branca gerado poderá ter codificações diferentes. Mas para a codificação não afetar o funcionamento do AES em si, deverá ser inserido uma rotina de decodificação logo depois se consultar dados nas tabelas codificadas. Depois desta etapa, a rotina de encriptação do AES de caixa branca fica da forma mostrada no Quadro 11.

```
para rodada = 1 até NUMERO_DE_RODADAS-1 {
      ShiftRows()
      aplicaTabelaConsultaSubBytesERoundKeyEMixColumnsCODIFICADA(rodada)
      decodifica()
}
ShiftRows()
aplicaTabelaConsultaUltimaRodadaCODIFICADA()
decodifica()
```

Quadro 11 – Passos da encriptação do AES de caixa branca na última etapa da adaptação

2.9 ALGORITMOS DE *HASHING* GERADOS ALEATORIAMENTE

De acordo com Grimen, Midtstraum e Mönch (2006, p. 5-7), algoritmos de *hashing* gerados aleatoriamente funcionam da mesma maneira que algoritmos de *hashing* comuns, a diferença é que eles podem ser produzidos dinamicamente e que a forma como cada um gera o *hash* é diferente dos demais. Ou seja, se dois algoritmos de *hashing* aleatórios forem gerados, o *hash* que eles irão gerar a partir de uma mesma mensagem de entrada será diferente. Mas se a mesma mensagem de entrada for submetida ao mesmo algoritmo, o *hash* será o mesmo.

Algoritmos de *hashing* gerados aleatoriamente podem ser utilizados para garantir que apenas quem possua determinada versão gerada do algoritmo consiga gerar um *hash* corretamente. Por exemplo, um software servidor distribui algoritmos de *hashing* gerados aleatoriamente para os softwares clientes confiáveis e pede que eles produzam o *hash* de determinada mensagem. Os softwares clientes que não possuírem um dos algoritmos gerados não conseguirão gerar o *hash* correto. Desta forma, mesmo um invasor que conheça todos os algoritmos de *hashing* utilizados atualmente, como SHA-1 ou MD5, não conseguirá gerar o *hash* correto. Então de acordo com Krief, M'Barka e Ly (2009, p. 3), algoritmos de *hashing* gerados aleatoriamente podem ser usados como um meio de autenticação entre softwares.

Grimen, Midtstraum e Mönch (2006, p. 7-9) defiram dois modos de geração de algoritmos de *hashing* aleatórios: através de filtros na entrada de dados e da composição de funções.

2.9.1 Geração com filtros na entrada de dados

Esta abordagem se baseia no uso de algoritmos de *hashing* já existentes e conhecidos. A idéia é modificar a entrada de dados destes algoritmos fazendo com que ela passe através de um filtro. Então um gerador deste tipo deve criar aleatoriamente filtros e combiná-los com a implementação de algum algoritmo de *hashing*. Por exemplo, o filtro pode separar os dados de entrada em blocos e alterar a ordem que estes blocos são fornecidos ao algoritmo de *hashing* SHA-1 (GRIMEN; MIDTSTRAUM; MÖNCH, 2006, p. 8).

2.9.2 Geração com composição de funções

Segundo Grimen, Midtstraum e Mönch (2006, p. 8-9), esta abordagem se baseia na criação aleatória de funções de *hashing* a partir do zero. Para isso, é necessário definir uma série de funções que executam operações básicas, como somar determinados bytes, por exemplo, que servirão de base para a montagem das funções de *hashing*. Depois disso, um motor de geração aleatória das funções de *hashing* pode ser criado. Por exemplo, dada as funções básicas `f1`, `f2`, `f3` e `f4`, o motor de geração poderá criar as seguintes funções de *hashing*:

a) `hashing1`: composta por `f1` e `f3`;
b) `hashing2`: composta por `f2`, `f3` e `f4`.

2.10 CONFIABILIDADE DE SOFTWARE REMOTO

Como confiar em um software executado em um ambiente remoto não confiável? Um ambiente não confiável é aquele em que os recursos utilizados pelo software (memória, processador, dispositivos de entra/saída) estão sob o controle de um possível usuário mal-intencionado. Com as ferramentas certas, o usuário pode alterar o comportamento do software para atender seus próprios interesses (KRIEF; M'BARKA; LY, 2009, p. 1).

De acordo com Krief, M'Barka e Ly (2009, p. 2), diversas soluções baseadas em mecanismos de proteção de software já foram propostas para este problema. A primeira linha de proteção seria verificar a integridade do código antes de ser carregado, para garantir que ele não foi modificado. Mas esta abordagem não é o bastante uma vez que o um invasor pode alterar o funcionamento do software em tempo de execução. O melhor seria verificar a integridade do código depois que foi carregado também, de forma contínua.

Entretanto, como muitas destas soluções são *offline*, não garantem que os dados resultantes do software não foram interceptados e alterados. Então além da verificação de integridade, é necessário autenticar o software remoto. Em outras palavras, o servidor precisa ser capaz de autenticar o software que está sendo executado no cliente remoto e verificar sua integridade em tempo de execução.

2.10.1 Monitoramento do software remoto

Segundo Krief, M'Barka e Ly (2009, p. 2-4) e Grimen, Midtstraum e Mönch (2006, p. 5-7), a integridade pode ser garantida por um software verificador, chamado de monitor. Ele é incorporado ao software remoto e passa a intermediar a comunicação entre o software servidor e o software remoto. Assume-se que este monitor é confiável, pois além dele verificar a integridade do software remoto, também verifica sua própria integridade. Além disso, o monitor pode verificar a integridade do ambiente de execução (hardware, chamadas do sistema, tempo de execução) a fim de detectar qualquer tentativa de adulteração.

A autenticação do software remoto pode ser feita de duas formas:

a) através do compartilhamento de uma chave secreta entre o servidor e o monitor. Neste caso, técnicas de criptografia de caixa branca devem ser incorporadas para que os dados sejam criptografados sem que a chave seja revelada;
b) através incorporação no monitor de um algoritmo de *hashing* gerado aleatoriamente pelo servidor.

Contudo, um invasor pode simplesmente se passar pelo software remoto e utilizar o mecanismo de autenticação do monitor para enviar falsas mensagens autênticas ao servidor. Para evitar este tipo ataque, o código do software remoto pode ser acoplado ao monitor por meio de técnicas de ofuscação de código. Desta forma, é mais difícil utilizar o monitor para criptografar mensagens que não tenham vindo do software remoto.

A confiabilidade no software remoto pode então ser alcançada com a validação de sua integridade e do ambiente de execução. A validação pode ser feita diretamente no cliente ou através do servidor:

a) pelo cliente: o monitor armazena as informações corretas que o software remoto e o ambiente de execução devem ter e ele mesmo valida-as. Envia para o servidor apenas um informativo com o status do cliente: válido ou inválido. Assim o servidor sabe se pode continuar confiando em determinado software remoto ou não. O problema de executar a validação diretamente no cliente é que toda a lógica de validação estará exposta para invasores, tornando um ataque muito mais simples;
b) pelo servidor: a lógica de validação fica toda no servidor, cabendo ao monitor apenas coletar as informações sobre o software remoto e o ambiente de execução e enviá-las para o software servidor. O software servidor valida as informações e

decide se pode continuar confiando em determinado software remoto ou não.

De acordo com Krief, M'Barka e Ly (2009, p. 2-4) e Grimen, Midtstraum e Mönch (2006, p. 5-7), independentemente de onde a validação ocorre, a comunicação entre o monitor e o servidor sobre a validação pode ser feita de dois modos diferentes:

a) contínua: o software servidor requere que o monitor envie as mensagens de validação periodicamente, em um canal de comunicação específico para a validação. Este modo separa a comunicação referente a validação da comunicação normal que o software servidor tem com o software remoto, o que pode trazer um bom tempo de resposta para a comunicação normal. Entretanto, a comunicação normal não sofre nenhuma validação, o que pode facilitar a adulteração das mensagens deste canal de comunicação;
b) reativa: o software servidor requere que o monitor anexe a validação juntamente com as mensagens de resposta que for receber do software remoto. Desta forma é possível validar além da integridade do software remoto e do ambiente, a integridade da resposta. Entretanto, o tempo de resposta da comunicação entre o software servidor e o software remoto pode ser prejudicado.

Os mecanismos apresentados fazem com que o software remoto seja mais difícil de adulterar, mas não impossível. Com tempo suficiente, um invasor pode quebrar as várias proteções propostas. Para dificultar ainda mais a adulteração, é possível utilizar técnicas de atualização dinâmica de código. O servidor deve periodicamente substituir parte do código executável do monitor enquanto ele está executando. A substituição de código tem que ser feita de forma que o novo monitor não possa ser quebrado de maneira automática por um invasor que quebrou uma versão anterior do monitor. Isto pode ser feito de maneira eficaz se os seguintes conceitos forem adotados:

a) realocação: mudar certas partes do código para outros lugares da memória, de maneira aleatória. Com isso, é necessário também atualizar as partes do código que não foram alteradas que referenciam a parte alterada, para evitar mudanças no fluxo das instruções;
b) diversificação: trocar um conjunto de instruções que faz determinada tarefa por outro conjunto de instruções que faz a mesma tarefa, mas com instruções diferentes.

Desta forma, um invasor não terá tempo suficiente para quebrar cada nova versão do monitor para manter sua adulteração.

Com isso, quatro técnicas de proteção são combinadas para proporcionar um nível de

segurança praticável para computação remota:

a) ofuscação de código;
b) monitoramento da execução do software;
c) criptografia de caixa branca ou algoritmos de *hashing* gerados aleatoriamente;
d) atualização dinâmica de código.

2.11 PROTEÇÃO DE CÓDIGO JAVA

Classes Java são fáceis de fazer engenharia reversa. As vantagens que permitem que aplicações Java rodem em qualquer lugar fazem a tradução reversa ser simples. Existe um número considerável de decompiladores Java no mercado que produzem um código incrivelmente legível (Excelsior, 2010).

Segundo Leskov (2010), a primeira solução que vem a mente para proteger o código Java é a encriptação de classes. No entanto, esta idéia é falha, por que a Java Virtual Machine (JVM) não permite carregar de maneira nativa as classes encriptadas. Então se torna bastante fácil interceptar a decriptação das classes logo antes de serem carregadas.

De acordo com Leskov (2010), outra idéia é deixar o código um pouco menos compreensível é através da ofuscação. Um código ofuscado deve produzir o mesmo resultado ao rodar na JVM, mas ele torna a decompilação muito mais difícil e as vezes até impede o funcionamento de alguns decompiladores. Existem alguns tipos de ofuscações que podem ser feitas em código Java:

a) ofuscação de nome: substituí identificadores como nome de classes e atributos, por caracteres ou palavras sem sentido. Por exemplo, de `com.mycompany.TradeSystem.Security.checkFingerPrint()` para `a.a0();`
b) encriptação de *String*: substitui literais *Strings* por uma chamada a um método que decripta a *String* encriptada passada como parâmetro, retornando a *String* original;
c) ofuscação de fluxo: modifica o fluxo normal do código para que torne impossível de se decompilar em um código Java bem estruturado ou pelo menos torne-o mais difícil de se entender.

Um exemplo de ofuscador de código é o Java *ByteCode Obfuscator* (JBCO), desenvolvido pelo grupo Sable, da universidade de McGill, no Canadá. Segundo Sable (2010), o JBCO transforma o código Java em algo mais complexo, esotérico ou obscuro a fim

de dificultar a engenharia reversa e ataques de decompilação.

2.12 ATUALIZAÇÃO DINÂMICA DE CÓDIGO JAVA

De acordo com Stadler, Wimmer e Würthinger (2010, p. 1), atualização dinâmica de código é uma técnica para alterar as instruções de um programa enquanto ele está executando. Para programas que rodam em máquinas virtuais, as possibilidades de atualização dinâmica de código são maiores por causa da camada adicional entre a execução do programa e o hardware. A habilidade de atualizar o código de um programa em execução traz vantagens para diversas áreas, como por exemplo:

a) *debugging*: geralmente um desenvolvedor faz pequenas alterações no código enquanto está fazendo o *debug* de alguma aplicação. Não ser obrigado a reiniciar a aplicação a cada uma destas pequenas alterações melhora muito a produtividade;
b) aplicações críticas: existem muitas aplicações que não podem ser paradas pois fornecem um serviço vital para muitas pessoas. Apesar disto, existirão atualizações que o aplicativo deverá sofrer. Com a atualização dinâmica de código este aplicativo poderá ser atualizado sem a necessidade de ser parado antes.

Segundo Yang (2006), o Java dispõe mecanismos para fazer a atualização dinâmica de código de uma maneira simples. A atualização deve seguir quatro passos:

a) obter o novo conteúdo da classe Java que se deseja atualizar;
b) compilar a classe Java em tempo de execução;
c) recarregar a classe Java em tempo de execução;
d) ligar a classe atualizada a quem deseja utilizá-la.

Este processo pode ser feito através do carregador de classes que o Java disponibiliza, a classe `java.lang.ClassLoader`.

2.13 TRABALHOS CORRELATOS

Existem várias ferramentas de segurança para jogos *online*, sendo que as mais utilizadas são as comerciais. Elas apresentam funcionalidades parecidas e por isso a

concorrência é acirrada. As seguintes ferramentas são analisadas: GameGuard (NPROTECT, 2010), HackShield (AHNLAB, 2009) e GameFort (GAMEFORT, 2010). Estas três ferramentas possuem mecanismos para bloqueio de clientes não oficiais, mas é de conhecimento que o GameGuard e o GameFort já tiveram este bloqueio burlado.

O GameGuard é uma ferramenta coreana utilizada em sua maioria em jogos de origem asiática. Segundo o nProtect (2010), sua principal funcionalidade no servidor é o bloqueio de *hacks* em geral. No cliente, monitora a memória para prevenir alterações ilegais, esconde o processo do jogo, finaliza qualquer aplicação suspeita e bloqueia determinadas chamadas ao DirectX e a funções do Windows. Sua proteção contra clientes não oficiais baseia-se em obrigar que as mensagens trocadas entre o cliente e o servidor passem por um software do GameGuard que roda no cliente junto com o jogo. Para burlar esta proteção foi necessário apenas fazer com que o cliente não oficial também passasse suas mensagens através deste software.

O HackShield assim como o GameGuard também é de origem coreana e com forte atuação nos jogos asiáticos. Segundo o AhnLab (2009), sua principal funcionalidade no servidor também é a prevenção contra *hacks* em geral. No cliente, ele protege a memória, bloqueia qualquer possível análise no código do executável do jogo, finaliza qualquer aplicação suspeita e bloqueia instruções de programas externos que movem o cursor do *mouse* automaticamente.

O GameFort é uma ferramenta de segurança brasileira que é um pouco mais simples que as suas concorrentes coreanas. Segundo o GameFort (2010), ela não garante proteção contra *hacks* em geral, apenas contra ataques mais simples como um *Distributed Denial of Service* (DDoS)[3]. Seu bloqueio contra clientes não oficiais se baseia em um algoritmo de criptografia de chave simétrica. O cliente tem que encriptar determinado pacote usando uma chave fixa para poder se conectar no servidor. Para burlar esta proteção, foi necessário apenas descobrir qual era o algoritmo de criptografia utilizado e aonde a chave fixa estava escondida. Desta forma, o cliente não oficial também consegue encriptar o pacote e se conectar no servidor.

[3] DDoS é um ataque de negação de serviço ampliado, ou seja, que utiliza até milhares de computadores para atacar uma determinada máquina. Ataques de negação de serviços consistem em tentativas de impedir usuários legítimos de utilizarem um determinado serviço de um computador. Para isso, são usadas técnicas que podem: sobrecarregar uma rede a tal ponto em que os verdadeiros usuários dela não consigam usá-la; derrubar uma conexão entre dois ou mais computadores; fazer tantas requisições a um *site* até que este não consiga mais ser acessado; negar acesso a um sistema ou a determinados usuários (ALECRIM, 2004).

3 DESENVOLVIMENTO

Neste capítulo são apresentados os requisitos, a especificação e o detalhamento da implementação do sistema. Ao final sua operacionalidade é demonstrada.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Seguem os requisitos do sistema de proteção:

a) o software cliente só pode ser executado e mostrado ao usuário depois que o conteúdo de seu arquivo executável e o estado do ambiente de execução forem validados pelo sistema de proteção (Requisito Funcional – RF);
b) o software servidor deve receber apenas conexões de softwares clientes validados pelo sistema de proteção (RF);
c) o sistema de proteção deve permitir que sejam feitas validações do estado do ambiente de execução e do software cliente durante sua execução (RF);
d) o software servidor deve ser desconectado de softwares clientes que se tornarem inválidos (RF);
e) as validações devem ser feitas automaticamente pelo sistema de proteção, sem a necessidade de interação com o usuário do software cliente (RF);
f) a comunicação feita pelo sistema de proteção referente a validação deve ser confidencial (Requisito Não Funcional – RNF)
g) as validações feitas pelo sistema de proteção não devem poder ser reproduzidas de forma automatizada por outro software (RNF);
h) o sistema de proteção deve ser compatível com qualquer sistema que utilize a arquitetura cliente/servidor e possua um software cliente próprio (RNF);
i) o sistema de proteção deve ser implementado utilizando a linguagem de programação Java (RNF).

3.2 ESPECIFICAÇÃO

A especificação do sistema foi feita através de diagramas *Unified Model Language* (UML), como o de casos de uso, de classes, de implantação e de atividades. Para auxiliar na construção da especificação foi utilizado a ferramenta Enterprise Architect.

3.2.1 Diagrama de Casos de Uso

A seguir são apresentados os diagramas de casos de uso, que estão divididos em dois pacotes: cliente e servidor.

Na Figura 3 é exibido o primeiro pacote, que apresenta os casos de uso referentes ao software servidor.

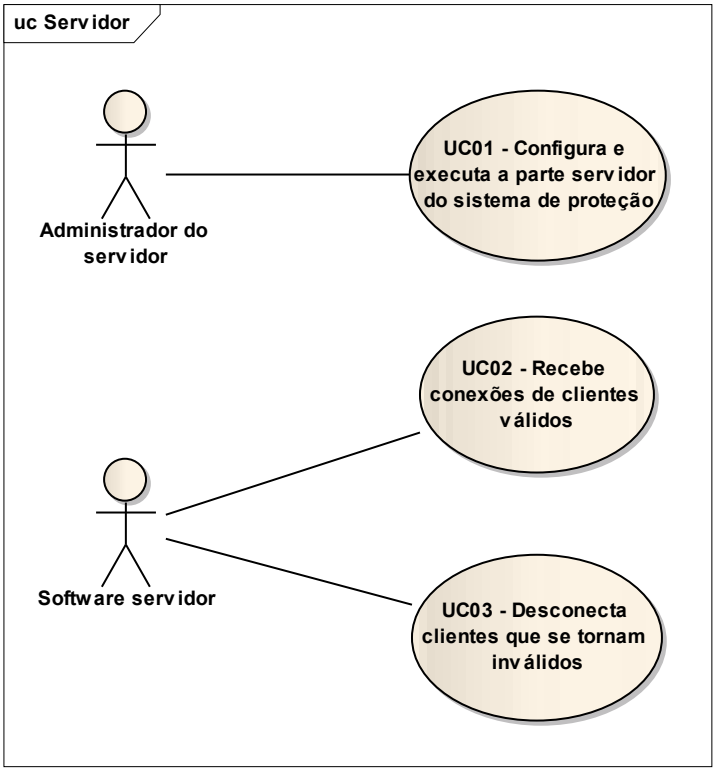


Figura 3 – Casos de uso do servidor

Nos quadros 12, 13 e 14 são mostradas as descrições dos casos de uso do pacote servidor.

UC01 – Configura e executa a parte servidor do sistema de proteção	
Cenário principal	1) configura a parte servidor do sistema de proteção 2) executa a parte servidor do sistema de proteção
Pós-condições	Parte servidor do sistema de proteção em execução.

Quadro 12 – Descrição do caso de uso `Configura e executa a parte servidor do sistema de proteção`

UC02 – Recebe conexões de clientes válidos	
Pré-condições	A parte servidor do sistema de proteção e o software servidor devem estar em execução.
Cenário principal	1) o sistema de proteção valida um software cliente e seu ambiente de execução. 2) software servidor recebe uma conexão vinda do sistema de proteção, que representa a conexão do software cliente válido.
Cenário alternativo	No passo 1, o software cliente ou o ambiente de execução não são validados: 1) software servidor não recebe a conexão.
Pós-condições	Um canal de comunicação entre o software servidor e o software cliente válido é aberto.

Quadro 13 – Descrição do caso de uso `Recebe conexões de clientes válidos`

UC03 – Desconecta clientes que se tornam inválidos	
Pré-condições	O software servidor deve ter um canal de comunicação aberto com um software cliente.
Cenário principal	1) as instruções do software cliente são alteradas ou o ambiente de execução sofre uma mudança inválida 2) o sistema de proteção detecta alteração e invalida o software cliente 3) a conexão que representava o cliente inválido entre o software servidor e o sistema de proteção é terminada
Pós-condições	O canal de comunicação entre o software servidor e o software cliente é fechado.

Quadro 14 – Descrição do caso de uso `Desconecta clientes que se tornam inválidos`

A Figura 4 mostra o segundo pacote, referente aos casos de uso do cliente.

