Segurança de Sistemas de Agentes Móveis

Nelson Uto

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Segurança de Sistemas de Agentes Móveis

Nelson Uto

Abril de 2003

Banca Examinadora:

- Prof. Dr. Ricardo Dahab
- Prof. Dr. Joni da Silva Fraga
- Prof. Dr. Paulo Lício de Geus
- Prof. Dr. Ricardo Anido

Segurança de Sistemas de Agentes Móveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Nelson Uto e aprovada pela Banca Examinadora.

Campinas, 10 de abril de 2003.

Prof. Dr. Ricardo Dahab

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Resumo

Um agente móvel é um programa que realiza uma tarefa de forma autônoma em nome de um usuário e que pode migrar para os servidores que forneçam os recursos necessários para a realização da tarefa delegada. Servidores de agentes fornecem um ambiente de execução para agentes e oferecem diversos serviços para migração, comunicação e acesso a recursos como bancos de dados, por exemplo. O conjunto de agentes, servidores de agentes e componentes auxiliares como diretório de nomes e uma terceira parte confiável forma uma infra-estrutura que suporta o paradigma de agentes móveis e é chamada de **sistema de agentes móveis**. A mobilidade de código presente nestes sistemas introduz novos problemas de segurança que devem ser considerados.

Esta dissertação apresenta um estudo detalhado dos aspectos de segurança de sistemas de agentes móveis, compreendendo as classes de ameaças existentes, os requisitos de segurança e os mecanismos propostos para solucionar ou amenizar tais ameaças. Uma vez que a linguagem Java é comumente utilizada para desenvolver tais sistemas, a arquitetura de segurança da plataforma Java é analisada com o objetivo de se entender como os mecanismos utilizados podem ser úteis para a segurança destes sistemas.

Diversos sistemas de agentes móveis, acadêmicos e comerciais, são analisados neste trabalho, com ênfase nas arquiteturas de segurança providas por cada um deles. As fraquezas encontradas são relatadas e melhorias, propostas. Para o sistema de agentes móveis Aglets, são adicionados novos mecanismos de segurança.

Abstract

A mobile agent is a program which executes tasks on behalf of a user and can autonomously migrate from server to server, searching for the necessary resources to complete the delegated tasks. Agent servers provide an execution environment and resources to agents, as well as several services for supporting migration and communication, among others activities. Agents, agent servers and auxiliary components such as a trusted third party form an infrastructure called **mobile agent system**. The code mobility in these systems raises new security issues that should be addressed.

This dissertation presents a detailed account of the security aspects of mobile agent systems, encompassing existing threats, security requirements and the corresponding mechanisms proposed in the literature to counter these threats. Since Java is one of the languages of choice for the development of such systems, the security architecture of the Java platform is analyzed, in order to understand its impact on the security of mobile agent systems.

As for specific systems, security mechanisms present in several proposals, academic and comercial, are detailed, pointing weaknesses and proposing strengthenings. For one system, Aglets, new, additional security features are presented and implemented.

Agradecimentos

Gostaria de agradecer a todas as pessoas que, direta ou indiretamente, contribuíram para que esta dissertação de mestrado se concretizasse:

- a Deus;
- ao meu orientador, Prof. Dr. Ricardo Dahab, que me guiou durante todo este tempo por esta vereda e me introduziu no mundo maravilhoso da Criptografia; além disso, sou-lhe grato pelas portas que me abriu durante estes anos de convivência;
- ao Prof. Dr. Cid Carvalho de Souza, que foi meu tutor no início deste mestrado e que me ajudou a convencer meu orientador a me aceitar como aluno;
- ao CNPq, pelos meses iniciais de bolsa de estudos;
- a Bosch, pelo auxílio financeiro na forma de bolsa de estudos;
- à minha família, por tudo;
- à Sandra Regina Rocha, por todos os belos momentos que passamos juntos; nunca me esquecerei dos dias 13/06/2000 e 27/07/2000...;
- aos meus grandes amigos Silvio Santana e Fernando Galdino, pelos anos da mais sincera amizade;
- ao pessoal do xadrez de Salto, pelas inúmeras horas de "pings" e torneios que participamos lado-a-lado;
- à Caíssa;
- aos amigos do Instituto de Computação, estudantes e funcionários.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Classes