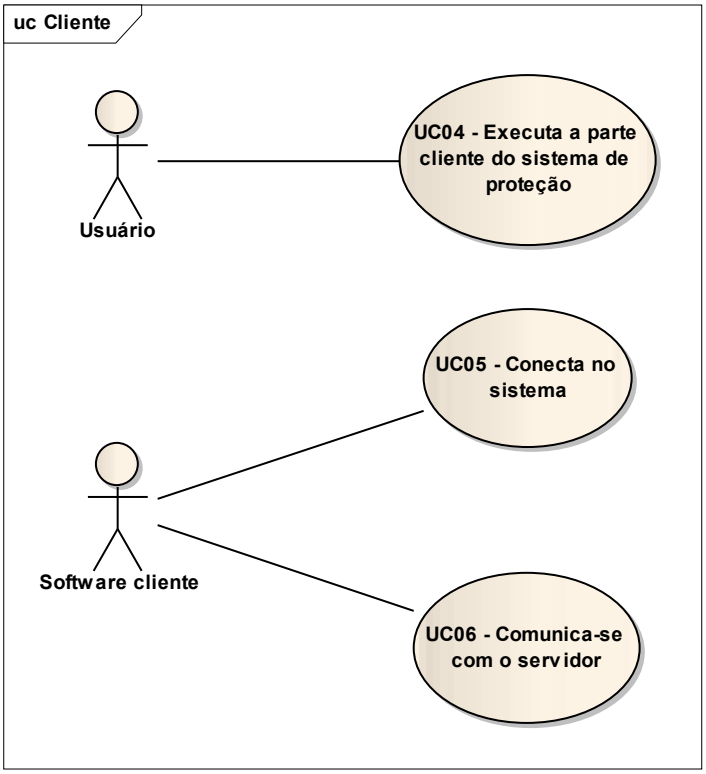


Figura 4 – Casos de uso do cliente

Nos quadros 15, 16 e 17 são mostradas as descrições dos casos de uso do pacote cliente.

UC04 – Executa a parte cliente do sistema de proteção	
Cenário principal	1) a parte cliente do sistema de proteção é executado no computador cliente. 2) o sistema de proteção valida o software cliente e seu ambiente de execução. 3) a parte cliente do sistema de proteção executa o software cliente.
Cenário alternativo	No passo 2, o software cliente ou o ambiente de execução não são validados 1) o software cliente não é executado. 2) a parte cliente do sistema de proteção é finalizada.
Pós-condições	Software cliente em execução.

Quadro 15 – Descrição do caso de uso `Executa a parte cliente do sistema de proteção`

UC05 – Conecta no sistema	
Pré-condições	O software cliente deve estar em execução.
Cenário principal	1) o software cliente se conecta no sistema de proteção.
Pós-condições	Um canal de comunicação entre o software cliente e o software servidor é aberto.

Quadro 16 – Descrição do caso de uso `Conecta no sistema`

UC06 – Comunica-se com o servidor	
Pré-condições	O software cliente deve ter um canal de comunicação aberto com um software servidor.
Cenário principal	1) o software cliente se comunica com o software servidor através da conexão feita com o sistema de proteção.
Cenário alternativo	No passo 1, as instruções do software cliente são alteradas ou o ambiente de execução sofre uma mudança inválida 1) o sistema de proteção detecta alteração e invalida o software cliente 2) a conexão entre o software cliente e o sistema de proteção é terminada e o software cliente é fechado.
Pós-condições	O canal de comunicação entre o software cliente e o software servidor é fechado.

Quadro 17 – Descrição do caso de uso `Comunica-se com o servidor`

3.2.2 Arquitetura do sistema

Esta seção apresenta a arquitetura do sistema de proteção desenvolvido.

O sistema é composto por dois componentes: o monitor, que roda no computador cliente e o validador, que roda no computador servidor. A Figura 5 demonstra esta estrutura. Note que o software cliente e o software servidor não possuem um canal de comunicação direto. Com o sistema de proteção, eles passam a se comunicar através de seus representantes, o monitor e o validador. Isto faz parte da proteção oferecida pelo sistema, para que o software servidor só se comunique com softwares clientes válidos.

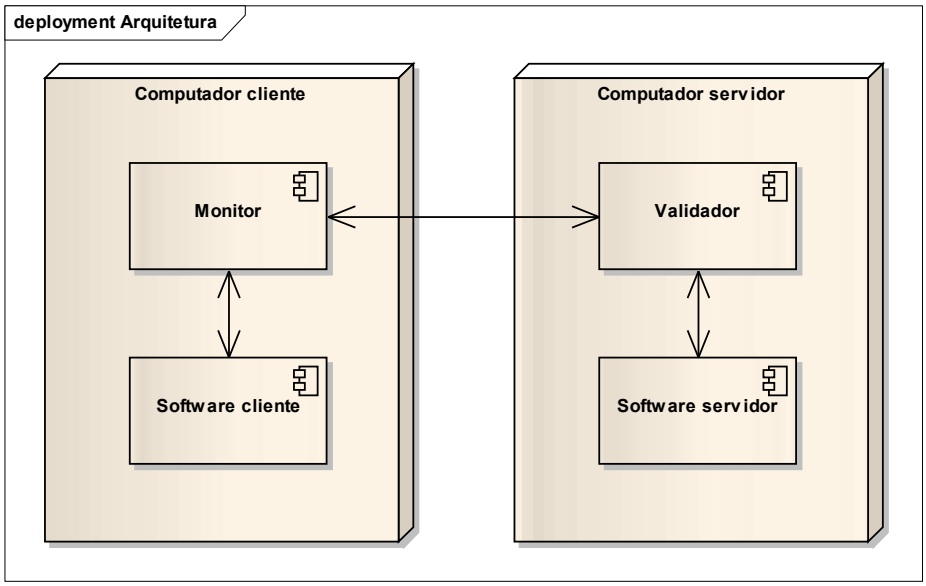


Figura 5 – Arquitetura do sistema

Os componentes envolvidos na arquitetura do sistema são detalhados a seguir. Posteriormente, são descritos os detalhes arquiteturais de partes do sistema, como a comunicação, validações e proteção do código do monitor.

3.2.2.1 Componente Validador

O validador é o principal componente do sistema, pois é ele quem toma todas as decisões e repassa para todos os monitores que estão ligados a ele. Além disso, é ele quem está ligado diretamente ao software servidor, que é software que o sistema desenvolvido visa proteger. O componente validador possui as seguintes atribuições:

a) aceitar conexões dos componentes monitores;
b) validar se os monitores conectados são autênticos (que não foram adulterados e que não são outros softwares além do monitor disponibilizado pelo sistema);
c) gerenciar cada componente monitor conectado, passando-lhe ordens com o que ele deve fazer;
d) receber as informações coletadas pelos monitores;
e) validar os softwares clientes;
f) gerar atualizações de código para os monitores;
g) gerenciar as conexões com o software servidor.

Para que o validador consiga proteger o software servidor de maneira eficaz, ele deve ser o único software com acesso aos meios de comunicação que software servidor disponibiliza para os softwares clientes. As portas abertas no computador servidor que proviam acesso aos softwares clientes antes da utilização do sistema de proteção devem passar a ser usadas pelo validador. Desta forma, o software servidor deverá ter outras portas abertas para a comunicação com os softwares clientes. Estas portas devem ser de uso exclusivo do validador. Isto pode ser feito, por exemplo, bloqueando o acesso externo a estas portas, já que o validador deve estar rodando no mesmo computador ou pelo menos na mesma rede local do software servidor. Assim, softwares externos não conseguirão se comunicar diretamente com o software servidor por estas portas.

3.2.2.2 Componente Monitor

O monitor é o componente que será executado no ambiente remoto não confiável. Por isso ele não possui quase nenhuma lógica que possa expor o funcionamento do sistema para invasores. Ele está preparado basicamente para apenas receber e executar ordens vindas do componente validador. Então a partir das ordens do validador, suas atribuições podem ser as seguintes:

a) coletar informações sobre o ambiente de execução, arquivo executável do software cliente e do estado do software cliente em execução;
b) enviar as informações coletadas para o validador;
c) integrar as atualizações de código recebidas;
d) executar o software cliente;
e) aceitar conexões do software cliente;
f) finalizar o software cliente.

Quando um usuário do software cliente quiser utilizá-lo, ele deve na verdade abrir o monitor. Isto porque o monitor sabe como se comunicar com o validador, que é o único meio de acesso ao software servidor. Para isso, a distribuidora do software cliente deve:

a) adicionar o monitor no instalador do software cliente;
b) fazer com que todos os atalhos de acesso ao software cliente apontem para o monitor;
c) alterar o software cliente (que originalmente tentaria conectar-se no software servidor remoto) para que se conecte no monitor local.

3.2.2.3 Comunicação

O validador possui uma linha exclusiva de comunicação com cada monitor para mandar as ordens e obter os dados dos monitoramentos. A comunicação que o software servidor normalmente teria com os softwares clientes é feita através de outras linhas que, por sua vez, são exclusivas para este fim. Desta forma, o validador terá `n + 1` linhas de comunicação com cada monitor, onde `n` é o número de linhas de comunicação que originalmente o software servidor teria com cada software cliente.

A linha de comunicação do monitoramento utiliza o protocolo de segurança SSL, se

tornando protegida contra ataques de caixa preta. As demais linhas de comunicação não serão protegidas pelo sistema, pois seu objetivo é influenciar o mínimo possível na comunicação e no tempo de resposta original entre o software servidor e os softwares clientes.

3.2.2.4 Validações

As validações têm o objetivo de validar a autenticidade do software cliente e do monitor e o estado do ambiente de execução remoto. A autenticidade do software cliente é verificada através de informações obtidas sobre o software cliente antes e durante sua execução. A autenticidade do monitor é verificada através de mensagens criptografadas com algoritmos AES de caixa branca e de valores *hash* obtidos com algoritmos de *hashing*, ambos gerados aleatoriamente pelo validador. Assumindo que apenas um monitor válido entenda os algoritmos gerados pelo validador, é possível validar sua autenticidade.

As validações são desencadeadas pelo validador, que manda ordens para os monitores obterem os dados necessários. Após a validação inicial, que permite a determinado monitor abrir o software cliente, ocorrerão validações periódicas. As validações periódicas ocorrem em intervalos de tempo aleatórios.

A validação inicial é feita a partir do conteúdo do arquivo executável do software cliente e do estado do ambiente de execução. Estas informações são mescladas em um dado só e são submetidos a um algoritmo de *hashing* gerado aleatoriamente pelo validador. O *hash* gerado é enviado ao validador, podendo ou não ser criptografado antes por um algoritmo AES de caixa branca gerado pelo validador, dependendo da ordem inicial dele. Desta forma, não se mantém um padrão na mensagem de validação inicial.

As validações periódicas são feitas a partir de informações sobre a execução do software cliente e do ambiente de execução em si. Estas informações são mescladas em um dado só, podendo ou não ser submetido a um algoritmo de *hashing* gerado aleatoriamente. O dado é então enviado para o validador, podendo ou não ser criptografado antes por um algoritmo AES de caixa branca gerado pelo validador.

3.2.2.5 Proteção do código do monitor

O código executável do monitor possui defesas contra os seguintes tipos de ataques:

a) engenharia reversa: o código é ofuscado utilizando o JBCO;
b) *debug* das instruções para reproduzir os comportamentos do monitor em outro software: partes do código são atualizadas dinamicamente, fazendo com que novos comportamentos sejam introduzidos e outros sejam descartados;
c) leitura das instruções na memória para reproduzir os comportamentos do monitor em outro software: os códigos atualizados dinamicamente no monitor sofrem realocação, evitando que os novos códigos fiquem sempre no mesmo lugar da memória;
d) *debug* a partir do único ponto de leitura dos dados, visando alterar os valores lidos e consequentemente determinados comportamentos: o monitor suporta diversos pontos de leitura concorrentes para o mesmo canal de comunicação, que podem ser remanejados com a atualização dinâmica de código.

A atualização dinâmica de código é feita no início da comunicação com um monitor (para contextualizá-lo) e depois periodicamente, em intervalos de tempo aleatórios.

3.2.3 Diagrama de atividades

Esta seção apresenta os diagramas de atividades que ilustram as principais operações realizadas pelo sistema de proteção.

A Figura 6 demonstra o processo genérico de envio de ordens do validador para o monitor. Depois de enviar a ordem, o validador aguarda um tempo pela resposta do monitor. Se esse tempo passar e não houver resposta, pode se tratar de um cliente que foi adulterado. Neste caso, o monitor torna-se inválido e é desconectado. O tempo que o validador aguarda pela resposta é determinado pelo tipo de ordem enviada.

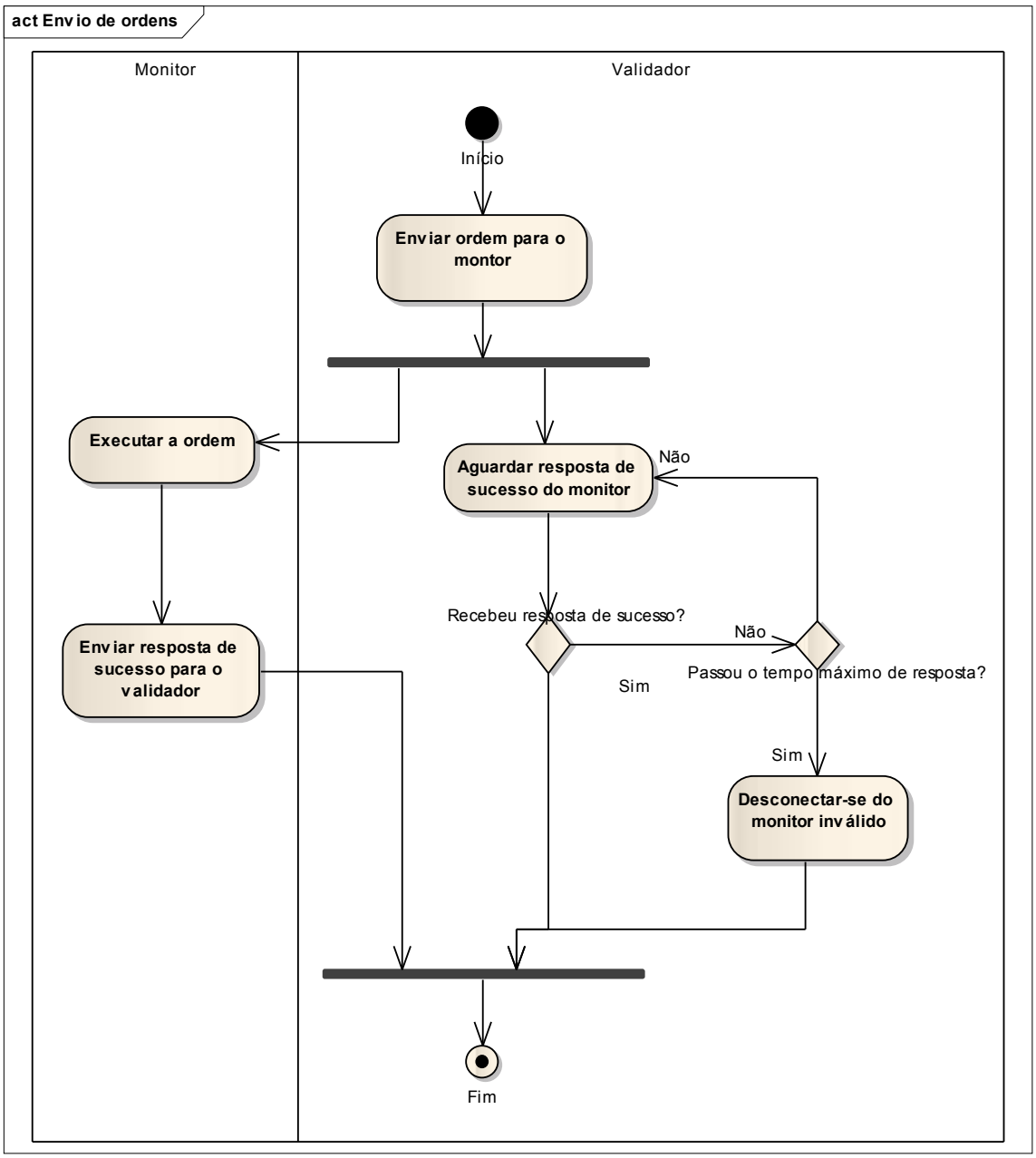


Figura 6 – Diagrama de atividades do envio de ordens

A Figura 7 demonstra o processo de atualização dinâmica de código. Este processo pode ocorrer de maneira paralela com as validações, por isso o validador mantém informações sobre o estado atual de cada monitor conectado a ele. Desta forma, ele sabe qual versão do código (algoritmos de criptografia e *hashing* gerados aleatoriamente, por exemplo) está executando em determinado monitor e assim pode validá-los de maneira correta. Apenas depois que o monitor confirmar a atualização do código é que o validador atualiza o estado daquele monitor. O tempo de espera máximo da ordem de atualização de código é proporcional ao tamanho do novo código.

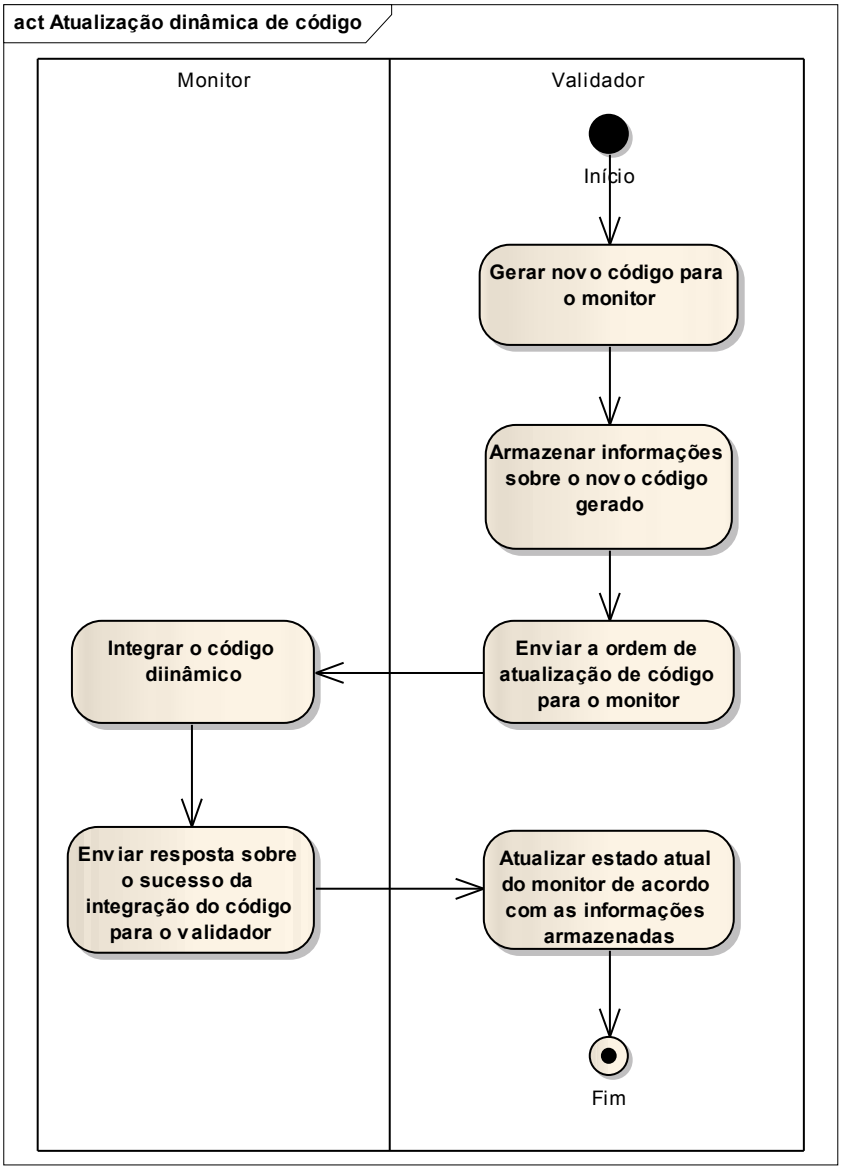


Figura 7 – Diagrama de atividades da atualização dinâmica de código

A Figura 8 demonstra a validação inicial feita entre um novo monitor que tenta se conectar no validador. Esta situação é desencadeada pelo usuário do software cliente, que abriu o monitor a fim de utilizar o software cliente. O processo de atualização dinâmica do código inicial utiliza o mesmo processo mostrado na Figura 7, por isso ele é abstraído da Figura 8.

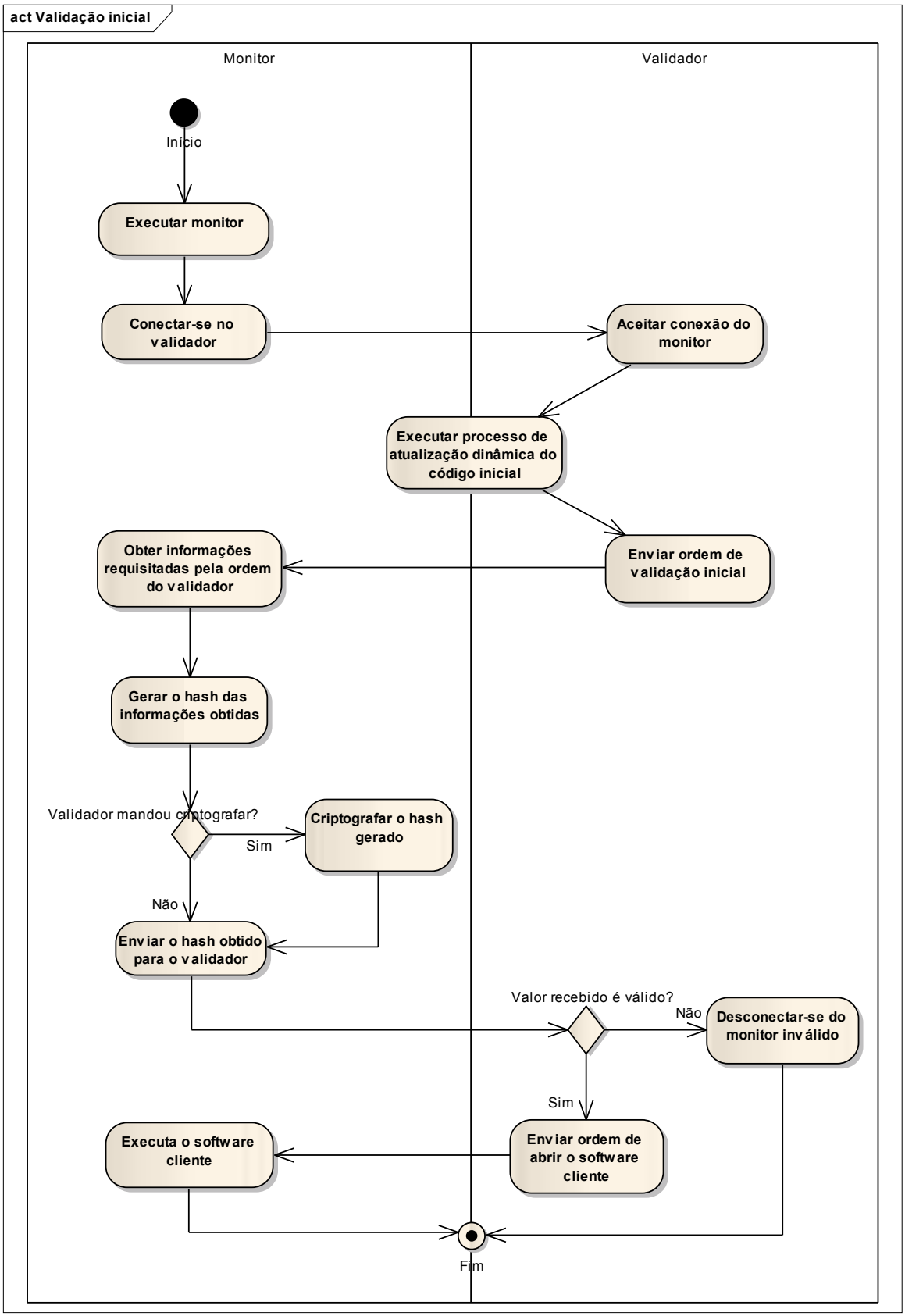


Figura 8 – Diagrama de atividades da validação inicial

A Figura 9 mostra como os canais de comunicação entre o software cliente válido e o servidor são criados e mantidos. O processo de execução das validações periódicas é

detalhado no diagrama da Figura 10, por isso foi abstraído deste diagrama.

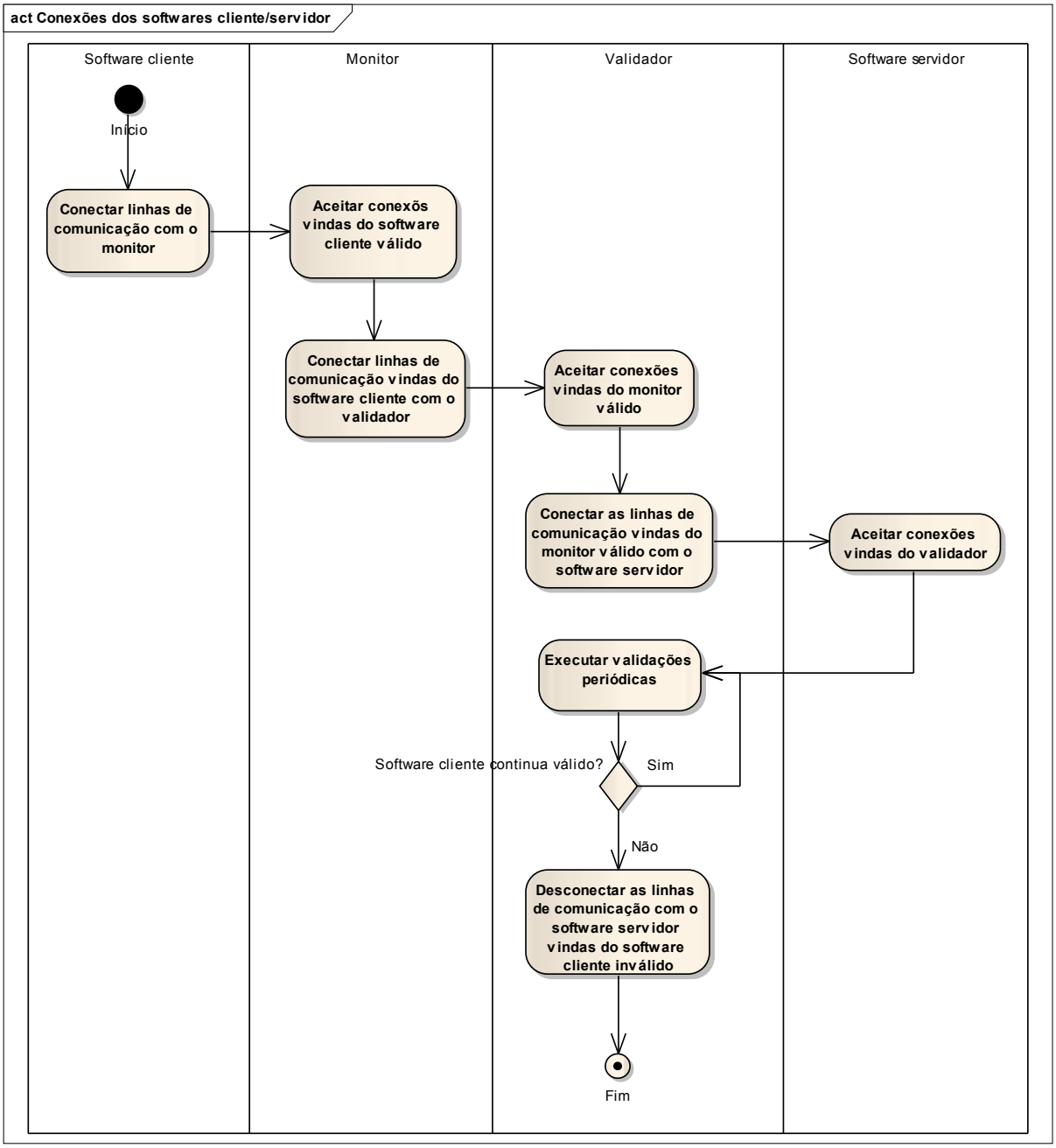


Figura 9 – Diagrama de atividades das conexões dos softwares cliente/servidor

A Figura 10 demonstra o processo de validação periódico, que ocorre em intervalos de tempo aleatórios. Estes intervalos são determinados pelo componente validador.

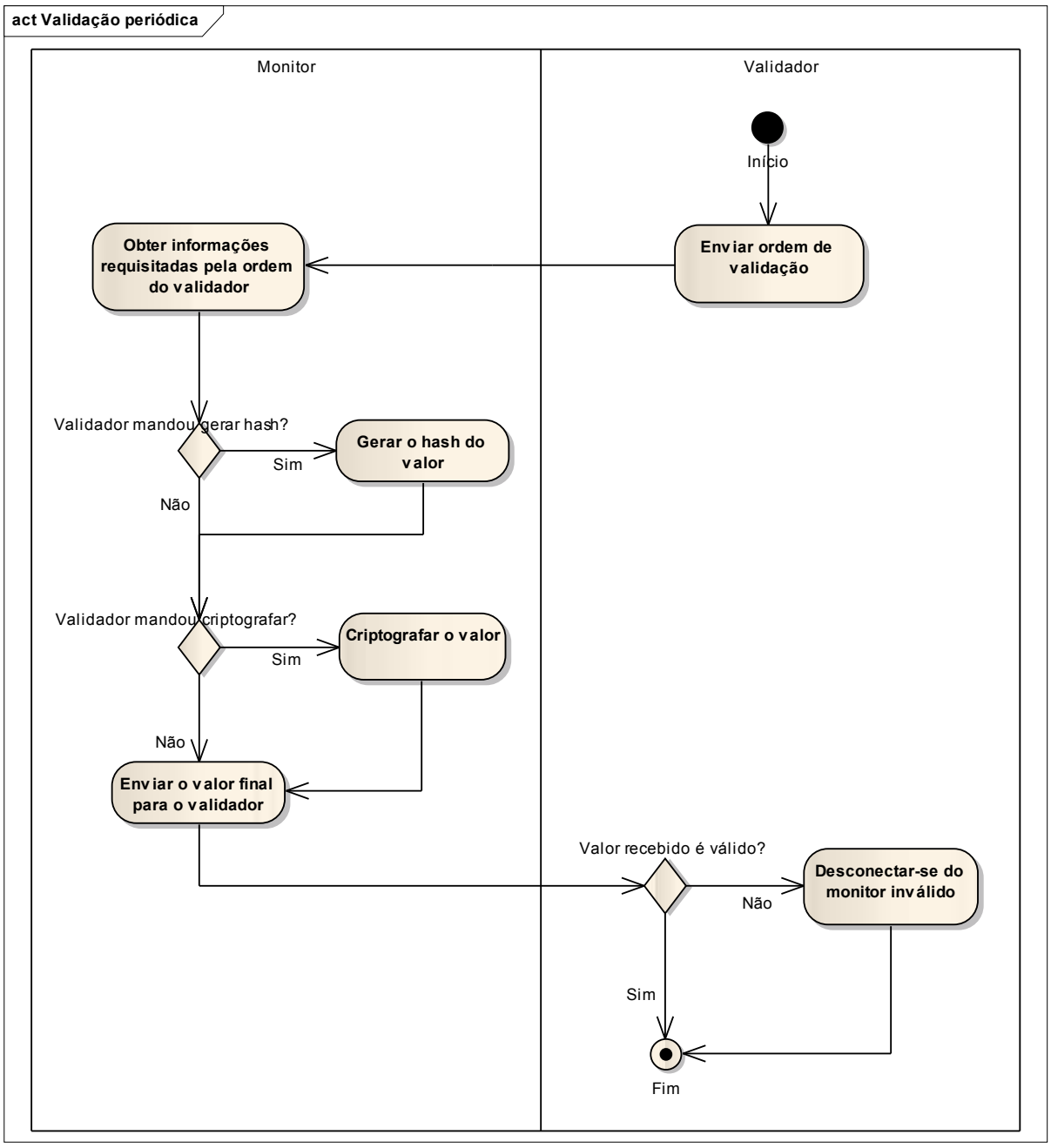


Figura 10 – Diagrama de atividades da validação periódica

3.2.4 Diagrama de classe

Nesta seção são apresentadas as classes utilizadas no desenvolvimento do sistema de proteção. As classes do sistema estão separadas em três partes:

a) básicas: classes que são comuns entre o validador e os monitores;

b) servidor: classes do validador;

c) cliente: classes do monitor.

3.2.4.1 Classes básicas

As classes básicas formam a estrutura base das operações que o monitor realiza sob as ordens do validador. Para que o monitor possua o mínimo possível de lógica própria, foi feita uma estrutura de classes trabalhadoras. Estas classes são geradas pelo validador, compiladas, enviadas para o monitor, carregadas e depois executadas, fazendo o trabalho que o validador lhes atribui. Depois de executados, eles continuam abertos para receberem ordens vindas do servidor. O validador controla todo o seu ciclo de vida, podendo ordenar também sua destruição.

As ordens vêm do canal de comunicação de validação que é compartilhado entre todos os trabalhadores. Por isso eles se organizam em uma fila circular, para que possíveis ordens lidas pelo trabalhador errado possam chegar ao trabalhador correto. Desta forma, a estrutura de trabalhadores serve para atender alguns detalhes arquiteturais:

a) alteração de comportamentos durante a execução do monitor;
b) realocação de código do monitor;
c) possuir vários pontos de leitura concorrentes, devido a maneira como as ordens são lidas.

A estrutura base das classes trabalhadoras pode ser vista na Figura 11. A seguir são descritas as classes envolvidas:

a) `AbstractWorker`: classe base de todo trabalhador. Deve ser rodada como uma *Thread* pelo monitor. Recebe no construtor as linhas de comunicação que o monitor tem com o validador e o software cliente. Já implementa o mecanismo de leitura concorrente de ordens do validador, restando as subclasses apenas implementar a execução das ordens que elas desejam seguir. Implementa ordens básicas como a destruição do trabalhador;
b) `Work`: representa uma ordem ou trabalhado enviado pelo validador. Guarda apenas os valores brutos, lidos diretamente em bytes.

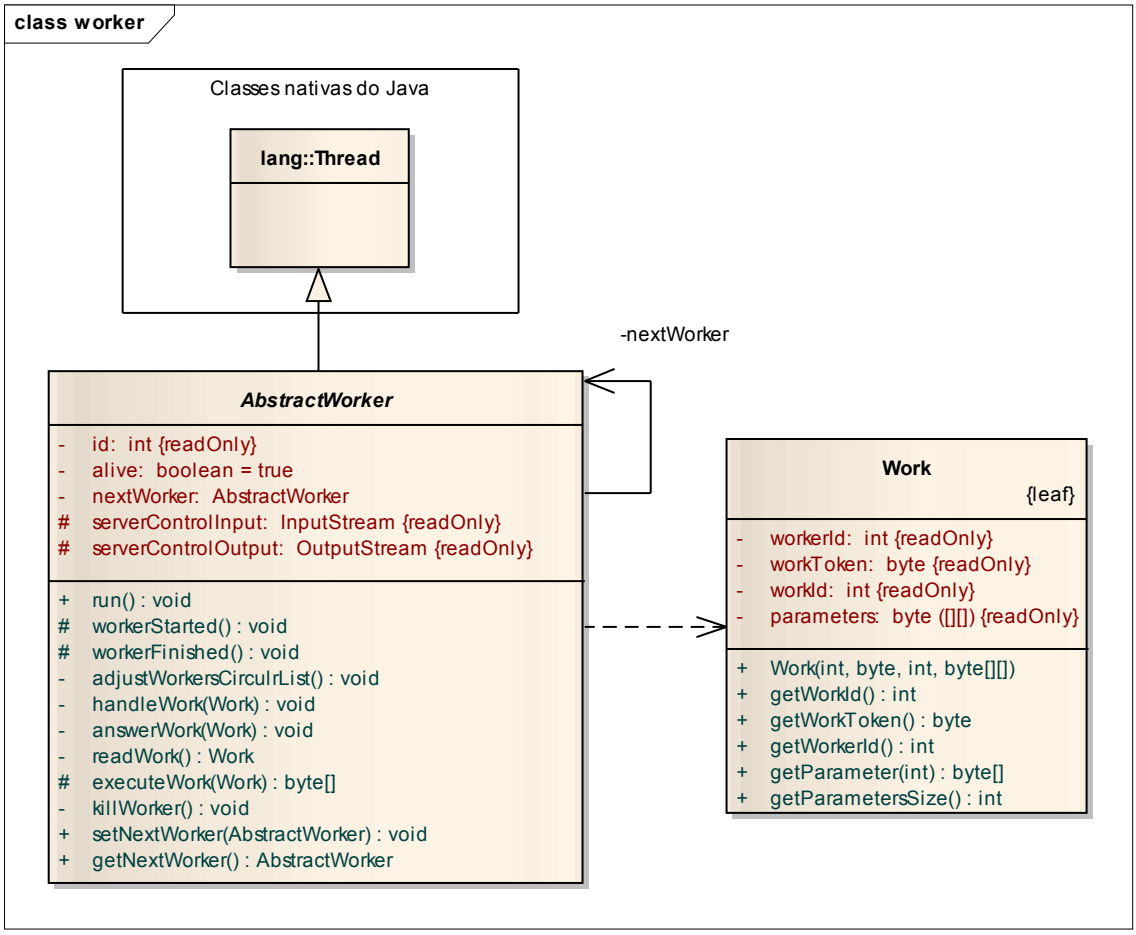


Figura 11 – Diagrama de classes da estrutura base de trabalhadores

A seguir, a Figura 12 mostra a estrutura base das classes de criptografia.

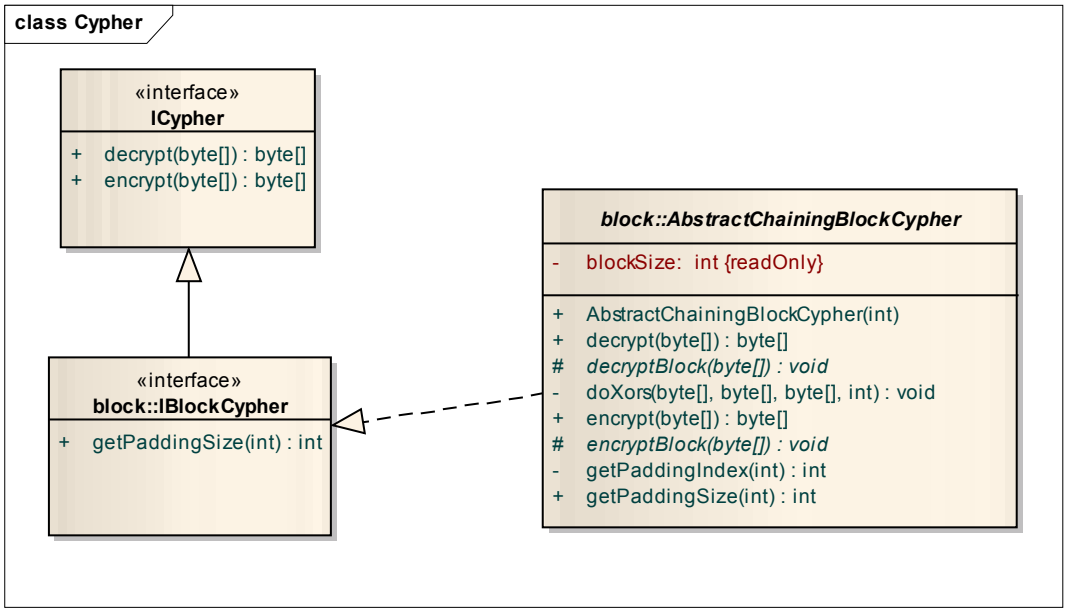


Figura 12 – Diagrama de classes da estrutura base de criptografia

Segue o detalhamento das classes da estrutura base de criptografia:

a) `ICypher`: interface base de todo algoritmo de criptografia do sistema. Define os métodos de criptografia;

b) `IBlockCypher`: interface base de todo algoritmo de criptografia de bloco do sistema. Define o método adicional necessário para se trabalhar com criptografia de bloco;

c) `AbstractChainingBlockCypher`: classe base de todo algoritmo de criptografia de bloco do sistema. Ela implementa a lógica de dividir a entrada em blocos, restando as subclasses apenas criptografar o bloco em si. A criptografia é feita seguindo a idéia do *Cipher Block Chaining* (CBC), que faz com que cada bloco criptografado influencie no próximo, tornando a cifra mais forte.

A Figura 13 mostra a estrutura base de algoritmos de *hashing* do sistema.

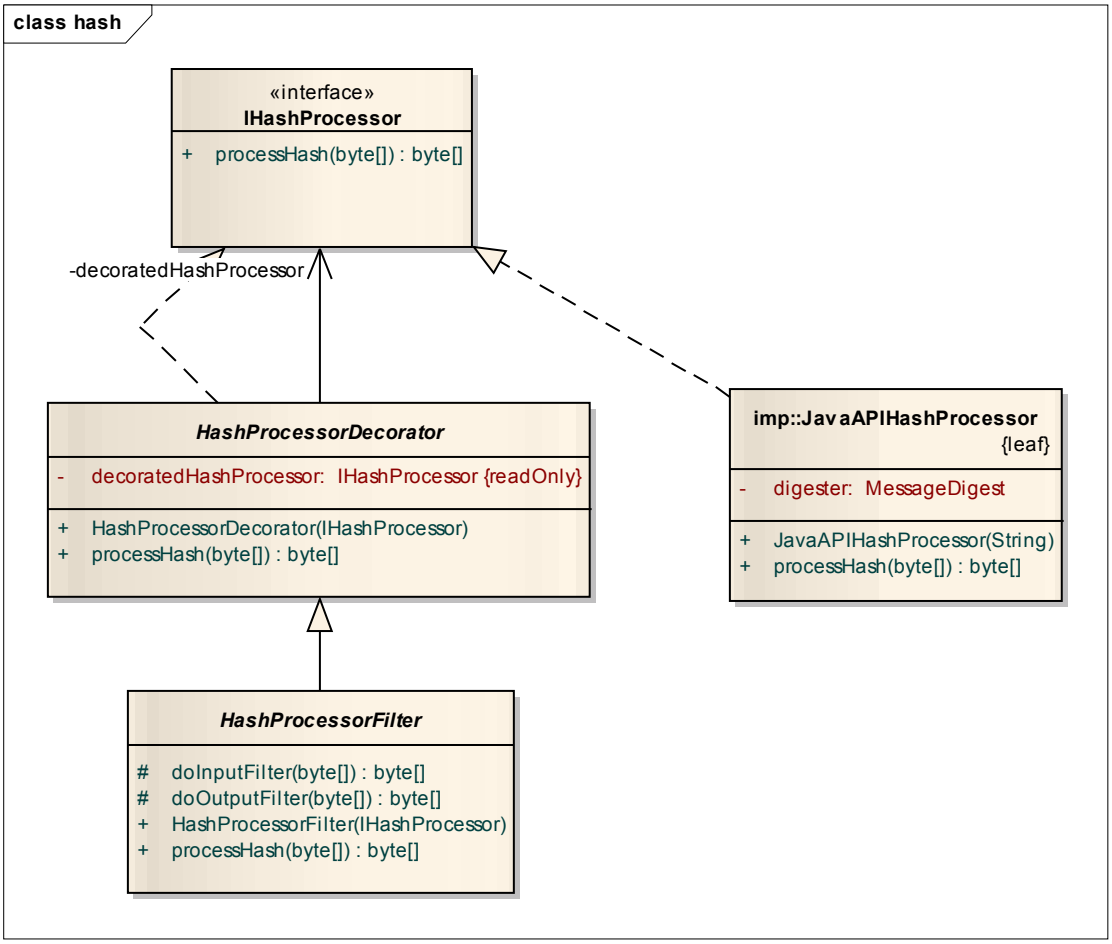


Figura 13 – Diagrama de classes da estrutura base de criptografia

Segue o detalhamento das classes da estrutura base de algoritmos de *hashing*:

a) `IHashProcessor`: interface base de todo algoritmo de *hashing*. Define o método de processamento de *hash*;

b) `HashProcessorDecorator`: decorador de algoritmos de *hashing*. Através dele, é possível alterar os valores de antes e depois da execução do algoritmo decorado;

c) `HashProcessorFilter`: um tipo de decorador especializado para aplicação de

filtros na entrada e na saída dos dados do algoritmo interno;

d) `JavaAPIHashProcessor`: processador de *hash* que pode utilizar qualquer algoritmo de *hashing* disponibilizado pela *Application Programming Interface* (API) do Java, como MD5 e SHA-1, por exemplo.