Capítulo 1

Introdução

Mobilidade de código não é um conceito novo e tem suas origens na década de 60, com a submissão de trabalhos em lotes para serem executados em *mainframes*. Desde então, diversos outros exemplos que empregam códigos móveis surgiram, como o comando `rsh` do sistema Unix e a linguagem PostScript. Mas foi somente com a explosão da Internet e com o sucesso da linguagem Java, na forma de *applets* para a dinamização das páginas HTML, que a mobilidade de código foi trazida ao dia-a-dia de um grande número de usuários. Diversos paradigmas foram propostos para explorar códigos móveis e um deles, o paradigma de agentes móveis, é a base dos sistemas explorados neste trabalho.

Um agente móvel é um programa que realiza uma tarefa de forma autônoma em nome de um usuário e que pode migrar para os servidores que forneçam os recursos necessários para a realização da tarefa delegada. Servidores de agentes fornecem um ambiente de execução para agentes e oferecem diversos serviços para migração, comunicação e acesso a recursos como bancos de dados, por exemplo. O conjunto de agentes, servidores de agentes e componentes auxiliares como diretório de nomes e uma terceira parte confiável forma uma infra-estrutura que suporta o paradigma de agentes móveis e é chamada de **sistema de agentes móveis**.

O uso de agentes móveis no desenvolvimento de aplicações distribuídas tem se mostrado atraente porque reúne, em um único *framework*, diversas vantagens como redução do tráfego de rede, redução na latência e execução assíncrona e autônoma. De modo geral, toda aplicação baseada em agentes móveis possui soluções igualmente eficientes utilizando-se outros paradigmas. Porém, estas soluções alternativas requerem uma combinação de técnicas que é específica a cada problema, enquanto que o paradigma de agentes móveis proporciona uniformidade na solução de uma grande gama de aplicações [38, 67, 28].

O advento destes sistemas trouxe também um conjunto de novas ameaças, específicas a este paradigma, cuja descrição detalhada será objeto de capítulos posteriores. Esta dissertação trata dos aspectos de segurança de sistemas de agentes móveis, abordando

os diversos tipos de ameaças, os requisitos de segurança e os mecanismos de proteção propostos na literatura e implementados por alguns sistemas.

1.1 Motivação

Um exemplo clássico utilizado para ilustrar alguns dos problemas de segurança que afetam os sistemas de agentes móveis é o do agente para viagens [17]. Neste exemplo, Alice deseja comprar uma passagem aérea para uma cidade que ela quer conhecer e, para tanto, utiliza as funcionalidades fornecidas por uma aplicação baseada em agentes móveis. Como ela é uma pessoa viajada e sempre usa esta aplicação para obter suas passagens, o sistema já possui muitas das informações que compõem seu perfil, como restrições alimentares e preferência de assentos, que não precisam ser novamente fornecidas. Enquanto viaja de táxi até a cidade onde se localiza o aeroporto, a partir de seu PDA, Alice configura o agente com o destino e horário de vôo e o despacha para realizar a compra. Note que dada a natureza assíncrona e autônoma dos agentes móveis, a qualidade de conexão não se torna um empecilho, pois após enviado, o agente realiza tudo por conta própria, sem a necessidade de interagir com Alice.

O agente móvel inicialmente visita um diretório de serviços, onde obtém uma lista dos servidores de companhias aéreas. Com base nessas informações, traça um itinerário e começa a visitar cada um dos servidores disponíveis, em busca da melhor oferta para o destino especificado por Alice e que esteja de acordo com as suas preferências. Para isso, em cada servidor visitado, o agente pesquisa o banco de dados de vôos e armazena o valor cobrado para o destino de interesse. Ao final do itinerário, o agente compara os preços coletados e faz uma reserva na companhia que possui o valor mais atraente. O agente então retorna ao PDA de Alice, para informá-la da compra efetuada e, no caso de ela não estar conectada, espera em algum servidor até que ela se reconecte à rede.