Na Figura 14 podem ser vistas as classes que provém utilidades diversas para o cliente e o servidor.

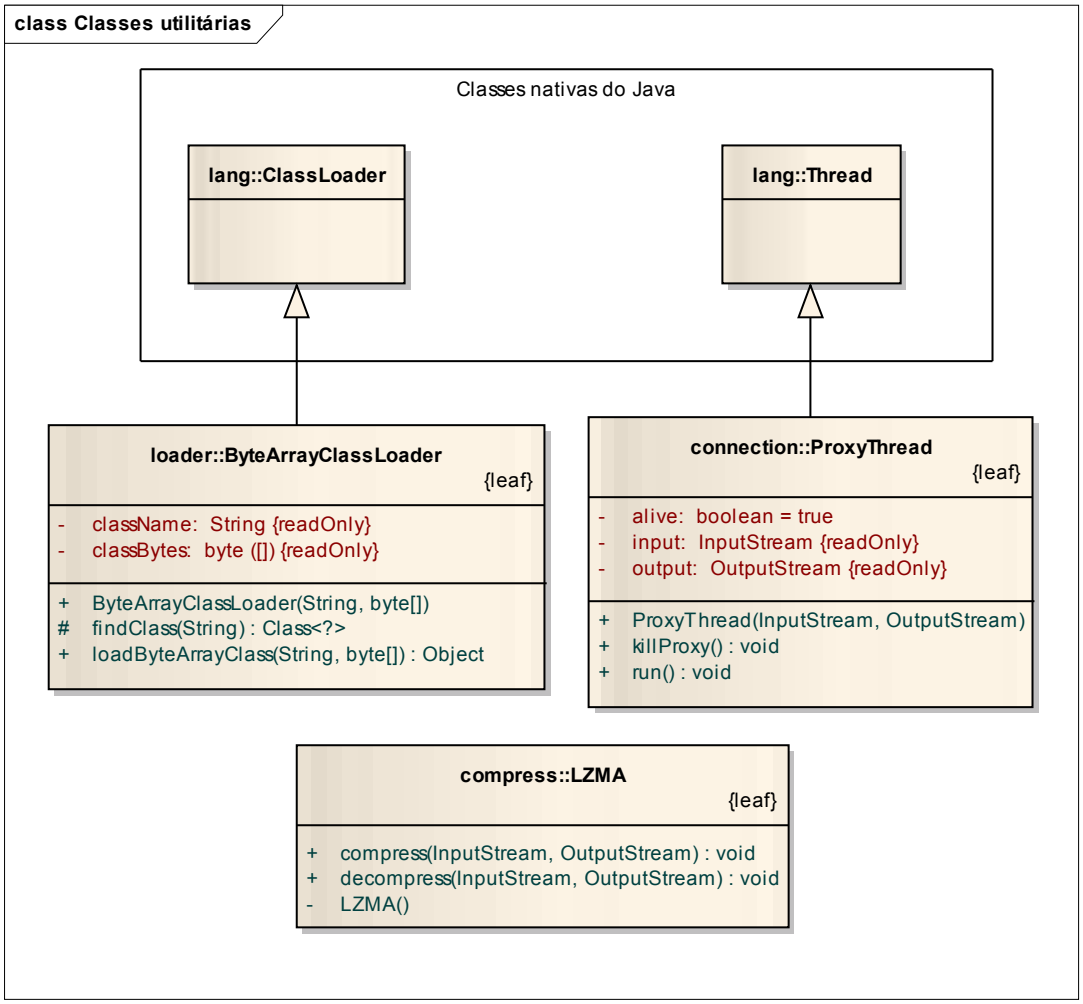


Figura 14 – Diagrama de classes das utilidades diversas

Segue o detalhamento das classes utilitárias:

a) `ByteArrayClassLoader`: um carregador de classes especializado que possibilita carregar uma classe a partir de um *array* de bytes. O *array* de bytes deve ser o conteúdo de uma classe Java compilada. Com ele é possível transferir e utilizar classes geradas remotamente;

b) `ProxyThread`: auxilia na transferência de dados de um canal de comunicação para outro;

c) `LZMA`: permite a compressão e descompressão de dados utilizando o algoritmo

Lempel Ziv Markov *Algorithm* (LZMA)[4]. A compressão torna-se necessária em alguns casos da atualização dinâmica de código, devido ao tamanho de alguns novos códigos. Desta forma, o validador pode comprimir o código antes de enviá-lo ao monitor, que depois irá descomprimi-lo.

3.2.4.2 Classes do servidor

As classes do servidor são aquelas que formam o validador, o componente que deve ser executado juntamente com o software servidor.

A Figura 15 mostra a estrutura base dos geradores de classes, que são utilizados para realizar as atualizações dinâmicas de código.

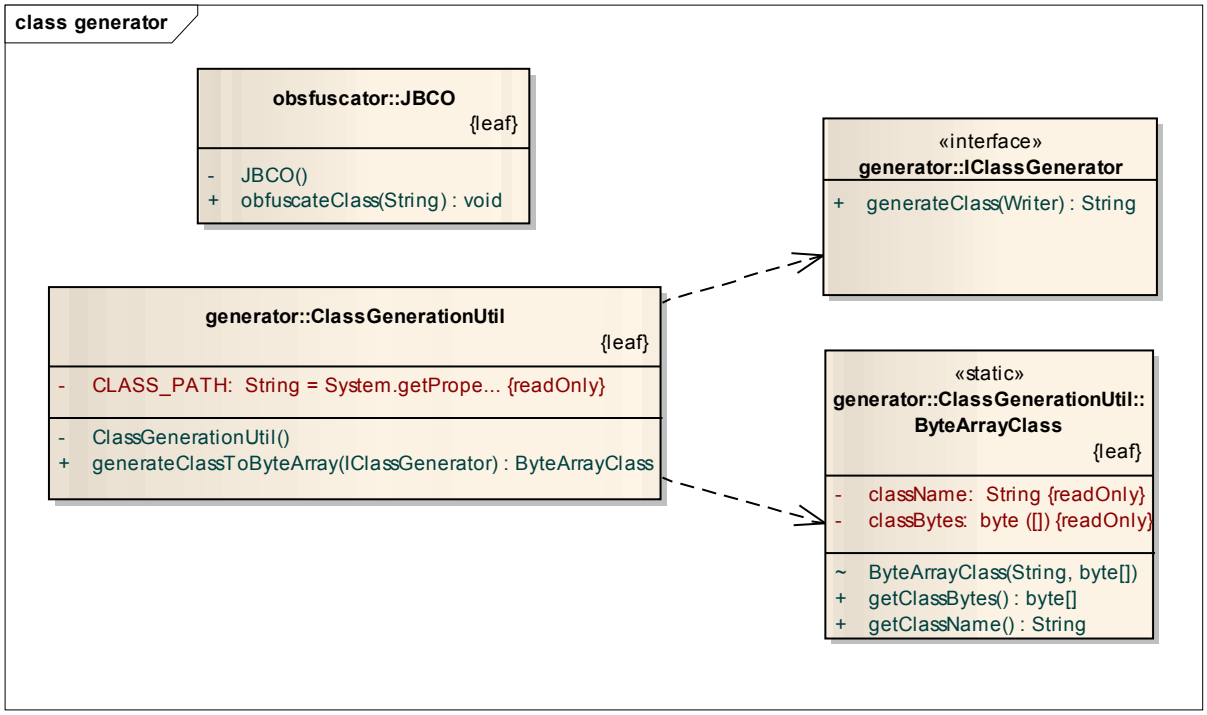


Figura 15 – Diagrama de classes da estrutura base dos geradores de classes

Segue o detalhamento das classes da estrutura base dos geradores de classes:

a) `JBCO`: permite a ofuscação de classes através da API do JBCO;
b) `IClassGenerator`: interface que todo gerador de classe deve implementar;
c) `ClassGenerationUtil`: utilitário para a geração de classes. A partir de um gerador de classes obtém uma classe gerada, compila-a, ofusca o código

[4] LZMA é o método de compressão padrão do formato 7z utilizado pelo software 7-zip. LZMA prove um alto índice de compressão e uma descompressão muito rápida (PALOV, 2009).

compilado e retorna diretamente um objeto contendo o nome e o *array* de bytes do conteúdo da classe gerada. Com estas informações é possível carregar e utilizar esta classe ou transferi-la para um monitor;

d) `ByteArrayClass`: objeto que armazena o nome e o *array* de bytes do conteúdo de uma classe gerada.

A seguir são apresentadas as classes que compõem a porta de entrada do validador. Todo monitor aberto em um computador cliente deve inicialmente se conectar no validador pelo canal de comunicação de controle. Com isso, o monitor é associado a uma sessão que armazena a validade do cliente que esse monitor representa. Se o monitor passa pela validação inicial, o software cliente pode ser aberto e então conectado ao software servidor, por intermédio do monitor e do validador. Esta comunicação ocorre em canais diferentes do canal de controle, mas eles também estão condicionados a validade da sessão. Ou seja, se em qualquer momento o validador detectar alguma irregularidade no cliente, irá invalidar a sessão e consequentemente provocar a desconexão de todos os canais de comunicação com determinado cliente. A Figura 16 mostra as classes que compõem esta estrutura. A seguir é feito o detalhamento das classes:

a) `AbstractSocketServer`: engloba as funcionalidades de um servidor *socket*, fazendo com que as classes filhas precisem apenas tratar as conexões recebidas;

b) `ControlSocketServer`: servidor *socket* que controla as conexões no canal de controle;

c) `ClientSocketServer`: servidor *socket* que controla as conexões de um dos canais de comunicação entre software cliente e o software servidor;

d) `ClientSession`: sessão de um cliente conectado no validador. Indica se o cliente é válido;

e) `IClientSessionListener`: as classes que implementam esta interface podem se tornar observadores de determinada sessão, sendo avisados quando ela for invalidada;

f) `ClientValidationException`: exceção que é lançada quando uma sessão é invalidada para terminar qualquer processo referente ao cliente inválido que está em execução;

g) `ClientSessionManager`: gerenciador de sessão do cliente. Cada monitor que se conecta no validador é gerenciado por uma instância desta classe. Ela controla o canal de comunicação de controle com o monitor e gerencia os trabalhadores que esse monitor executa;

h) `IWorkScheduler`: representa um agendador de trabalhos. O gerenciador de sessão implementa esta interface e disponibiliza-a para os trabalhadores poderem enviar seus trabalhos ou agendar tarefas.

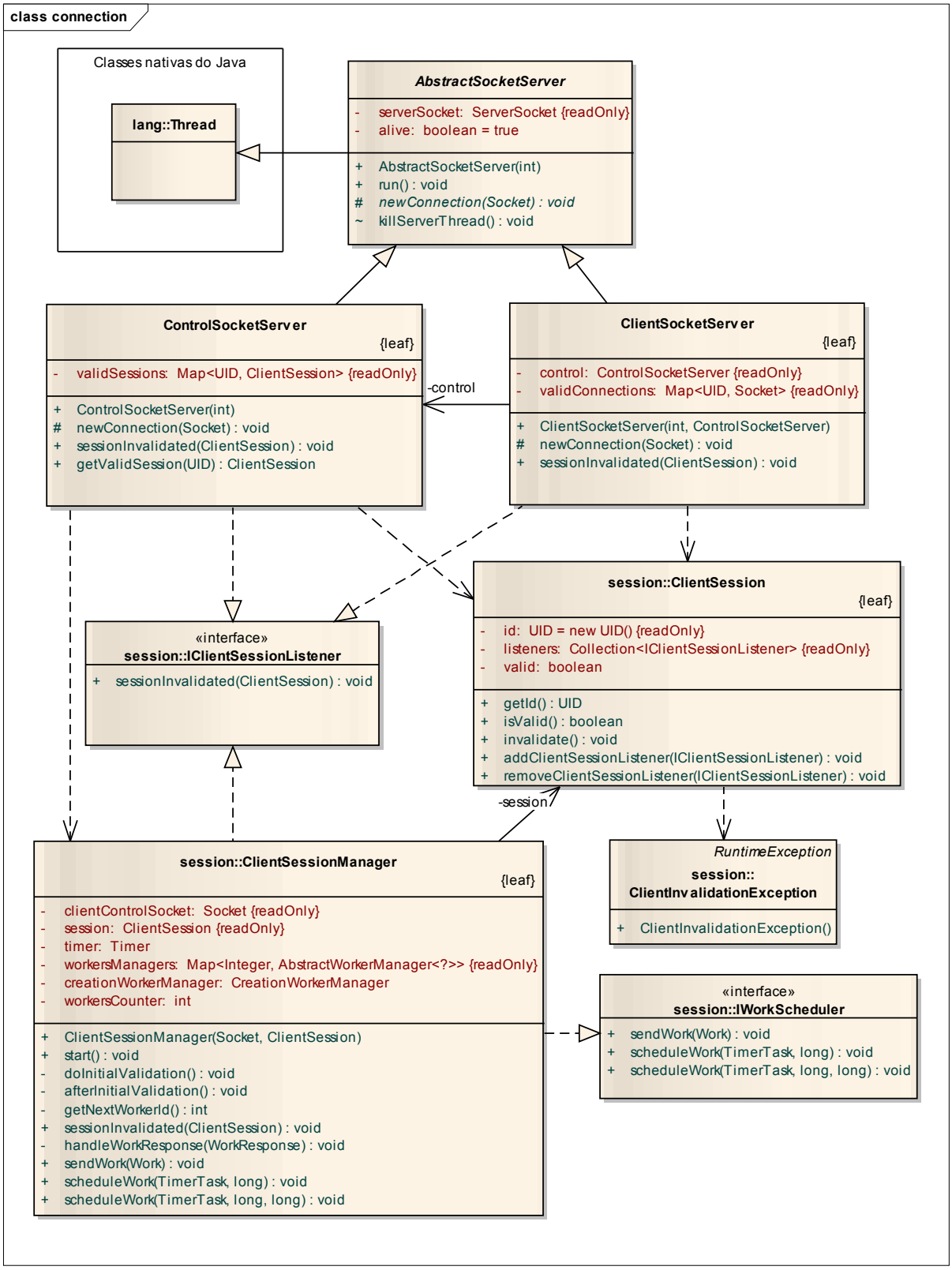


Figura 16 – Diagrama de classes da estrutura de conexão do validador

A seguir são mostradas as classes da estrutura de trabalhadores que o validador utiliza.

Os trabalhadores são os agentes que executam os trabalhos que o validador determina no monitor, como validações, atualização de código e até a execução do software cliente. Os trabalhadores implementados neste sistema visam atender os objetivos deste trabalho, mas a estrutura permite a implementação de trabalhadores diferentes para atender outros objetivos. Cada trabalhador precisa implementar três tipos classes para que seja operacional no monitor e siga as ordens do validador:

a) implementação: a implementação do trabalhador em si que vai ser executada no monitor. Ela precisa estender a classe `AbstractWorker`;
b) gerador: o gerador de classe do trabalhador. Ele deve se basear na implementação do trabalhador e gerar a classe que será executada no monitor;
c) gerenciador: o gerenciador do trabalhador que executa no validador. Ele controla as ordens que são enviadas para o trabalhador e valida suas respostas, podendo invalidar a sessão do cliente a qualquer momento se alguma resposta não estiver correta.

A Figura 17 mostra as classes da parte da implementação dos trabalhadores utilizados pelo sistema. A seguir é feito o detalhamento das classes:

a) `AbstractWorker`: classe base dos trabalhadores. Ela é detalhada na parte das classes básicas da seção de diagrama de classe;
b) `ProxyWorker`: trabalhador que executa a transmissão de dados dos canais de comunicação do software cliente para o validador;
c) `ProxyThread`: classe auxiliar utilizada pelo `ProxyWorker`. Ela é detalhada na parte das classes básicas da seção de diagrama de classe;
d) `AbstractValidationWorker`: classe base dos trabalhadores de validação. Disponibiliza o uso de um algoritmo de *hashing* (gerado aleatoriamente) e de um algoritmo de criptografia (AES de caixa branca). Implementa a execução de ordens de atualização dos algoritmos de *hashing* e de criptografia;
e) `IHashProcessor`: algoritmo de *hashing*. Esta classe é detalhada na parte das classes básicas da seção de diagrama de classe;
f) `IBlockCypher`: algoritmo de criptografia de bloco. Esta classe é detalhada na parte das classes básicas da seção de diagrama de classe;
g) `ClientSoftwareValidationWorker`: trabalhador da validação inicial do software cliente. Implementa as ordens de validação do conteúdo do arquivo executável do software cliente e seu ambiente de execução e de execução do software cliente validado;

h) `ClientRuntimeValidationWorker`: trabalhador da validação periódica do software cliente. Implementa as ordens de validação em tempo de execução do software cliente e seu ambiente de execução e da finalização do software cliente que se torna inválido.

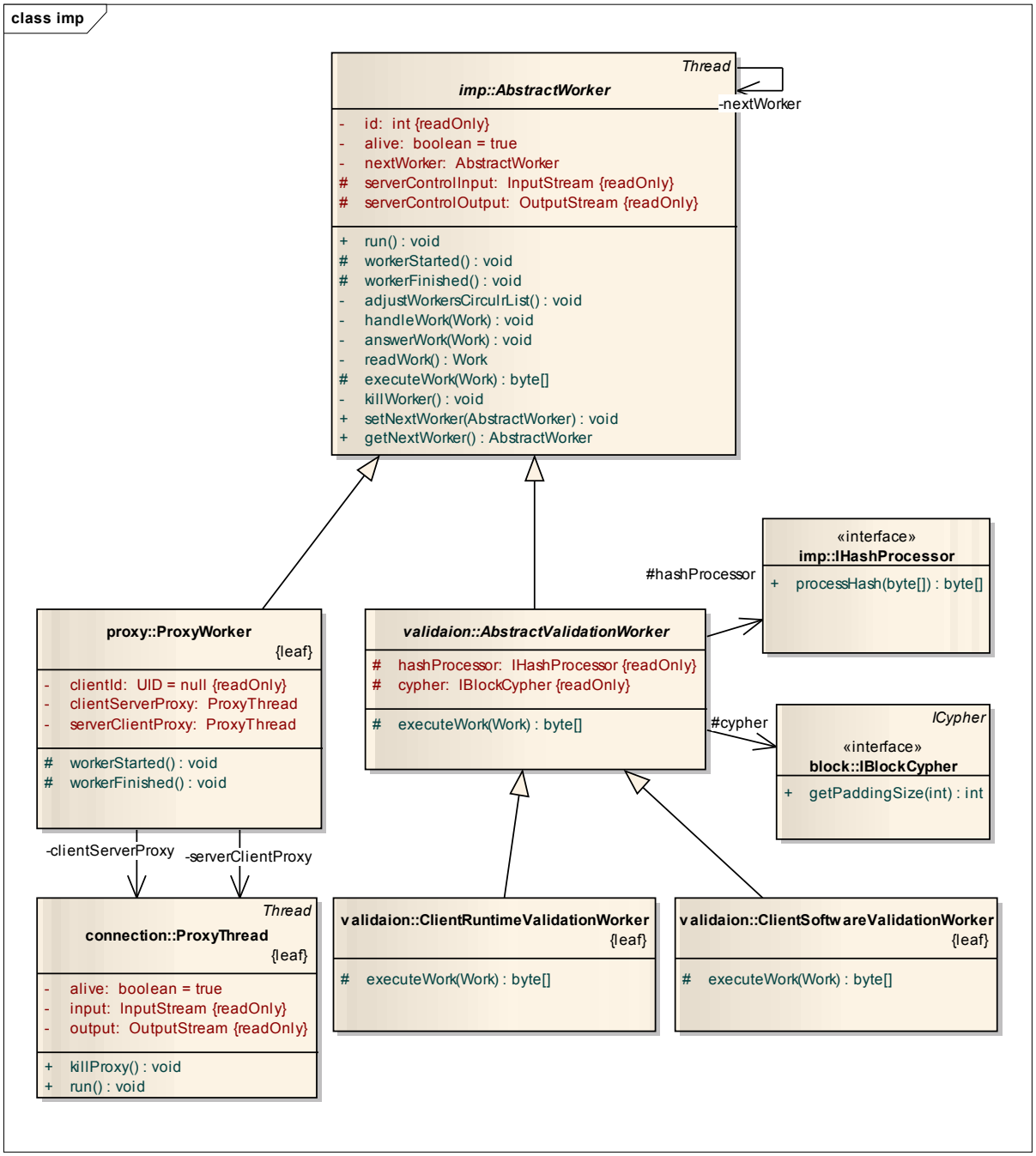


Figura 17 – Diagrama de classes da implementação dos trabalhadores

A Figura 18 mostra as classes da parte de geradores dos trabalhadores utilizados pelo sistema.

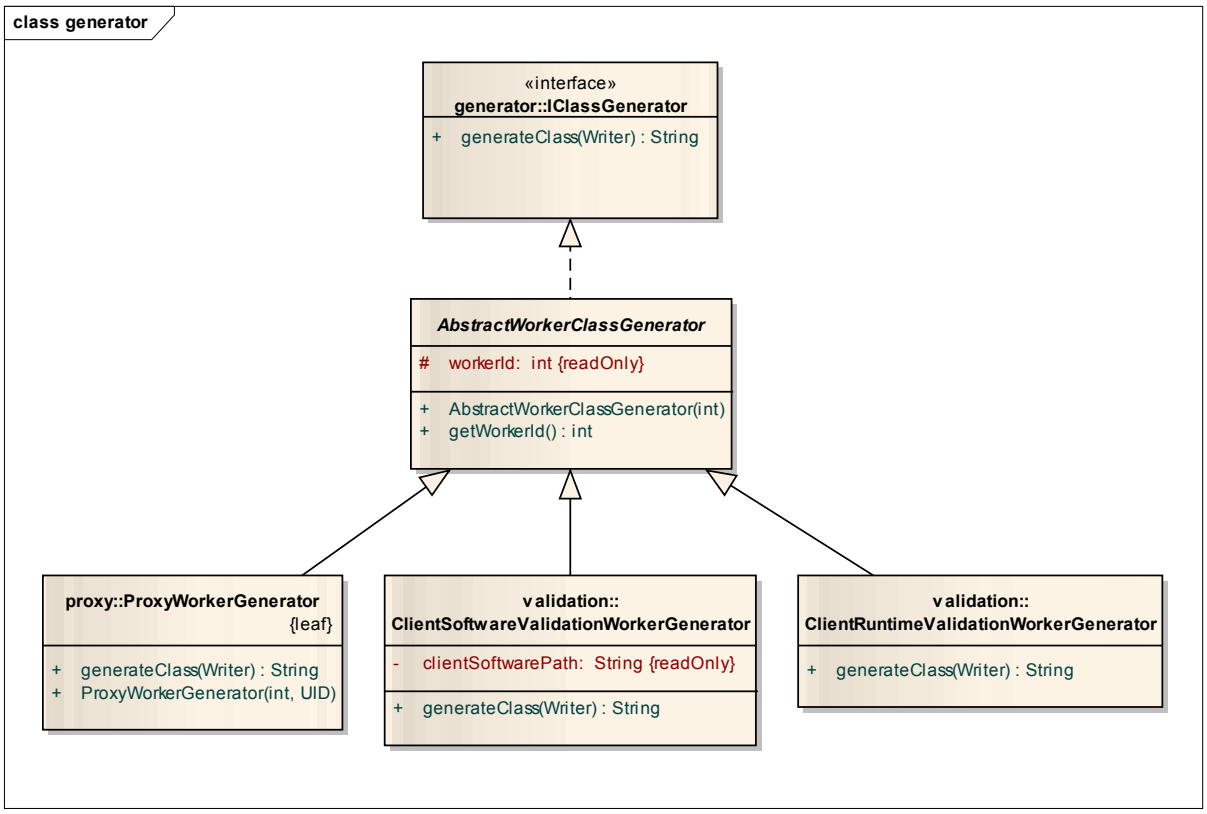


Figura 18 – Diagrama de classes dos geradores dos trabalhadores

A seguir é feito o detalhamento das classes:

a) `IClassGenerator`: interface que todo gerador de classe deve implementar. Ela é detalhada no início da parte das classes do servidor, na seção de diagrama de classe;

b) `AbstractWorkerClassGenerator`: classe base dos geradores de código dos trabalhadores;

c) `ProxyWorkerGenerator`: gerador de código de trabalhadores do tipo `ProxyWorker`;

d) `ClientSoftwareValidationWorkerGenerator`: gerador de código de trabalhadores do tipo `ClientSoftwareValidationWorker`;

e) `ClientRuntimeValidationWorkerGenerator`: gerador de código de trabalhadores do tipo `ClientRuntimeValidationWorker`.

A Figura 19 mostra as classes da parte de gerenciadores dos trabalhadores utilizados pelo sistema.

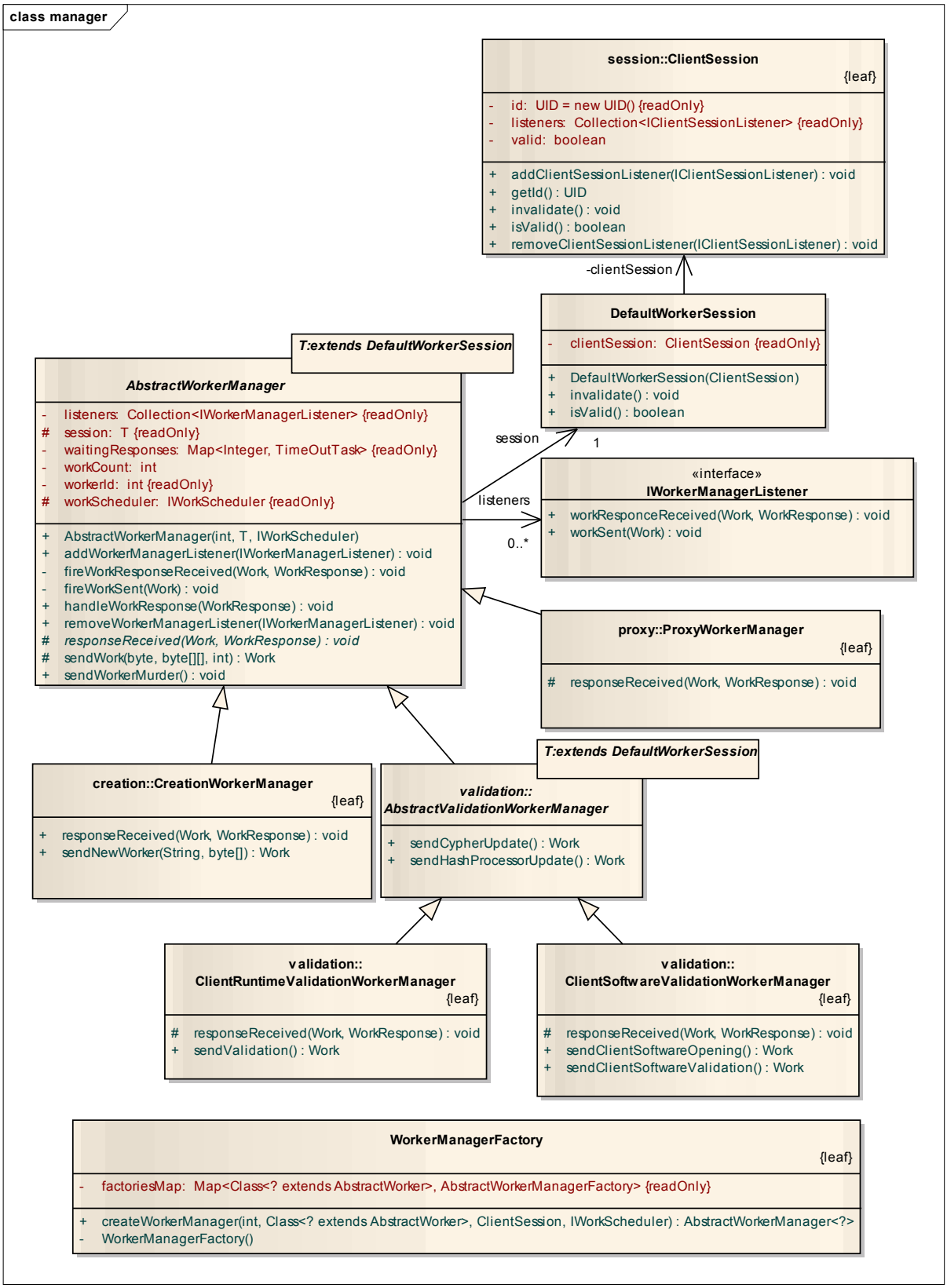


Figura 19 – Diagrama de classes dos gerenciadores dos trabalhadores

A seguir é feito o detalhamento das classes:

a) `AbstractWorkerManager`: classe base dos gerenciadores de trabalhadores. Implementa a estrutura básica do envio de ordens e do recebimento das respostas,

restando as subclasses apenas criar a ordem em si e tratar sua resposta. Possibilita o envio da ordem de destruição do trabalhador gerenciado;

b) `IWorkManagerListener`: interface disponibilizada para aqueles que desejam ser avisados dos eventos que ocorrem em um gerenciador de trabalhador. O `AbstractWorkerManager` possui métodos para adicionar e remover observadores de seus eventos. Os eventos que podem ser observados são o envio de uma ordem e o recebimento de uma resposta válida;

c) `DefaultWorkerSession`: sessão padrão dos gerenciadores de trabalhadores. As classes de gerenciadores de trabalhadores mais específicas podem possuir classes de sessões mais específicas também (a fim de armazenar informações adicionais). Cada sessão de cada gerenciador de trabalhador está ligada a sessão do cliente, então se a sessão de um trabalhador for invalidada acabará invalidando também a sessão do cliente;

d) `ClientSession`: sessão de um cliente conectado no validador. Esta classe é detalhada na parte das classes do servidor, na seção de diagrama de classe;

e) `CreationWorkerManager`: gerenciador dos trabalhadores de criação. A implementação deste trabalhador está diretamente no monitor, por isso ele não precisa ter a implementação e o gerador no validador. Entretanto, o gerenciador é indispensável, pois é ele quem vai controlar este trabalhador. O trabalhador de criação tem a função de receber as classes dos trabalhadores vindas do validador e de criá-las e executá-las no monitor. Por isso, o `CreationWorkerManager` possibilita o envio de novos trabalhadores para o monitor. O detalhamento do trabalhador de criação é feito na parte de classes do cliente, na seção de diagrama de classe;

f) `ProxyWorkerManager`: gerenciador de trabalhadores do tipo `ProxyWorker`. Possibilita o envio da ordem de início e parada da transmissão de dados entre software cliente e o validador;

g) `AbstractValidationWorkerManager`: gerenciador base dos gerenciadores de trabalhadores de validação. Possibilita o envio de atualização do algoritmo de *hashing* e do algoritmo de criptografia;

h) `ClientSoftwareValidationWorkerManager`: gerenciador de trabalhadores do tipo `ClientSoftwareValidationWorker`. Possibilita o envio das ordens de validação do conteúdo do arquivo executável do software cliente e seu ambiente de

execução e de execução do software cliente validado;

i) `ClientRuntimeValidationWorkerManager`: gerenciador de trabalhadores do tipo `ClientRuntimeValidationWorker`. Possibilita o envio das ordens de validação em tempo de execução do software cliente e seu ambiente de execução e da finalização do software cliente que se tornou inválido;
j) `WorkerManagerFactory`: fábrica de gerenciadores de trabalhadores. Possibilita a criação do gerenciador adequado dado o tipo do trabalhador.

A Figura 20 mostra as classes da estrutura de criptografia do sistema. As implementações dos algoritmos de criptografia foram utilizadas durante o desenvolvimento do gerador de classes de criptografia de AES de caixa branca. Quando ele foi terminado, estas classes passaram a não ser mais utilizadas. Entretanto, elas ainda podem ser usadas por futuras implementações, por isso ainda estão integradas no sistema. A seguir é feito o detalhamento das classes:

a) `AbstractChainingBlockCypher`: classe base dos algoritmos de criptografia de bloco. Ela é detalhada na parte das classes básicas da seção de diagrama de classe;
b) `AESCypher`: implementação do algoritmo de criptografia de chave simétrica AES;
c) `WhiteBoxAESCypher`: implementação do algoritmo de criptografia de chave simétrica AES para cenários de caixa branca. Dada uma chave comum do AES de 128 bits, ele gera uma versão customizada do algoritmo específica para esta chave, armazenando as tabelas de consulta geradas na memória;
d) `IClassGenerator`: interface que todo gerador de classe deve implementar. Ela é detalhada no início da parte das classes do servidor, na seção de diagrama de classe;
e) `WhiteBoxAESCypherGenerator`: gerador de classes de algoritmos de criptografia AES de caixa branca. As classes geradas são baseadas no `WhiteBoxAESCypher`, mas as tabelas de consulta geradas são colocadas diretamente na nova classe. Desta forma são criadas classes específicas para chaves do AES;
f) `RandomSubBoxGenerator`: gerador de `S-Boxes` que seguem a mesma idéia do `S-Box` do AES, mas gerados de maneira aleatória. Eles são utilizados para fazer a codificação aleatória das caixas de consulta dos algoritmos de criptografia de AES de caixa branca gerados.

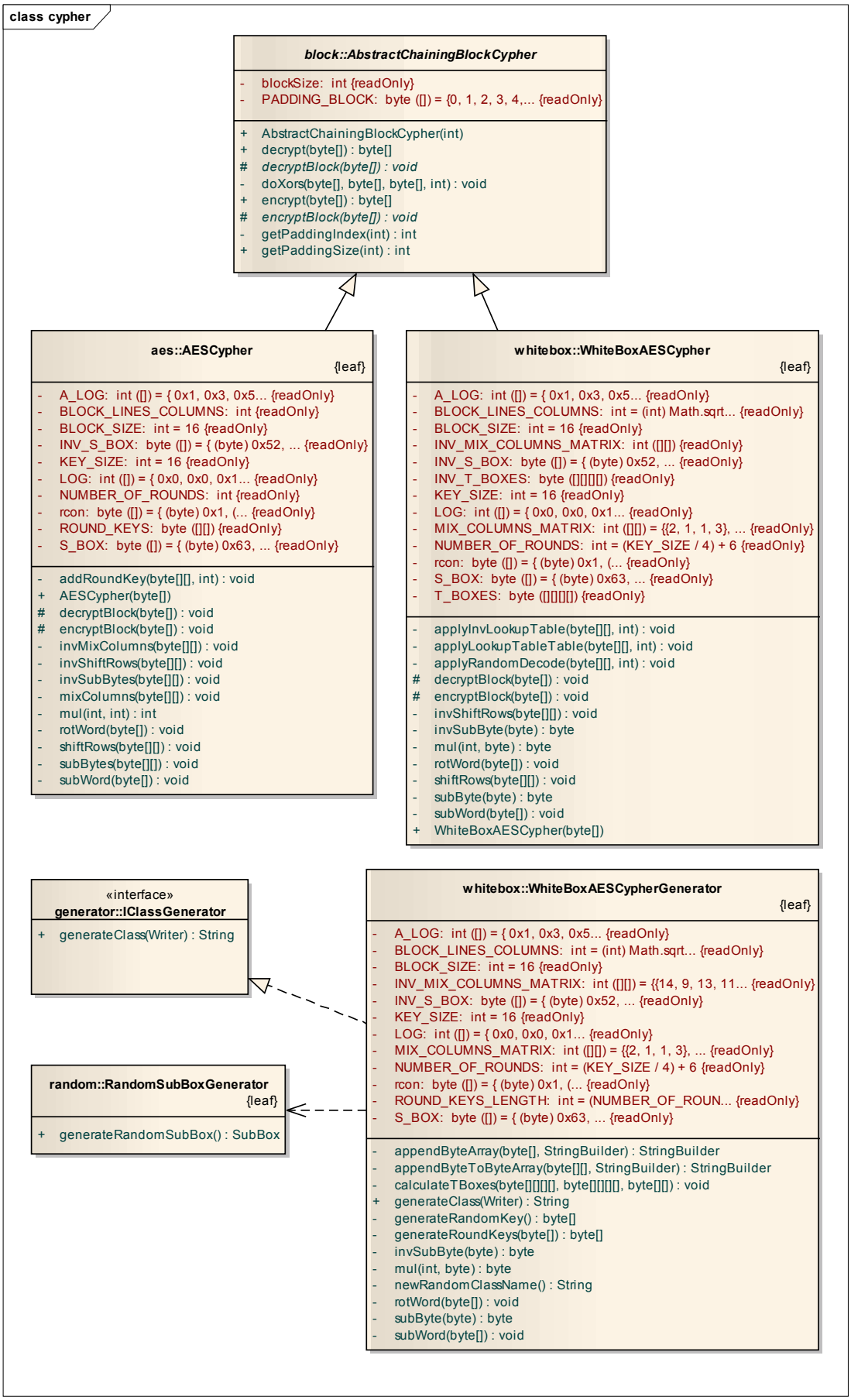


Figura 20 – Diagrama de classes da estrutura de criptografia

A Figura 21 mostra as classes da estrutura de algoritmos de *hashing* do sistema.

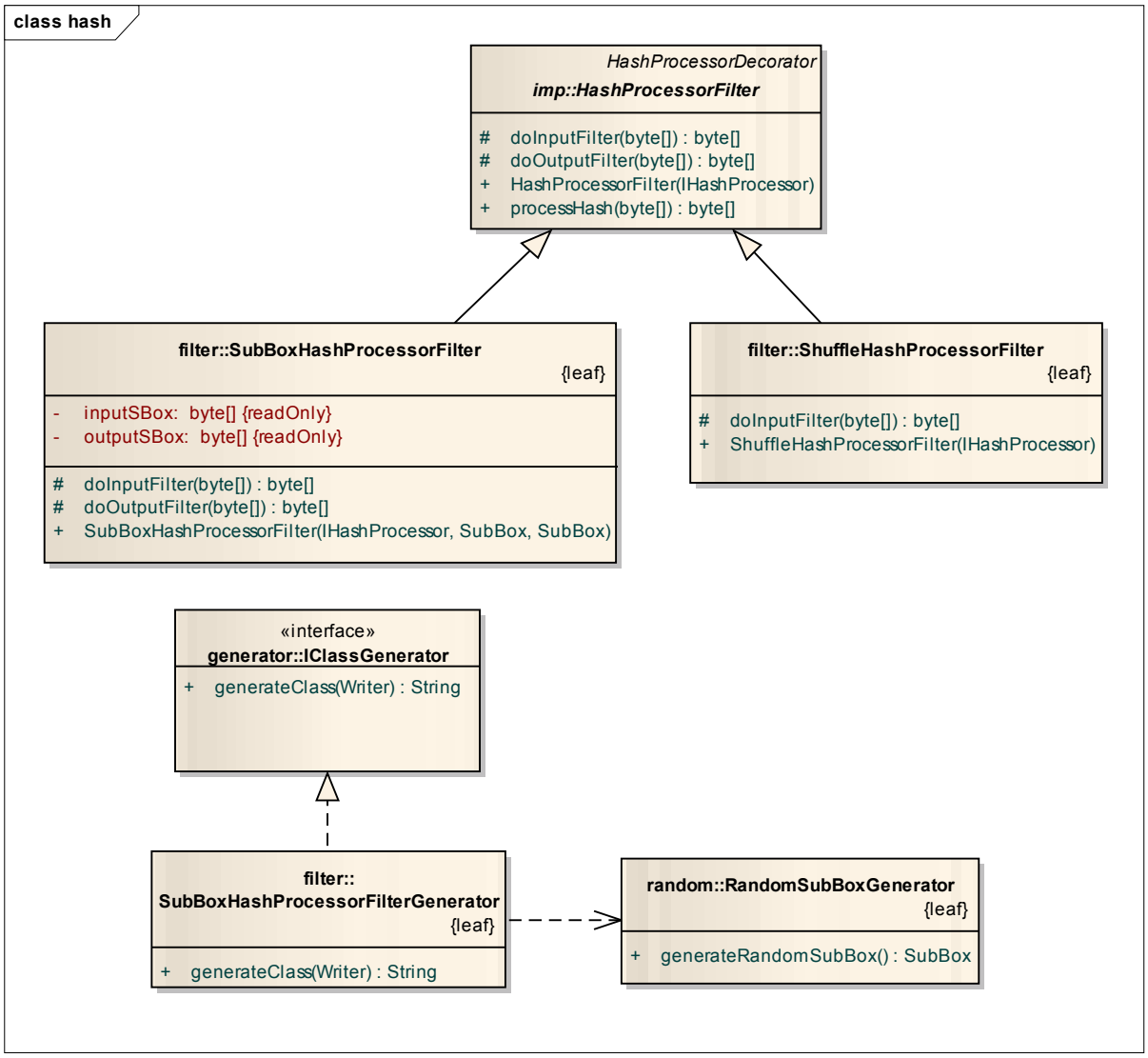


Figura 21 – Diagrama de classes da estrutura de algoritmos de *hashing*

A seguir é feito o detalhamento das classes:

a) `HashProcessorFilter`: decorador especializado para aplicação de filtros na entrada e na saída dos dados do algoritmo interno. Esta classe é detalhada na parte das classes básicas da seção de diagrama de classe;

b) `ShuffleHashProcessFilter`: filtra a entrada do algoritmo de *hashing* interno, separando os dados em blocos e embaralhando-os;

c) `SubBoxHashProcessorFilter`: filtra a entrada e a saída do algoritmo de *hashing* interno, substituindo os bytes de acordo com as `S-Boxes` definidas;

d) `IClassGenerator`: interface que todo gerador de classe deve implementar. Ela é detalhada no início da parte das classes do servidor, na seção de diagrama de classe;

e) `RandomSubBoxGenerator`: gerador de `S-Boxes` que seguem a mesma idéia do `S-Box` do AES, mas gerados de maneira aleatória;

f) `SubBoxHashProcessorFilterGenerator`: gerador de classes de algoritmos de *hashing* aleatórios. As classes geradas são baseadas no `SubBoxHashProcessorFilter`, sendo que as `S-Boxes` são geradas através do `RandomSubBoxGenerator` e colocadas diretamente na classe gerada.

Por fim, o validador deve ser executado através da classe `QuimeraServer`. Ela inicia a estrutura de controle e os servidores que escutam as conexões dos canais de comunicação vindas dos monitores.

3.2.4.3 Classes do cliente

As classes do cliente são compostas por apenas duas classes: o trabalhador de criação e a classe que inicia o monitor e faz a conexão com o validador. Depois de iniciado, o monitor passa a receber novas classes através do trabalhador de criação, mas isso depende das ordens do validador. Estas classes podem ser vistas na Figura 22.