Dada a natureza competitiva entre as companhias aéreas, não se pode esperar que elas confiem umas nas outras. Assim, diversos tipos de ataques podem ocorrer nesse cenário. Por exemplo, uma companhia maliciosa pode aumentar os valores armazenados no agente, que foram coletados nos servidores das companhias concorrentes, favorecendo-a no processo de escolha. De modo análogo, pode-se ajustar o preço da viagem de acordo com a melhor oferta contida no estado do agente. Também é possível alterar o número de reservas que o agente deve realizar, para lotar um vôo de uma empresa concorrente. Por outro lado, os servidores das companhias aéreas também estão sujeitos a ataques perpetrados pelos agentes móveis. Um agente malicioso pode roubar dados confidenciais ou alterar o número de vagas nos vôos da companhia.

A mobilidade de código, presente no paradigma de sistemas de agentes móveis, introduz novos problemas de segurança tanto para o agente móvel como para o servidor

[18, 104]. Por um lado, os programas (agentes móveis) deixam de ser executados apenas em uma máquina na qual se confia (a do proprietário do agente) e passam a ser executados em diversas máquinas, algumas das quais não confiáveis. Por outro lado, os servidores devem receber e executar agentes escritos por terceiros muitas vezes desconhecidos. É importante ressaltar que muitos dos ataques possíveis nestes sistemas, como *eavesdropping* e negação de serviço, possuem equivalentes em sistemas cliente-servidor tradicionais [36].

Com tudo isso, fica clara a necessidade de se fornecer mecanismos de segurança para conter os ataques que podem ocorrer nesses ambientes. Conforme Chess *et al.* [7], enquanto a disponibilidade de um forte aparato de segurança não seria suficiente para uma ampla adoção de sistemas de agentes móveis, a ausência de mecanismos de segurança certamente tornaria tais sistemas muito pouco atraentes.

1.2 Objetivos do trabalho

O principal objetivo desta dissertação é apresentar um estudo detalhado dos aspectos de segurança dos sistemas de agentes móveis, compreendendo as classes de ameaças existentes, os requisitos de segurança e os mecanismos propostos para solucionar ou amenizar tais ameaças. Uma vez que a linguagem Java é comumente utilizada para desenvolver tais sistemas, a arquitetura de segurança da plataforma Java deve ser estudada para se entender como os mecanismos utilizados podem ser úteis para a segurança destes sistemas. Também devem ser analisados os mecanismos de segurança disponíveis nos diversos sistemas de agentes móveis, acadêmicos e comerciais, em busca de fraquezas e possíveis melhorias. Por fim, serão implementados alguns mecanismos de segurança para complementar a arquitetura de segurança do sistema Aglets.

Outro objetivo é escrever uma dissertação que sirva de introdução e referência àqueles que desejem aprofundar-se no estudo da segurança de sistemas de agentes móveis. Assim, a intenção é redigir um texto relativamente auto-contido e suficientemente abrangente, cobrindo a maior parte dos tópicos desta área de pesquisa. Ao longo da dissertação, uma grande quantidade de referências será fornecida para que o leitor possa se aprofundar em tópicos específicos.

1.3 Convenções tipográficas

As seguintes convenções tipográficas são utilizadas nesta dissertação:

- *itálico* é utilizado para escrever termos em inglês e nomes de entidades;
- **negrito** é usado para introduzir conceitos e ressaltar pontos importantes; e

- `texto em fonte monoespaçada` é utilizado para nomes de classes, objetos, trechos de código, comandos e nomes de domínio.

1.4 Organização da dissertação

A seguir resumimos os demais capítulos desta dissertação.