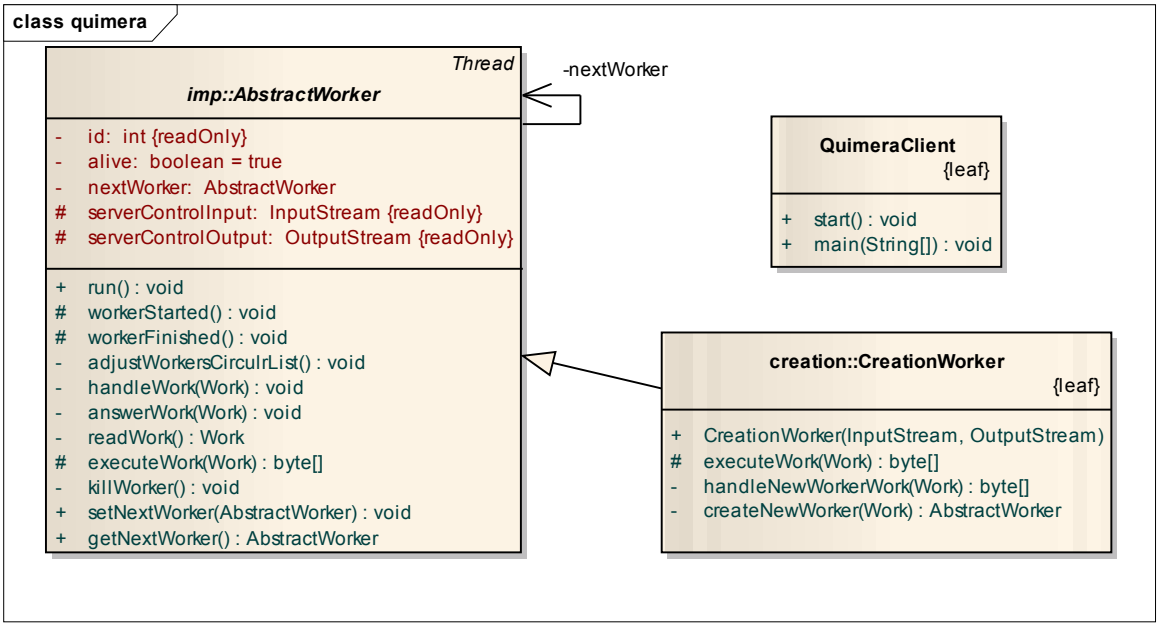


Figura 22 – Diagrama de classes do cliente

A seguir é feito o detalhamento das classes:

a) `AbstractWorker`: classe base dos trabalhadores. Ela é detalhada na parte das classes básicas da seção de diagrama de classe;

b) `CreationWorker`: trabalhador de criação. Recebe as classes dos trabalhadores

vindas do validador, as instancia e integra-as na rede de trabalhadores mantidas no monitor;

c) `QuimeraClient`: classe que executa o monitor, inicia a conexão com o validador e inicia o `CreationWorker`.

3.3 IMPLEMENTAÇÃO

Nesta seção são mostradas as técnicas e ferramentas utilizadas no desenvolvimento do sistema de proteção, o detalhamento de partes importantes da implementação e a operacionalidade do sistema.

3.3.1 Técnicas e ferramentas utilizadas

Nesta seção são apresentadas as técnicas e ferramentas utilizadas para determinados assuntos do desenvolvimento do sistema de proteção. São elas:

a) linguagem de programação: Java 1.6;
b) ambiente de desenvolvimento: *Integrated Development Environment* (IDE) Eclipse 3.6;
c) geração de código: biblioteca Apache Velocity 1.6.4;
d) compilação de código: biblioteca *tools*, nativa do Java *Development Kit* (JDK);
e) algoritmo de compressão de dados: LZMA;
f) ofuscador de código Java: ferramenta JBCO;
g) testes automatizados: JUnit 4.3.1.

3.3.2 Gerador de algoritmos de AES de caixa branca

Nesta seção é descrito todo o processo de criação do gerador de algoritmos de AES de caixa branca.

Inicialmente foi necessário obter uma implementação de AES em Java com o código aberto que fosse bem implementada, com as operações do algoritmo bem separadas em

métodos distintos. A implementação escolhida foi a feita por Raj (2004). O próximo passo foi verificar se esta implementação era válida. Para isto, foram criados testes automatizados (com o auxílio da biblioteca JUnit) que comparam os bytes dos pacotes encriptados e decriptados pelo algoritmo com a implementação de AES oficial do Java. Com o algoritmo validado, foi criada a classe `AESCypher`, contendo o algoritmo de AES de Raj. O Quadro 18 mostra o corpo do método de encriptação da implementação de AES utilizada.

```
addRoundKey(state, 0);
for (int i = 1; i <= 9; i++) {
      subBytes(state);
      shiftRows(state);
      mixColumns(state);
      addRoundKey(state, i);
}

subBytes(state);
shiftRows(state);
addRoundKey(state, 10);
```

Quadro 18 – Método de encriptação da implementação de AES tradicional

A partir do `AESCypher` foi criada a classe `WhiteBoxAESCypher`, onde o algoritmo de AES tradicional seria adaptado para a versão de caixa branca. Antes de iniciar a adaptação, foram criados testes automatizados similares aos usado anteriormente, mas para o algoritmo de AES de caixa branca. Desta forma, poderia ser verificado se a adaptação estava ocorrendo de maneira correta, pois o AES de caixa branca sem as codificações aleatórias deve retornar os mesmos resultados de um AES tradicional.

O primeiro passo da adaptação foi esconder a chave juntamente com as `S-Boxes` em tabelas de consulta (denominadas de `T-Boxes`). Ou seja, juntar as operações `AddRoundKey` e `SubBytes`. Como a chave é atribuída no construtor da classe do algoritmo, foi necessário calcular as tabelas de consulta neste mesmo ponto. O Quadro 19 mostra a implementação deste cálculo.

```
/*
 * Cria as T-boxes.
 * Uma T-box é composta por 256 posições, assim como a S-box.
 * Haverão 16 T-boxes para cada rodada, uma para cada byte dos blocos que
 * serão criptografados.
 *
 */
T_BOXES = new byte[NUMBER_OF_ROUNDS][4][256];

/*
 * Cada byte de uma T-box da rodada de 1 até a 9 é composta por:
 * T-box(b) = S-box(b xor key)
 * O byte da S-box baseado no índice do byte da T-box em ou-exclusivo com
 * o byte da chave de rodada correspondente.
 */
for (int roundIndex = 0; roundIndex < NUMBER_OF_ROUNDS-1; roundIndex++) {
      int curRoundKeyIndex = roundIndex * BLOCK_SIZE;

      for (int blockByteIndex = 0; blockByteIndex < BLOCK_SIZE;
blockByteIndex++) {
            byte keyByte = roundKeys[curRoundKeyIndex + blockByteIndex >>
2][blockByteIndex % 4];

            for (int tboxByte = 0; tboxByte < S_BOX.length; tboxByte++) {
                  T_BOXES[roundIndex][blockByteIndex][tboxByte] =
subByte((byte) (tboxByte ^ keyByte));
            }
      }
}

//Calcula as últimas T-boxes, que utilizam as duas últimas chaves de
rodada.
int penultimateRoundKeyIndex = NUMBER_OF_ROUNDS-1 * BLOCK_SIZE;
int lastRoundKeyIndex = penultimateRoundKeyIndex + BLOCK_SIZE;
for (int blockByteIndex = 0; blockByteIndex < BLOCK_SIZE;
blockByteIndex++) {
      int line = blockByteIndex >> 2;
      int column = blockByteIndex % 4;
      byte penultimateKeyByte = roundKeys[penultimateRoundKeyIndex +
line][column];

      //O último byte tem que ser referente a posição que os bytes ficam
depois de um shift row.
      int lastCol = (column - line) & 3;
      byte lastKeyByte = roundKeys[lastRoundKeyIndex + line][lastCol];

      for (int tboxByte = 0; tboxByte < S_BOX.length; tboxByte++) {
            T_BOXES[NUMBER_OF_ROUNDS-1][blockByteIndex][tboxByte] =
subByte((byte) (tboxByte ^ penultimateKeyByte)) ^ lastKeyByte;
      }
}
```

Quadro 19 – Cálculo das tabelas de consulta iniciais

Depois do primeiro passo, o método de encriptação foi alterado, podendo ser visto no Quadro 20.

```
for (int i = 1; i <= 9; i++) {
      //As operações de subBytes e addRoundKey estão contidas nas T_Boxes,
então basta apenas substituir.
      for (int l = 0; l < 4; l++) {
            for (int c = 0; c < 4; c++) {
                  state[l][c] = T_BOXES[i-1][(l * 4) +
                                 c][state[l][c] & 0xFF];
            }
      }

      shiftRows(state);
      mixColumns(state);
}

shiftRows(state);
//As operações de subBytes e addRoundKey estão contidas nas T_Boxes, então
basta apenas substituir.
for (int l = 0; l < 4; l++) {
      for (int c = 0; c < 4; c++) {
            state[l][c] = T_BOXES[9][(l * 4) +
                           c][state[l][c] & 0xFF];
      }
}
```

Quadro 20 – Método de encriptação depois do primeiro passo da adaptação

O próximo passo foi a conversão da operação de `MixColumns` em tabelas de consulta. O Quadro 21 mostra como estas tabelas são calculadas.

```
MIX_COLUMN_BOXES = new byte[4][4][256];
for (int i = 0; i < 256; i++) {
      byte b = (byte) i;
      MIX_COLUMN_BOXES[0][0][i] = mul(2, b);
      MIX_COLUMN_BOXES[0][1][i] = b;
      MIX_COLUMN_BOXES[0][2][i] = b;
      MIX_COLUMN_BOXES[0][3][i] = mul(3, b);

      MIX_COLUMN_BOXES[1][0][i] = mul(3, b);
      MIX_COLUMN_BOXES[1][1][i] = mul(2, b);
      MIX_COLUMN_BOXES[1][2][i] = b;
      MIX_COLUMN_BOXES[1][3][i] = b;

      MIX_COLUMN_BOXES[2][0][i] = b;
      MIX_COLUMN_BOXES[2][1][i] = mul(3, b);
      MIX_COLUMN_BOXES[2][2][i] = mul(2, b);
      MIX_COLUMN_BOXES[2][3][i] = b;

      MIX_COLUMN_BOXES[3][0][i] = b;
      MIX_COLUMN_BOXES[3][1][i] = b;
      MIX_COLUMN_BOXES[3][2][i] = mul(3, b);
      MIX_COLUMN_BOXES[3][3][i] = mul(2, b);
}
```

Quadro 21 – Cálculo das tabelas de consulta de `MixColumns`

Depois disso, o método que executa o `MixColumns` foi alterado para substituir os bytes pelos das tabelas de consulta ao invés de fazer a multiplicação de matrizes. Apesar disso, as somas ainda precisaram ser mantidas. O Quadro 22 mostra a implementação deste método

antes e depois da criação das tabelas de consulta.

```
//Antes
for (int i = 0; i < 4; i++) {
      state[0][i] = (mul(2, tmp[0][i]) ^ mul(3, tmp[1][i]) ^
                     tmp[2][i] ^ tmp[3][i]);
      state[1][i] = (mul(2, tmp[1][i]) ^ mul(3, tmp[2][i]) ^
                     tmp[0][i] ^ tmp[3][i]);
      state[2][i] = (mul(2, tmp[2][i]) ^ mul(3, tmp[3][i]) ^
                     tmp[1][i] ^ tmp[0][i]);
      state[3][i] = (mul(2, tmp[3][i]) ^ mul(3, tmp[0][i]) ^
                     tmp[2][i] ^ tmp[1][i]);
}
```

```
//Depois
for (int c = 0; c < 4; c++) {
      for (int l = 0; l < 4; l++) {
            state[l][c] = (MIX_COLUMN_BOXES[0][tmp[0][c] & 0xFF][l] ^
                           MIX_COLUMN_BOXES[1][tmp[1][c] & 0xFF][l] ^
                           MIX_COLUMN_BOXES[2][tmp[2][c] & 0xFF][l] ^
                           MIX_COLUMN_BOXES[3][tmp[3][c] & 0xFF][l]);
      }
}
```

Quadro 22 – Implementações da operação `MixColumns`

O próximo passo foi a integração das tabelas de consulta das operações de `AddRoundKey` e `SubBytes` com as da operação `MixColumns`. Ou seja, integrar as `T_BOXES` com as `MIX_COLUMNS_BOXES`. Para isto, os cálculos das tabelas mostradas anteriormente foram substituídos por uma única rotina de cálculo. O Quadro 23 mostra esta rotina, que é o cálculo das tabelas de consulta finais do algoritmo de AES de caixa branca.

```
T_BOXES = new byte[NUMBER_OF_ROUNDS][BLOCK_SIZE][4][256];

/*
 * Calcula as primeiras T-Boxes.
 */
final int LAST_ROUND = NUMBER_OF_ROUNDS-1;
final int PENULTIMATE_ROUND_KEY_INDEX = LAST_ROUND * 4;
final int LAST_ROUND_KEY_INDEX = PENULTIMATE_ROUND_KEY_INDEX + 4;
for (int roundIndex = 0; roundIndex < NUMBER_OF_ROUNDS; roundIndex++) {
      boolean isLastRound = roundIndex == LAST_ROUND;
      int curRoundKeyIndex = roundIndex * 4;

      for (int blockByteIndex = 0; blockByteIndex < BLOCK_SIZE;
blockByteIndex++) {
            int column = blockByteIndex >> 2;
            int line = blockByteIndex % 4;
            byte keyByte = roundKeys[curRoundKeyIndex + line][column];

            //Usa a coluna multiplicadora de acordo com a posição do byte
depois de sofrer um shift rows.
            int[] mixColumnMultipliers = MIX_COLUMNS_MATRIX[column];
            for (int tboxByte = 0; tboxByte < S_BOX.length; tboxByte++) {
                  //Cálculo da T-Box.
                  byte tBoxResult = subByte((byte) (tboxByte ^ keyByte));

      //Multiplica cada byte por seus possíveis estados em um mixColumn.
                  for (int multIndex = 0; multIndex <
mixColumnMultipliers.length; multIndex++) {
                        if (!isLastRound) {

      T_BOXES[roundIndex][blockByteIndex][multIndex][tboxByte] =
mul(mixColumnMultipliers[multIndex], tBoxResult);
                        }
                  }
            }
      }
}

/*
 * Calcula as últimas T-boxes.
 */
for (int blockByteIndex = 0; blockByteIndex < BLOCK_SIZE;
blockByteIndex++) {
      int column = blockByteIndex >> 2;
      int line = blockByteIndex % 4;

      //O último byte tem que ser referente a posição que os bytes ficam
depois de um shift row.
      int shiftedCol = (line - column) & 3;
      byte lastKeyByte = roundKeys[LAST_ROUND_KEY_INDEX +
shiftedCol][column];
      byte penultimateKeyByte = roundKeys[PENULTIMATE_ROUND_KEY_INDEX +
line][column];

      for (int tboxByte = 0; tboxByte < S_BOX.length; tboxByte++) {
            //Calcula os bytes da última T-box.
            T_BOXES[LAST_ROUND][blockByteIndex][0][tboxByte] =
subByte((byte) (tboxByte ^ penultimateKeyByte)) ^ lastKeyByte;
      }
}
```

Quadro 23 – Cálculo das tabelas de consultas finais

O Quadro 24 mostra como o método de encriptação ficou depois da implementação do último passo apresentado.

```
//implementação da encriptação
for (int round = 0; round < 9; round++) {
      shiftRows(state);

      applyLookupTable(state, round);
}

shiftRows(state);

applyLookupTable(state, 10);
```

```
//implementação do método applyLookupTable
/*
 * As operações de subBytes, addRoundKey e mixColumns já estão contidas
 * nas T_BOXES, então basta apenas substituir e
 * executar os xors com os bytes nas posições de antes do shift rows.
 */
for (int c = 0; c < 4; c++) {
      for (int l = 0; l < 4; l++) {
      state[l][c] = T_BOXES[round][c][tmp[0][c] & 0xFF][l] ^
        T_BOXES[round][4 + ((c + 1) % 4)][tmp[1][c] & 0xFF][l] ^
        T_BOXES[round][(2 * 4) + ((c + 2) % 4)][tmp[2][c] & 0xFF][l] ^
        T_BOXES[round][(3 * 4) + ((c + 3) % 4)][tmp[3][c] & 0xFF][l];
      }
}
```

Quadro 24 – Método de encriptação depois do terceiro passo da adaptação

Para completar a adaptação resta apenas a inserção de codificações geradas aleatoriamente. Através da classe `RandomSubBoxGenerator` é possível gerar `S-Boxes` aleatórias, que são utilizadas para codificar os bytes das tabelas de consulta (pela substituição). Com isso, é necessário decodificar os bytes depois de cada rodada do AES feita em cima das tabelas codificadas. O Quadro 25 mostra como o método de encriptação ficou depois disso.

```
for (int round = 0; round < 9; round++) {
      shiftRows(state);

      applyLookupTable(state, round);

      applyRandomDecode(state, round);
}

shiftRows(state);

applyLookupTable(state, 10);

applyRandomDecode(state, 10);
```

Quadro 25 – Método de encriptação depois último passo da adaptação

Os estudos sobre o AES de caixa branca não descrevem como seria a decriptação neste tipo de algoritmo. Este ponto foi explorado por este trabalho. Concluiu-se que é possível

adaptar os mesmos passos de conversão para as lógicas inversas da decriptação. Com isso, foi possível gerar tabelas de consultas inversas para utilizar no método de decriptação. Desta forma, pode-se esconder a chave em um algoritmo utilizado tanto para encriptar como decriptar dados.

Posteriormente, foi criada a classe do gerador de algoritmos de AES de caixa branca: `WhiteBoxAESCypherGenerator`. Ele faz todos os cálculos das tabelas de consultas e gera uma classe com as tabelas definidas diretamente em atributos dela. A geração de classe é feita utilizando o Velocity, baseando-se em um modelo de classe criado a partir do `WhiteBoxAESCypher`. O Quadro 26 mostra como é feita a geração da classe de algoritmo de AES de caixa branca através do Velocity.

```
Velocity.init();

VelocityContext context = new VelocityContext();
context.put("className", className);

//Aplica as T-Boxes no modelo
context.put("tBoxes", tBoxesStrings);

//Aplica as T-Boxes invertidas no modelo.
context.put("invTBoxes", invTBoxesStrings);

//Aplica as caixas de decodificação.
context.put("decodBoxes", decodBoxesStrings);

//Gera a classe.
Velocity.evaluate(context, writer, "WhiteBoxAESCypher", new FileReader(
"quimera/cypher/generator/block/aes/whitebox/WhiteBoxAESCypherTemplate.vm"));
writer.flush();
```

Quadro 26 – Geração da classe de algoritmo de AES de caixa branca através do Velocity

Um dos problemas enfrentados na utilização deste mecanismo de geração de classes foi uma limitação do próprio Java. Ele não suporta classes que possuam métodos que ocupem mais do que 64 KB. Uma rede de tabelas de consulta do algoritmo de AES de caixa branca gasta 160 KB e existe uma rede para encriptação e uma para decriptação. Como estas tabelas de consulta são inseridas diretamente no código da classe gerada em forma de atributos, elas são iniciadas no construtor da classe, que é um método. Desta forma, o construtor ocuparia 320 KB, o que ultrapassa o limite. Para contornar este problema, foi feito com que as tabelas de consulta fossem distribuídas em vários métodos pelo modelo. Com isso, a rede de tabelas é composta por várias chamadas para esses métodos que retornam parte das tabelas, fazendo com que no final a rede seja montada corretamente na memória.