Capítulo 2 – Sistemas de Agentes Móveis
Este capítulo conta um pouco da história da mobilidade de código e descreve o modelo de Vigna e Picco para tecnologias de códigos móveis. Com base neste modelo, discutimos mobilidade fraca e forte e os paradigmas de projeto existentes. Um modelo simples de sistemas de agentes móveis é apresentado, bem como o modelo conceitual comum da OMG. Também são listadas as vantagens destes sistemas e dados alguns exemplos de aplicações empregando esta tecnologia.

Capítulo 3 – Segurança de Sistemas de Agentes Móveis
Este capítulo apresenta um modelo simplificado de sistema de agentes móveis, que é utilizado para classificar os diversos ataques e ameaças possíveis nestes ambientes. O capítulo também discute os requisitos de segurança desejáveis em tais sistemas e descreve os diversos mecanismos para proteção do servidor e do agente móvel propostos na literatura.

Capítulo 4 – Arquitetura de Segurança de Java
Este capítulo apresenta a arquitetura de segurança da plataforma Java. Primeiro, mostramos o modelo de segurança original, conhecido como *sandbox*, e a evolução do modelo ao longo das grandes versões. Em seguida, os elementos que compõem a *sandbox* básica e as adições ocorridas na versão 2 da plataforma Java são tratados em detalhes. O capítulo também aborda os pontos não contemplados pela arquitetura de segurança e mostra alguns ataques baseados em *bugs* que afetavam versões mais antigas da plataforma. O capítulo é concluído ilustrando como os mecanismos de segurança de Java podem ser usados na segurança de sistemas de agentes móveis.

Capítulo 5 – Um *Survey* sobre Sistemas de Agentes Móveis
Neste capítulo são descritos e analisados alguns sistemas de agentes móveis, com ênfase nos aspectos de segurança. Cada análise é estruturada em quatro partes: informações gerais, descrição do sistema, aspectos de segurança e comentários. Ao final, um resumo das principais características dos sistemas é apresentado em forma de quadros sinóticos. Uma versão mais antiga deste capítulo foi publicada nos anais do 3º Simpósio Segurança em Informática [106] e como um relatório técnico [105].

Capítulo 6 – Extensões aos Mecanismos de Segurança do Sistema Aglets
Este capítulo aborda os mecanismos adicionados à arquitetura de segurança do sistema de agentes móveis Aglets, como resultado deste trabalho.

Capítulo 7 – Conclusões
Este capítulo conclui o trabalho e lista as principais contribuições desta dissertação.

Apêndice A – Fundamentos
Este apêndice apresenta uma breve introdução à segurança da informação, listando alguns de seus objetivos, e mostra as principais primitivas fornecidas pela Criptografia, bem como alguns protocolos.

Apêndice B – Outros Tópicos sobre Segurança em Java
Este apêndice trata de tópicos sobre segurança em Java que não foram cobertos no Capítulo 4. De modo geral, os tópicos abordados estão mais relacionados à implementação de sistemas com características de segurança do que com a arquitetura de segurança da plataforma Java propriamente dita.

Apêndice C – Alguns Aglets Maliciosos
Este apêndice lista o código-fonte de dois aglets maliciosos que foram facilmente adaptados a partir de exemplos desenvolvidos para explorar fraquezas do modelo de segurança da plataforma Java.

Apêndice D – Acrônimos

Apêndice E – Glossário

Apêndice F – Notação

Capítulo 2

Sistemas de Agentes Móveis

Este capítulo está organizado da seguinte forma: a Seção 2.1 apresenta um pouco de história sobre mobilidade de código e introduz os elementos do modelo de Vigna e Picco [109, 81] para tecnologias de códigos móveis. Com base neste modelo, são discutidos os tipos de mobilidade e os paradigmas de projeto existentes. A Seção 2.2, descreve os componentes de um sistema de agentes móveis a partir de um modelo simples, enquanto que o modelo conceitual comum da OMG é dado na Seção 2.3, junto com os conceitos relevantes. As vantagens apresentadas por estes sistemas são listadas na Seção 2.4 e, por fim, alguns exemplos de aplicações empregando agentes móveis são dados na Seção 2.5

2.1 Mobilidade de código

2.1.1 Breve histórico

A mobilidade de código permite que um trecho de código seja enviado pela rede para ser executado em um computador remoto. Este conceito originou-se na década de 60 com a linguagem JCL (Job Control Language), que permitia submeter, a partir de minicomputadores, trabalhos em lotes para serem executados em *mainframes* [8]. Outro exemplo introduzido em 1984 é o comando `rsh` do sistema operacional Unix que possibilita que um usuário envie um *shell script* para ser executado em uma máquina remota.

Um uso menos evidente, porém corriqueiro, de mobilidade de código é dado pela linguagem PostScript, introduzida em 1985 pela Adobe Systems Inc. e considerada uma linguagem de descrição de página. O propósito principal da linguagem é fornecer um meio conveniente de descrever imagens independentemente de dispositivo. Impressoras compatíveis com a linguagem recebem um conjunto de comandos que são processados para montagem da página a ser impressa.

Em 1995, a linguagem Java, na forma de *applets*, foi introduzida para adicionar

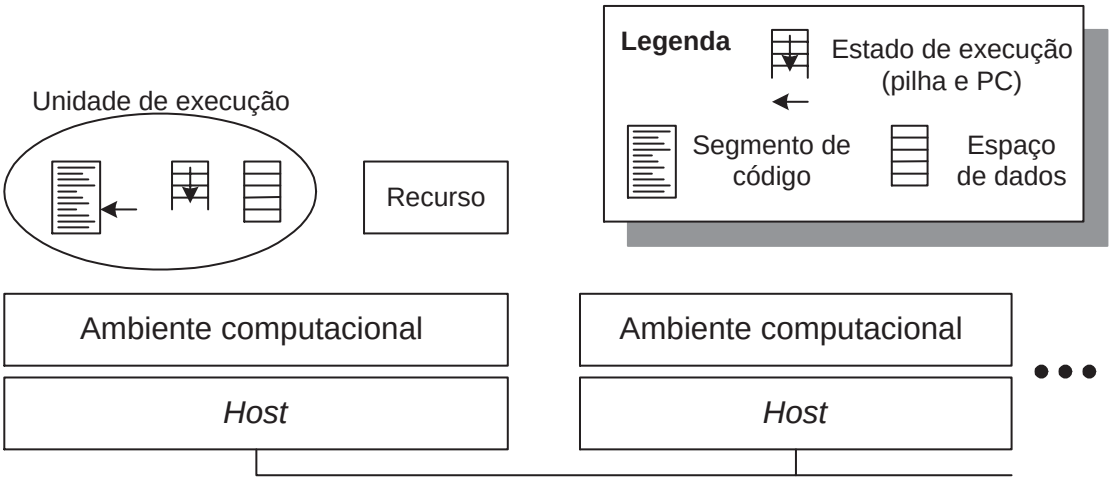

Figura 2.1: Componentes de tecnologias que suportam mobilidade de código.

conteúdo dinâmico a páginas Web. Os *applets* eram embutidos em páginas HTML que, ao invés de exibir apenas gráficos e textos estáticos, ganharam vida com animações, áudio e interatividade. Em seguida, Java tornou-se uma linguagem de propósito geral e serviu de base para a implementação de diversos sistemas de agentes móveis.

Na mesma época, a General Magic Inc. desenvolveu e lançou o sistema de agentes móveis Telescript (Seção 5.1). A linguagem usada para o desenvolvimento de aplicações para este sistema também era chamada de Telescript e a necessidade de aprendizagem de uma linguagem completamente nova foi um dos motivos de fracasso do produto.

2.1.2 Elementos de tecnologias de códigos móveis

Esta seção apresenta os elementos existentes em tecnologias que suportam mobilidade de código, conforme modelo de Vigna e Picco [109, 81]. O modelo é ilustrado na Figura 2.1 e será utilizado para descrever as formas de mobilidade e os paradigmas de códigos móveis nas Seções 2.1.3 e 2.1.4, respectivamente.