3.3.3 Atualização dinâmica de código

Foi possível utilizar a atualização dinâmica de código no sistema graças a estrutura extensível de carregamento de classes que o Java disponibiliza. Esta estrutura possibilita a criação de carregadores de classes customizados. Neste trabalho foi criado um carregador de classes baseado em um *array* de bytes, utilizado para carregar classes que são mandadas do validador para os monitores: o `ByteArrayClassLoader`. Ele é específico para carregar apenas a classe configurada pelos dados passados no construtor. O Quadro 27 mostra a implementação desta classe.

```
public final class ByteArrayClassLoader extends ClassLoader {
	private final String className;
	private final byte[] classBytes;

	public ByteArrayClassLoader(String className, byte[] classBytes) {
		this.className = className;
		this.classBytes = classBytes;
	}

	@Override
	protected Class<?> findClass(String name) throws
ClassNotFoundException {
		if (!className.equals(name)) {
			throw new ClassNotFoundException(name);
		}

		return defineClass(name, classBytes, 0, classBytes.length);
	}
}
```

Quadro 27 – Implementação do `ByteArrayClassLoader`

O trabalhador de criação utiliza o `ByteArrayClassLoader` para carregar as classes que lhe são passadas pelo validador. Com isso, ele pode instanciar o novo trabalhador e colocá-lo para executar. O Quadro 28 mostra o processo de criação de um novo trabalhador no monitor.

```
private AbstractWorker createNewWorker(Work work) {
	//Obtém os parâmetros
	String className = new String(work.getParameter(0));
	byte[] classBytes = work.getParameter(1);
	//Cria o ClassLoader e carrega a classe.
	ByteArrayClassLoader classLoader = new
ByteArrayClassLoader(className, classBytes);
	Class<?> workerClass = classLoader.loadClass(className);

	//Obtém o construtor do worker e instancia.
	Constructor<?> workerConstructor =
workerClass.getConstructor(InputStream.class, OutputStream.class);
	return (AbstractWorker)
workerConstructor.newInstance(serverControlInput, serverControlOutput);
}
```

Quadro 28 – Processo de criação de um novo trabalhador

```
public static ByteArrayClass generateClassToByteArray(IClassGenerator
classGenerator) {
    //Cria um arquivo temporário.
    File javaClassfile = File.createTempFile("gen", "java");

    //Cria o writer pro arquivo.
    FileWriter fileWriter = new FileWriter(javaClassfile);
    String className;
    try {
        //Gera a classe no arquivo.
        className = classGenerator.generateClass(fileWriter);
    } finally {
        fileWriter.close();
    }

    //Renomeia a classe para que ela tenha o mesmo nome da classe gerada.
    File parentFolder = javaClassfile.getParentFile();
      javaClassfile.renameTo(new File(parentFolder, className + ".java"));

    //Compila a classe.
    StringWriter logWriter = new StringWriter();
    int resultCode = Main.compile(new String[] {"-classpath", CLASS_PATH,
javaClassfile.getAbsolutePath()}, new PrintWriter(logWriter));

    //Exclui o arquivo java.
    javaClassfile.delete();

    //Se deu algum erro de compilação, lança exceçao.
    if (resultCode != 0) {
      throw new RuntimeException("Compilation error in file: " +
javaClassfile.getAbsolutePath() + "\n" +
                                              "Log: " + logWriter);
    }

    //Obtém a classe compilada.
    File compiledClassFile = new File(parentFolder, className + ".class");
    assert compiledClassFile.exists();

    //Ofusca o código da classe compilada.
    JBCO.obfuscateClass(compiledClassFile.getPath());

    //Lê os bytes da classe compilada.
    FileInputStream fileInput = new FileInputStream(compiledClassFile);
    byte[] bytes = new byte[fileInput.available()];
    try {
      fileInput.read(bytes);
    } finally {
      fileInput.close();
    }

    //Exclui a classe compilada.
    compiledClassFile.delete();

    //Retorna os bytes e o nome da classe gerada.
    return new ByteArrayClass(className, bytes);
}
```

Quadro 29 – Processo de compilação de classe gerada

Os geradores de classes facilitam a atualização dinâmica de código. Eles são utilizados para a criação de classes de trabalhadores, de algoritmo de *hashing* e de algoritmos de

criptografia. Mas simplesmente gerar uma classe não faz com que ela possa ser carregada diretamente. Ela primeiro precisa ser compilada e depois os bytes da classe compilada precisam ser obtidos. Para auxiliar neste processo, a classe `ClassGenerationUtil` foi criada. O Quadro 29 mostra o processo executado pelo método `generateClassToByteArray` desta classe a partir de um gerador de classes.

3.3.4 Gerenciamento de sessão

Quando um monitor se conecta no validador pelo canal de controle, ele é associado a uma sessão e esta sessão é gerenciada por um `ClientSessionManager`. A classe `ControlSocketServer` é quem recebe a conexão e é responsável por este processo. O Quadro 30 mostra o tratamento que as novas conexões do canal de controle recebem.

```
@Override
protected void newConnection(Socket socket) {
      //Cria a sessão.
      ClientSession session = new ClientSession();

    //Cria e inicia o manager deste client.
    ClientSessionManager sessionManager = new ClientSessionManager(socket,
session);
    sessionManager.start();

    //Se a sessão está valida, coloca no mapa de sessões válidas.
    synchronized (session) {
      if (session.isValid()) {
            validSessions.put(session.getId(), session);
      }
    }
}
```

Quadro 30 – Tratamento de novas conexões no canal de controle

O `ControlSocketServer` também é responsável por armazenar as sessões válidas. Desta forma, ele permite que os servidores dos canais de comunicação entre o software cliente e o software o consultem para aceitarem apenas conexões vindas de clientes validados. O Quadro 31 mostra o tratamento de novas conexões em um canal de comunicação entre os softwares cliente e servidor.

O gerenciador de sessão (`ClientSessionManager`) controla o envio de ordens de trabalho para o monitor e o recebimento das respostas dele. Estes trabalhos e respostas são mensagens que trafegam pelo canal de comunicação de controle estabelecido entre o monitor e o validador. As mensagens seguem um formato específico, conforme demonstrado na Figura 23.

```
@Override
protected void newConnection(Socket socket) {
	//Lê o id do cliente, que deve ser o primeiro dado enviada por um
cliente.
	UID clientId = UID.read(new
DataInputStream(socket.getInputStream()));

	ClientSession session = control.getValidSession(clientId);
	//Se não for válido, retorna.
	if (session == null) {
		return;
	}

	//Se adiciona como listener da sessão, para que quando ela for
invalidada, desconecte o servidor do client inválido.
	session.addClientSessionListener(this);

	//Cria as threads de transmissão de dados.
	ProxyThread clientServerThread = new
ProxyThread(socket.getInputStream(), server.getOutputStream());
	clientServerThread.start();
	ProxyThread serverClientThread = new
ProxyThread(server.getInputStream(), socket.getOutputStream());
	serverClientThread.start();

	//Coloca no mapa de conexões válidas.
	validConnections.put(clientId, socket);
}
```

Quadro 31 – Tratamento de novas conexões em um canal cliente e servidor

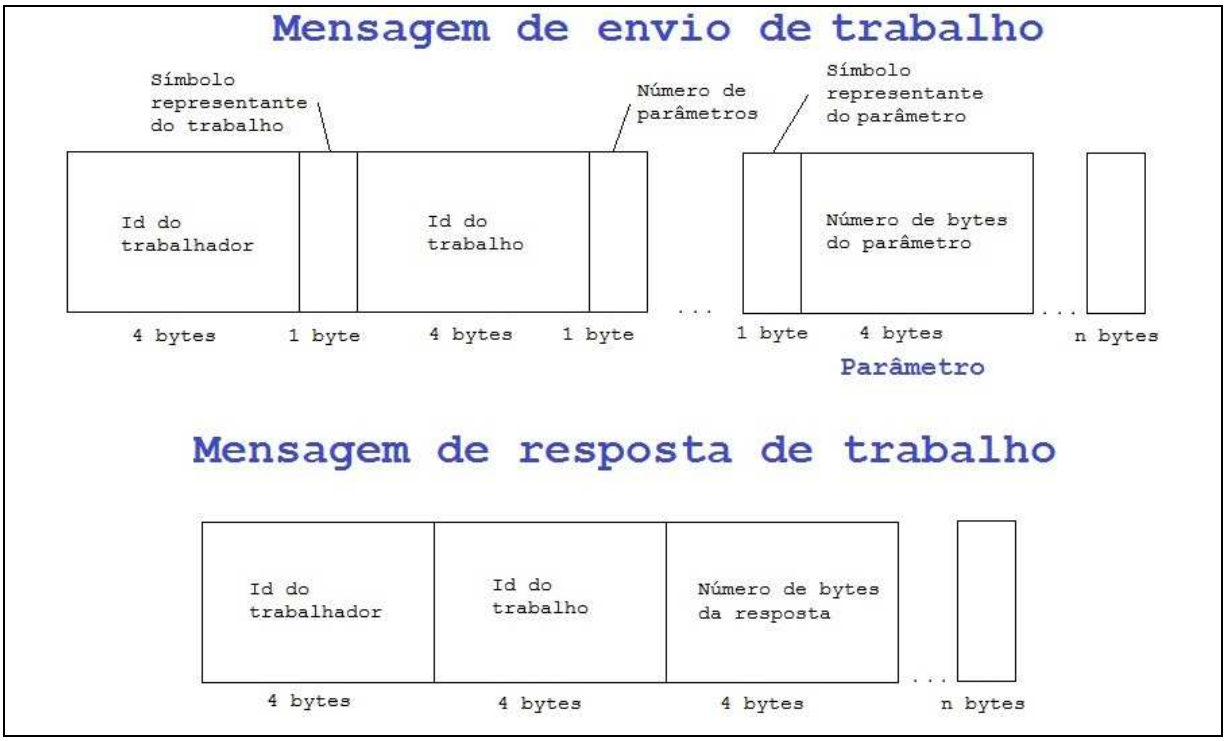


Figura 23 – Formato das mensagens de envio e resposta de trabalho

Segue o detalhamento das partes das mensagens:

a) `id do trabalhador`: identificador único do trabalhador em relação aos outros trabalhadores do mesmo monitor;

b) `símbolo representante do trabalho`: símbolo que define qual tipo de trabalho deve ser executado;

c) `id do trabalho`: identificador único do trabalho em relação aos outros trabalhos do mesmo trabalhador;

d) `número de parâmetros`: número de parâmetros do trabalho. O número máximo de parâmetros é 255. Todos os parâmetros devem ser adicionados no fim da mensagem;

e) `número de bytes do parâmetro/resposta`: o número de bytes do dado que o parâmetro/resposta possui.

O gerenciador de sessão também é responsável pela criação dos trabalhadores (através do trabalhador de criação) e de seus gerenciadores. Ele faz isto a partir do gerador de classes do trabalhador e da classe do trabalhador em si. O Quadro 32 mostra como este processo é feito.

```
private Work sendAndRegisterNewWorker(AbstractWorkerClassGenerator
workerClassGenerator, Class<? extends AbstractWorker> workerClass) {
      //Gera a classe do trabalhador.
    ByteArrayClass workerByteArrayClass =
ClassGenerationUtil.generateClassToByteArray(workerClassGenerator);

    //Envia para o client
    Work work =
creationWorkerManager.sendNewWorker(workerByteArrayClass.getClassName(),
workerByteArrayClass.getClassBytes());

    //Cria e registra o manager.
    int workerId = workerClassGenerator.getWorkerId();
      AbstractWorkerManager<?> workerManager =
WorkerManagerFactory.createWorkerManager(workerId, workerClass, session,
this);
    workersManagers.put(workerId, workerManager);

    return work;
}
```

Quadro 32 – Criação de um novo trabalhador

Por fim, o gerenciador de sessão disponibiliza meios para os gerenciadores de trabalhadores registrarem tarefas agendadas, que podem ser repetidas periodicamente. Para isto ele utiliza a classe `Timer`, que é nativa do Java. O `AbstractWorkerManager` utiliza tarefas agendadas para invalidar a sessão caso uma ordem demorar muito para receber resposta e o `ClientRuntimeValidationWorkerManager` utiliza para executar a validação periodicamente.

3.3.5 Operacionalidade da implementação

Esta seção apresenta a operacionalidade do sistema de proteção desenvolvido através de um estudo de caso. O sistema de proteção foi aplicado no jogo Ragnarök Online (GRAVITY, 2009). O software servidor utilizado foi o eAthena (EATHENA, 2010). O software cliente não oficial utilizado para testar a proteção foi um *bot* específico para este jogo, o OpenKore (OPENKORE, 2010).

Inicialmente foi testado o jogo em si sem a proteção do sistema desenvolvido. Para isto é necessário executar o servidor do jogo. O software servidor do jogo é dividido em três processos, um para cada canal de comunicação do software cliente com o servidor, conforme pode ser visto na Figura 24.

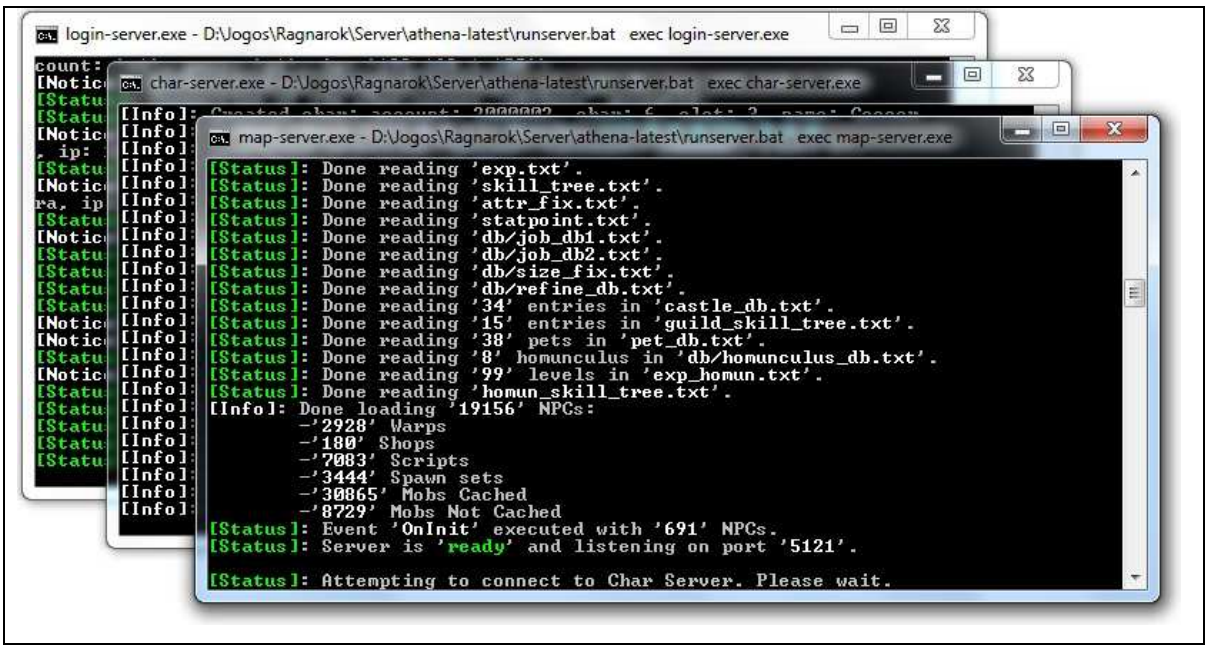


Figura 24 – Execução do software servidor de Ragnarök Online

Com o servidor no ar, é possível que o software cliente oficial se conecte e permita que o usuário jogue. A execução do software cliente oficial pode ser vista na Figura 25.

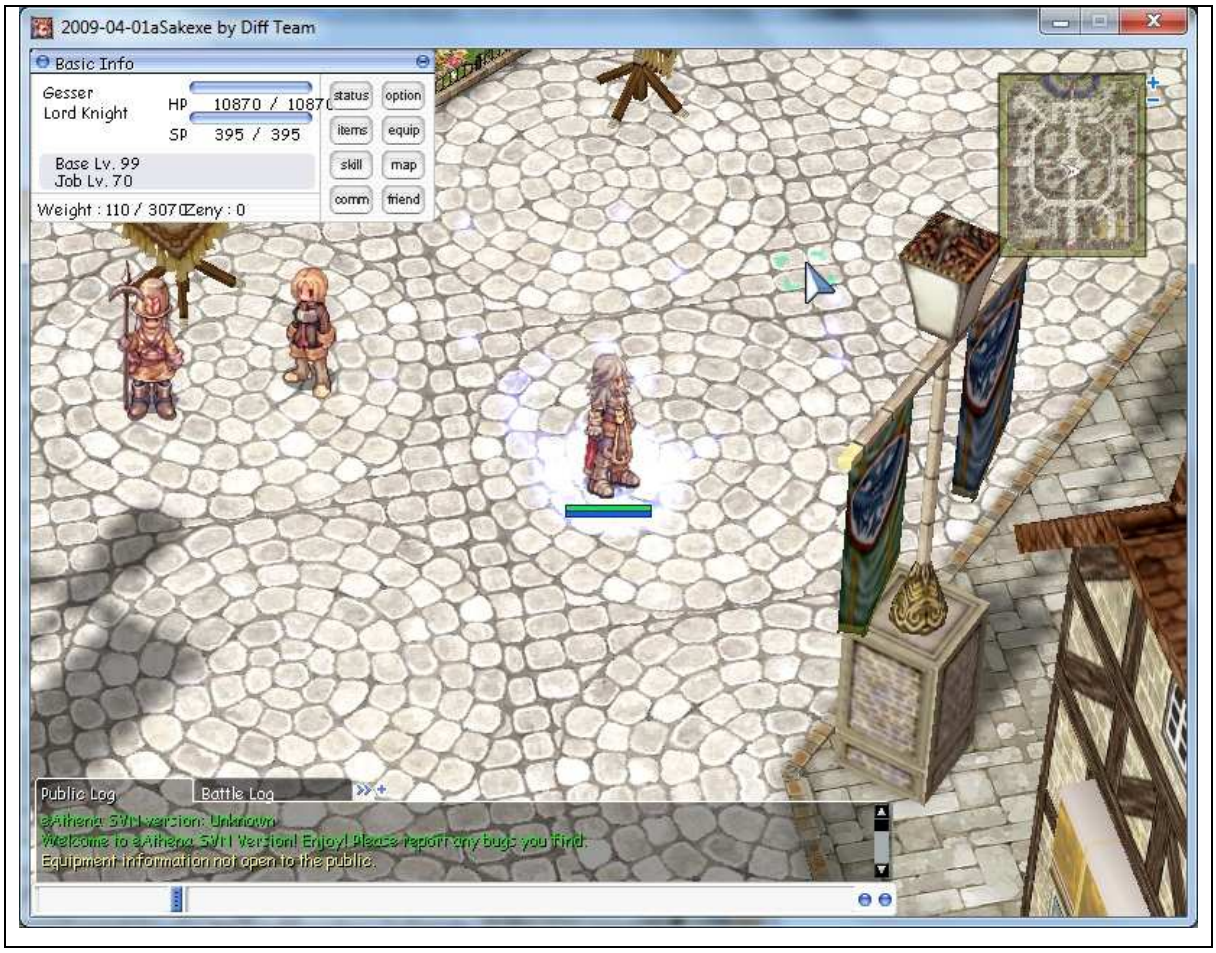


Figura 25 – Execução do software cliente oficial de Ragnarök Online

Como o servidor está desprotegido, é possível que softwares clientes não oficiais também se conectem, como o OpenKore. A execução deste *bot* pode ser vista na Figura 26.

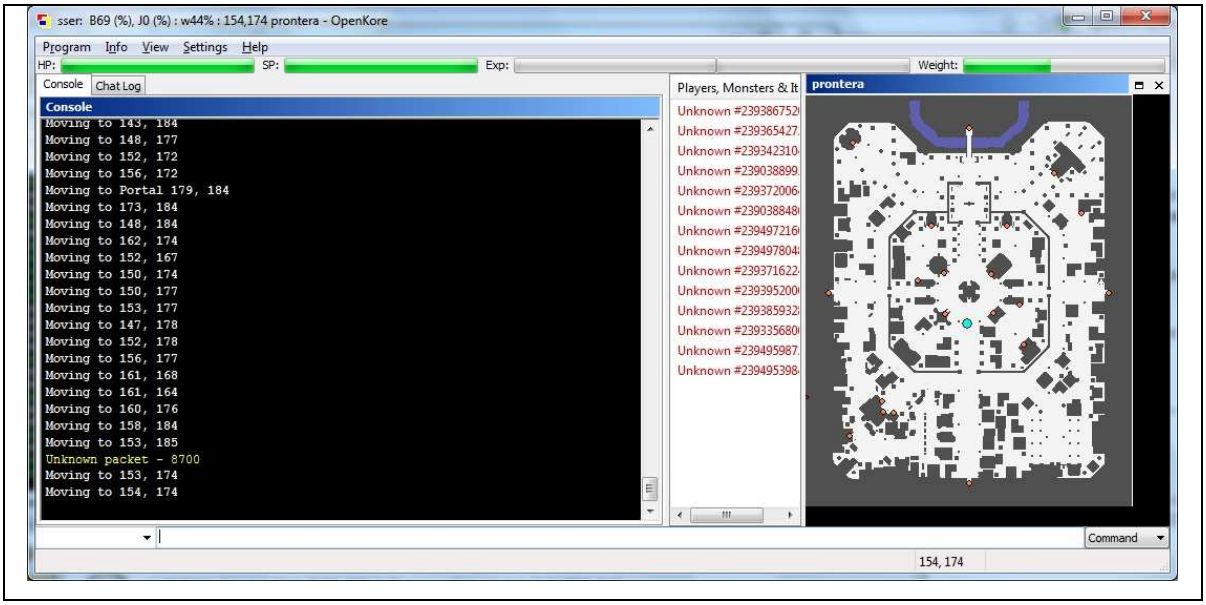


Figura 26 – Execução do software cliente não oficial de Ragnarök Online

O OpenKore permite que um jogador deixe seu personagem ser controlado de maneira automática, sem que ele tenha qualquer esforço, podendo ficar conectado 24 horas por dia.

Isto é considerada uma trapaça, pois o usuário deste *bot* leva vantagem em relação aos outros jogadores, já que eles estão dedicando seu tempo no jogo e geralmente não podem ficar jogando 24 horas por dia.

O sistema de proteção desenvolvido visa barrar este tipo de trapaça. Por isso ele foi instalado no computador servidor do Ragnarök Online. Primeiramente foi necessário configurar o componente servidor do sistema (validador) com os canais de comunicação que ele vai intermediar. O validador toma as portas que originalmente o servidor do Ragnarök Online utilizava e precisa saber em quais portas o servidor passou a atender. Também foi necessário copiar o arquivo executável do software cliente oficial para o servidor e apontar seu caminho. Estas configurações devem ser feitas no arquivo `config.json`, que está localizado juntamente com o executável do validador. O Quadro 33 mostra como as configurações foram feitas.

```
{
      clientSoftwarePath: "C:/Servidor/Ragnarok/clienteOficial.exe",
      controlPort: 123,
      serverProxies: [{
            originalServerPort: 6900
            newServerPort: 500
      }, {
            originalServerPort: 6121
            newServerPort: 501
      }, {
            originalServerPort: 5121
            newServerPort: 502
      }]
}
```

Quadro 33 – Configuração do validador

O próximo passo foi alterar a configuração do servidor do jogo para que os canais de comunicação atendam nas novas portas. A Figura 27 mostra esta alteração.