O **ambiente computacional (AC)** tem como propósito permitir que aplicações realoquem dinamicamente seus componentes em diferentes *hosts* [109, 81]. Assim, além de servir de ambiente de execução para os aplicativos, um AC utiliza os canais de comunicação da camada de rede para fornecer serviços de relocação de código e estado para outros computadores da rede. Um ponto importante é que o AC explicita a identidade do *host* no qual está localizado. Dessa forma, o uso de serviços requer o conhecimento dos nós da rede que os fornecem.

Os componentes hospedados pelos ACs são divididos em **unidades de execução (UEs)** e **recursos**. Uma UE é composta de código, espaço de dados e estado de execução.

Este último compreende a pilha de chamadas e o *program counter*, enquanto que o espaço de dados é o conjunto de referências a componentes que podem ser acessados pela UE. Como exemplos de UEs podemos citar processos e *threads* individuais. Vale notar que no contexto desta dissertação uma UE corresponde a um agente móvel. Por fim, os recursos são entidades que podem ser compartilhadas pelas UEs como, por exemplo, um arquivo qualquer ou um banco de dados.

2.1.3 Formas de mobilidade

Com base no modelo de referência apresentado na seção anterior, em [6, 109, 81], a mobilidade é classificada como fraca ou forte dependendo de quais componentes da unidade de execução são transferidos para o ambiente computacional destino.

Na **mobilidade forte**, toda a UE (código, dados e estado de execução) é transportada para o AC destino. Assim, a execução da UE continua no novo AC a partir da instrução seguinte àquela que causou a migração da UE. A mobilidade forte pode ser provida por meio de **migração** ou **clonagem remota**. Na migração, a UE tem sua execução suspendida e é transferida para o AC destino, onde continua a ser executada. Se a migração ocorrer por determinação da UE, ela é chamada de **pró-ativa**. Se ocorrer como resposta a algum evento gerado por uma outra UE, a migração é denominada **reativa**. No mecanismo de clonagem remota, uma cópia da UE é gerada no AC remoto e a UE original é mantida no mesmo AC. Igualmente à migração, pode ser classificada como pró-ativa ou reativa.

Na **mobilidade fraca**, somente o código é transferido entre ACs e, eventualmente, dados de inicialização. Como o estado de execução não é transportado, o AC destino utiliza o código recebido para iniciar uma nova UE ou então para ligá-lo dinamicamente a uma UE já existente. Os mecanismos de mobilidade fraca podem ser classificados de acordo com a direção da transferência de código em **busca de código** e **envio de código**. Uma classificação mais detalhada de mobilidade fraca pode ser encontrada em [109, 81].

2.1.4 Paradigmas de projeto para códigos móveis

Esta seção apresenta os paradigmas de projeto para códigos móveis com base nos elementos do modelo discutido na Seção 2.1.2. Os paradigmas são descritos em termos das interações entre os componentes do modelo para a realização de um dado serviço. Como exemplos de interações podemos citar o envio de uma mensagem e a relocação de código para um novo ambiente computacional.

Os paradigmas **avaliação remota**, **código sob demanda** e **agente móvel** descritos a seguir estendem o tradicional paradigma **cliente-servidor** para explorar a mobilidade de código.

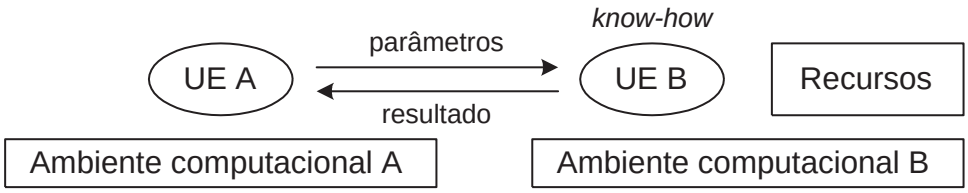

Figura 2.2: Paradigma cliente-servidor.