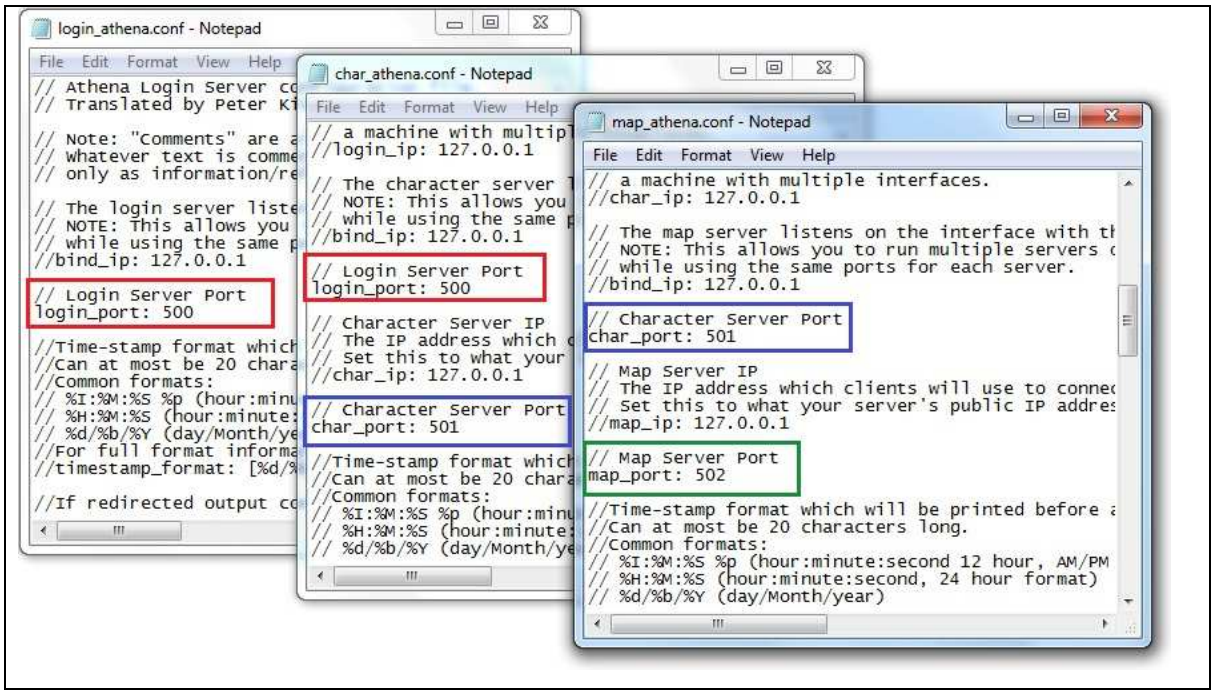


Figura 27 – Configuração das novas portas do servidor de Ragnarök Online

Com isso, foi possível executar o validador juntamente com software servidor do Ragnarök Online. Neste momento, qualquer software cliente que tentar se conectar no servidor não vai conseguir, pois é necessário primeiro que o software cliente seja validado. O único que pode fazer isto é o componente cliente do sistema (monitor). Mesmo que outros softwares tentem imitar seu comportamento, dificilmente terão sucesso, pois o monitor utiliza de criptografia de caixa branca, algoritmos de *hashing* gerados aleatoriamente e ainda tem seu código atualizado dinamicamente. Desta forma o validador pode confiar que apenas o monitor possui os meios corretos de autenticar um software cliente.

Tornou-se necessário então instalar o monitor no computador cliente. Depois de instalado, foram definidas suas configurações através do arquivo `config.json` que se localiza juntamente com o executável do monitor. A principal configuração é o caminho para o arquivo executável do software cliente. A configuração do monitor pode ser vista no Quadro 34.

```
{
      clientSoftwarePath: "D:/Jogos/Ragnarok/Ragnarok.exe",
      controlPort: 123
}
```

Quadro 34 – Configuração do monitor

Apesar de neste estudo de caso a configuração do monitor ter sido feita manualmente, o correto é ela ser preenchidas automaticamente para os usuários dos softwares clientes. Basta fazer com que o monitor seja instalado juntamente com o software cliente.

O próximo passo foi alterar a configuração do software cliente para que se conecte no monitor ao invés de diretamente no software servidor. A Figura 28 mostra esta alteração.

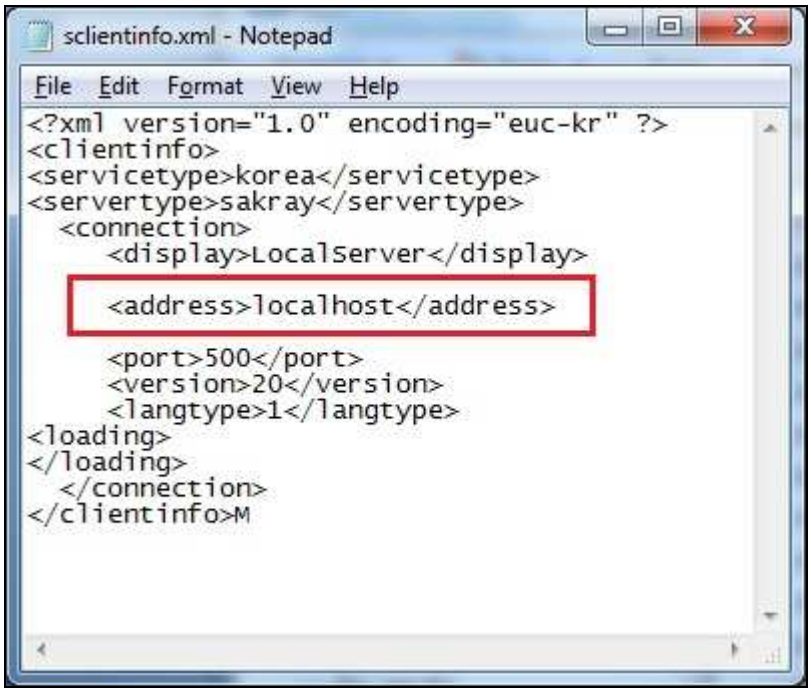


Figura 28 – Configuração do novo endereço de conexão do software cliente de Ragnarök Online

Com o monitor devidamente instalado no computador cliente, foi possível validar se ainda se conseguia jogar através do software cliente oficial. Para isto, o monitor teve que ser executado. Depois de executado, ele se conectou no validador pelo canal de comunicação de controle, que então começou o processo de validação inicial do software cliente apontado pelo monitor. Após ser validado, o software cliente foi executado através de uma ordem do validador. O software cliente então se conectou no monitor ao invés de diretamente no software servidor, que por sua vez se conectou no validador. O validador verificou se a conexão veio de um cliente validado e depois se conectou no software servidor. Desta forma os dados passam do software cliente para o software servidor e vice-versa, mas por meios de comunicação controlados pelo sistema de proteção. O resultado foi o software cliente em execução, permitindo que o usuário jogue normalmente sem nem saber da existência do monitor, da mesma forma como demonstrado na Figura 25.

A segunda validação é verificar se o sistema de proteção barra a conexão do OpenKore. Para isto, a configuração do monitor teve que ser alterada para apontar para o arquivo executável do OpenKore. Depois disso, o monitor foi executado e quando o validador fez a validação inicial, percebeu que o software cliente era inválido. Desta forma, ordenou que o monitor mostrasse uma mensagem de erro e fechou a conexão com ele. A mensagem de erro pode ser visto na Figura 29.

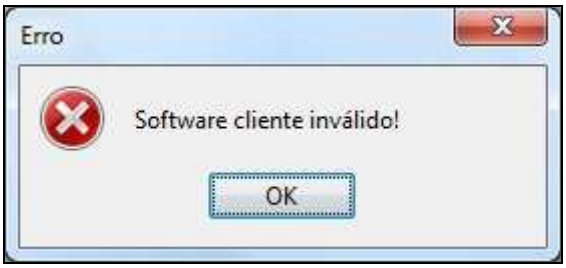


Figura 29 – Mensagem de erro mostrada quando o sistema detecta um cliente inválido

Desta forma, o software servidor do jogo Ragnarök Online tornou-se protegido contra a conexão de softwares clientes não oficiais.

3.4 RESULTADOS E DISCUSSÃO

Com o término deste trabalho pode-se verificar que é possível oferecer proteção para softwares servidores que precisam confiar nos softwares clientes que conectam neles. Isto não se limita apenas para servidores de jogos *online*, já que o sistema foi desenvolvido de maneira genérica. O sistema desenvolvido explorou diversos assuntos diferentes da área de segurança e de outras áreas para proporcionar esta proteção, por isso em alguns deles não foi possível se aprofundar o tanto quanto se poderia. Entretanto, foi criada uma base sólida para que futuras implementações possam desenvolver outras formas de proteção de maneira simplificada.

Inicialmente foi visto que as técnicas tradicionais de segurança como a assinatura digital não conseguiriam resolver os problemas de segurança enfrentados. Mas descobriu-se que existem formas de criptografar dados sem revelar a chave de criptografia, através da criptografia de caixa branca. Com isso pode ser desenvolvido o AES de caixa branca. Apesar disso, não foi contemplada toda a implementação proposta em seus estudos. Era possível esconder ainda mais as chaves através de bijeções aleatórias.

O monitor desenvolvido não possui quase nenhum código porque ele recebe os códigos que deve executar diretamente do validador. Os pontos positivos disso são os proporcionados pela atualização dinâmica de código em si e que o validador tem controle total da execução do monitor, não sofrendo com quebra de compatibilidade. O ponto negativo é que um invasor pode tentar extrair as poucas classes Java que o monitor possui para então interceptar a comunicação de atualização de código e aplicar estas atualizações em um software monitor não oficial. Para tirar vantagem desta situação, o invasor teria que modificar o código que recebe das atualizações para, por exemplo, conseguir executar um software

cliente não oficial. Entretanto, todo código enviado pelo validador é previamente ofuscado, o que dificulta muito modificações deste tipo. De qualquer forma, este ainda é um ponto que talvez possa receber mais segurança

O sistema possui estruturas que permitem que sejam feitas verificações no ambiente de execução e no estado do software cliente em execução. Estas verificações podem ser feitas juntamente com a validação inicial ou as validações periódicas. Com elas, pode-se prevenir que algum invasor altere o ambiente de execução ou a execução do software cliente oficial para gerar efeitos similares aos de uma adulteração direta no código do software cliente.

A estrutura de trabalhadores do sistema desenvolvido permite que o validador crie qualquer lógica de validação sem a necessidade de alterar o monitor. Desta forma, futuras implementações podem criar trabalhadores para executar validações adicionais ou qualquer outra ação importante.

4 CONCLUSÕES

Os jogos *online* apresentam uma nova perspectiva de diversão. Independente da distância física entre os jogadores, eles podem vivenciar aventuras, compartilhar emoções, como se estivessem jogando um ao lado do outro. Por isso, este tipo de jogo se tornou muito famoso, conquistando milhões de jogadores no mundo todo. Mas como na maioria dos jogos, também há como de trapacear nos jogos *online*. E como na maioria dos jogos de computador, a exploração destas trapaças é possível graças a problemas de segurança. Um dos principais problemas de segurança de jogos *online* é a possibilidade de se alterar o software cliente do jogo ou de criar um software cliente alternativo (como um *bot*), pois isto pode proporcionar vantagens em relação aos demais jogadores. Para evitar isto, o software servidor do jogo *online* deve se comunicar apenas com softwares clientes oficiais, que foram disponibilizados pela empresa do jogo. Mas isto explicita um problema mais amplo, que não envolve apenas servidores de jogos *online*, mas sim de qualquer aplicação que segue a arquitetura cliente/servidor: a confiabilidade em softwares executados em ambientes remotos não confiáveis. Este foi de fato o problema que este trabalho visou resolver.

Para isso, foram estudadas diversas técnicas tradicionais de segurança para garantir a autenticidade de pessoas e de softwares. As técnicas de autenticidade de pessoas não se aplicam no problema estudado, pois o software servidor lida com outros softwares. As técnicas autenticidade de softwares dependem principalmente da confiabilidade entre as partes. Por exemplo, pode-se fazer o *download* de um software assinado digitalmente. Através da assinatura é possível garantir por quem o software foi criado. Com isso, é possível decidir se o criador do software é confiável ou não antes de executá-lo.

O problema estudado por este trabalho lida com um cenário diferente, em que um software servidor se comunica com softwares clientes não confiáveis. Eles não são confiáveis porque o software servidor apenas se comunica com eles, não sabendo qual software de fato está sendo executado no computador cliente. Por isso, o sistema de proteção desenvolvido visou estender o controle do servidor para o computador cliente, para que ele próprio garanta que o software executado no computador cliente é confiável.

Foram desenvolvidos dois componentes para o sistema:

a) Monitor: monitora o computador cliente e o software cliente executado, seguindo as ordens vindas do servidor;
b) validador: controla os monitores e executa as validações dos clientes a partir das

informações monitoradas.

A partir das validações é possível garantir a confiabilidade em determinado cliente. Estes componentes intermediam a comunicação entre o software servidor e os softwares clientes, para garantir que o software servidor só se comunicará com softwares clientes válidos.

O validador precisa garantir que apenas se comunica com monitores confiáveis, já que um invasor pode tentar criar um software que imite o comportamento do monitor e assim passar informações falsas para o validador. Para isso, foram incorporadas no sistema técnicas de criptografia de caixa branca, algoritmos de *hashing* gerados aleatoriamente, ofuscação e atualização dinâmica de código. Desta forma, apenas o monitor desenvolvido neste trabalho sabe como responder a determinadas mensagens de validação vindas validador. Um invasor pode até tentar descobrir a chave de criptografia utilizada, mas ela estará bem escondida na implementação de AES de caixa branca. O invasor também tentar analisar o código do monitor para copiá-lo, mas seu código é ofuscado e atualizado periodicamente. Garantindo assim pouco tempo para que se execute uma invasão bem sucedida.

Diante do exposto, constata-se que os três objetivos deste trabalho foram concluídos com êxito.

4.1 EXTENSÕES

Como trabalhos de extensão do sistema de proteção sugerem-se:

a) integrar as bijeções aleatórias no gerador de AES de caixa branca;
b) criar outros mecanismos de codificação aleatória para utilização no gerador de AES de caixa branca;
c) estudar as fraquezas do algoritmo de AES de caixa branca e buscar soluções para elas;
d) criar outros filtros aleatórios para os algoritmos de *hashing* gerados aleatoriamente;
e) aplicar o sistema de proteção em outros jogos *online* ou outros softwares, verificando há impacta no tempo de resposta do software cliente com o software servidor;
f) implementar verificações do ambiente de execução e do estado do software cliente

em execução;

g) implementar novos trabalhadores para a execução de novas validações no cliente;
h) colocar o sistema de proteção a prova, utilizando qualquer técnica de invasão conhecida;
i) identificar possíveis falhas de segurança no sistema e propor soluções para elas;
j) fazer análises mais profundas em outras ferramentas de segurança, comparando os pontos negativos e positivos com o sistema desenvolvido;
k) aplicar o sistema de segurança em sistemas *web*, que no caso teriam como software cliente um *browser*.

REFERÊNCIAS BIBLIOGRÁFICAS

AHNLAB. **HackShield**: hacking prevention solutions for online games. [S.l.], 2009. Disponível em: <http://www.hackshields.com/product.html>. Acesso em: 28 mar. 2010.

ALECRIM, Emerson. **Ataques DoS (Denial of Service) e DDoS (Distributed DoS)**. [S.l.], 2004. Disponível em: <http://www.infowester.com/col091004.php>. Acesso em: 29 mar. 2010.

BRANDÃO, Roberto P. **SSL**. Rio de Janeiro, 2003. Disponível em: <http://www.gta.ufrj.br/grad/00_2/ssl/ssl.htm>. Acesso em: 28 mar. 2010.

BURNETT, Steve; PAINE, Stephen. **Criptografia e segurança**: o guia oficial RSA. Tradução Edson Furmankiewicz. Rio de Janeiro: Elsevier, 2002.

CARNEGIE MELLON UNIVERSITY. **The official Captcha site**. [Pittsburgh], 2009. Disponível em: <http://www.captcha.net>. Acesso em: 26 mar. 2010.

CHOI, Hyun-Jin; YAN, Jianxin J. Security issues in online games. **The Electronic Library**, Bingley, v. 20, n. 2, p. 125-133, ago. 2002.

CHOW, Stanley et al. White-Box Cryptography and an AES Implementation. In: INTERNATIONAL WORKSHOP ON SELECTED AREAS IN CRYPTOGRAPHY, 9., 2002, St. John's. **Proceedings…** St. John's: Springer, 2002. p. 250-270.

EATHENA. **eAthena support board**. [S.l.], 2010. Disponível em: <http://eathena.ws>. Acesso em: 25 out. 2010.

EXCELSIOR. **Solid Java code protection**. [S.l.], 2010. Disponível em: <http://www.excelsior-usa.com/jetprotection.html>. Acesso em: 5 out. 2010.

GAMEFORT. **Game Fort**: hacking prevention system. [S.l.], 2010. Disponível em: <http://www.gamefort.com.br>. Acesso em: 28 mar. 2010.

GARFINKEL, Simson; SPAFFORD, Gene. **Web security, privacy & commerce**. 2nd ed. Sebastopol: O'Reilly & Associates, 2002.

GRIMEN, Gisle; MIDTSTRAUM, Roger; MÖNCH, Christian. Tamper protection of online clients through random checksum algorithms. In: INTERNATIONAL CONFERENCE ISTA, 5., 2006, Klagenfurt. **Proceedings…** Bonn: GI-Edition, 2006. p. 67-79.

GRAVITY. **Ragnarök Online**. [S.l.], 2009. Disponível em: <http://www.ragnarokonline.com>. Acesso em: 25 out. 2010.

GURTNER, Christian. **Criptografia para leigos:** parte 3. [S.l.], 2007. Disponível em: <http://cristiantm.wordpress.com/2007/07/25/criptografia-para-leigos-parte-3>. Acesso em: 30 mar. 2010.

FALEIROS, Antônio C.; ROSA, Rafael A. S. **Análise do algoritmo vencedor do AES**: o Rijndael. São José dos Campos: Instituto Tecnológico de Aeronáutica (ITA), 2003. Disponível em: <http://www.bibl.ita.br/ixencita/artigos/FundRafaelAntonio1.pdf>. Acesso em: 20 out. 2010.

INTRODUÇÃO às Redes de Computadores/Protocolos de aplicação – princípios gerais. In: WIKIVERSIDADE, a universidade livre. [S.l.]: Wikimedia Foundation, 2010. Disponível em: <http://pt.wikiversity.org/wiki/Introdução_às_Redes_de_Computadores/Protocolos_de_aplicação_–_princípios_gerais>. Acesso em: 27 set. 2010.

JOGOS on-line. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2010. Disponível em: <http://pt.wikipedia.org/wiki/Jogo_on-line>. Acesso em: 27 mar. 2010.

KRIEF, Francine; M'BARKA, Moez B.; LY, Oliver. Entrusting remote software executed in an untrusted computation helper. In: N2S/IEEE INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE SECURITY, 1., 2009, Paris. **Proceedings…** Paris: ESRGroups France, 2009. p. 1-5.

LESKOV, Dmitry. **Protect your Java code:** through obfuscators and beyond. [S.l.], 2010. Disponível em: <http://www.excelsior-usa.com/articles/java-obfuscators.html>. Acesso em: 5 out. 2010.

MOREIRA, André. **Criptografia**. Porto, 2002. Disponível em: <http://www.dei.isep.ipp.pt/~andre/documentos/criptografia.html>. Acesso em: 26 mar. 2010.

NPROTECT. **nProtect global site**. [S.l.], 2010. Disponível em: <http://global.nprotect.com/product/gg.php>. Acesso em: 28 mar. 2010.

OPENKORE. **OpenKore wiki**. [S.l.], 2010. Disponível em: <http://www.openkore.com>. Acesso em: 25 out. 2010.

PALOV, Igor. **LZMA SDK**. [S.l.], 2009. Disponível em: <http://www.7-zip.org/sdk.html>. Acesso em: 10 nov. 2010.

PIROPO, Benito. **Atributos digitais II**: assinatura digital. [S.l.], 2009. Disponível em: <http://www.bpiropo.com.br/fpc20071210.htm>. Acesso em: 29 set. 2010.

PLASMANS, Marjanne. **White-box cryptography for digital content protection**. 2005. 82 f. Dissertação (Mestrado em Matemática Aplicada) - Departametno de Matemática e Ciência da Computação, Universidade de Tecnologia de Eindhoven, Eindhoven.

RAGNARÖK online. In: WIKIPÉDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2010. Disponível em: <http://pt.wikipedia.org/wiki/Ragnar%C3%B6k_Online>. Acesso em: 27 mar. 2010.

RAJ, Ashok. **AES java implementation**. [S.l.], 2004. Disponível em: <http://www.derkeiler.com/Newsgroups/sci.crypt/2004-03/1353.html>. Acesso em: 30 out. 2010.

SABLE. **JBCO:** the Java ByteCode obfuscator. [Montreal], 2010. Disponível em: <http://www.sable.mcgill.ca/JBCO>. Acesso em: 5 out. 2010.

SILVA, Lino S. **Public key infraestructure**: PKI. São Paulo: Novatec Editora, 2004.

STADLER, Lukas; WIMMER, Christian; WÜRTHINGER, Thomas. Dynamic code evolution for Java. In: INTERNATIONAL CONFERENCE ON THE PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 8., 2010, Vienna. **Proceedings…** Nova Iorque: ACM, 2010. p. 10-19.

YANG, Li. **Add dynamic Java code to your application**. [S.l.], 2006. Disponível em: <http://www.javaworld.com/javaworld/jw-06-2006/jw-0612-dynamic.html>. Acesso em: 21 out. 2010.

WYSEUR, Brecht. **White-box cryptography**. 2009. 205 f. Tese (Doutorado em Engenharia Elétrica) - Departamento de Engenharia Elétrica, Universidade Católica de Lovaina, Lovaina.