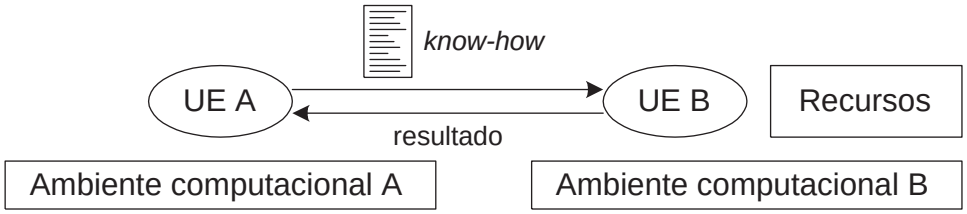

Figura 2.3: Paradigma de avaliação remota.

Cliente-servidor

O tradicional paradigma cliente-servidor é muito utilizado e está ilustrado na Figura 2.2. Neste paradigma, uma unidade de execução B localizada em um ambiente computacional B (o servidor) oferece uma série de serviços para prover acesso a recursos localizados no AC B. Neste caso, dizemos que UE B possui o *know-how* para a realização do serviço. Quando uma UE A remota (o cliente) necessita de algum recurso disponível no AC B, ela utiliza os serviços providos por UE B que realiza a operação e devolve o resultado para UE A. Vale a pena observar que o servidor detém o *know-how*, os recursos e que é responsável pelo processamento do serviço. Exemplos da aplicação deste paradigma são o RPC e RMI Java.

Avaliação remota

No paradigma de avaliação remota, ilustrado na Figura 2.3, uma unidade de execução A possui o *know-how* para a realização de uma tarefa, mas os recursos necessários estão localizados em um ambiente computacional remoto B. Assim, para ter a tarefa realizada, UE A envia o código que implementa o serviço para UE B. Esta, então, executa o código recebido com os recursos contidos em AC B e devolve o resultado do processamento para UE A. Um exemplo deste paradigma é a impressão de um arquivo PostScript por uma impressora que suporte a linguagem.

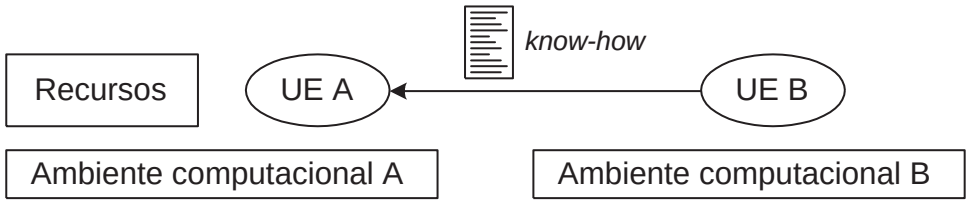

Figura 2.4: Paradigma de código sob demanda.

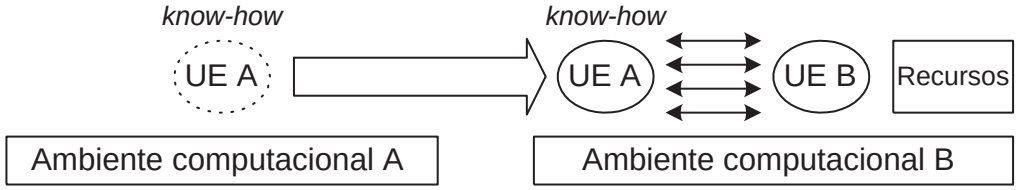

Figura 2.5: Paradigma de agente móvel.

Código sob demanda

A Figura 2.4 ilustra o paradigma de código sob demanda. Neste paradigma, a unidade de execução A possui acesso aos recursos necessários à realização de uma tarefa presentes no mesmo ambiente computacional que UE A. O elemento ausente é o código (*know-how*) para processar os recursos em AC A. Este código é obtido em um ambiente computacional remoto, AC B, e executado localmente por UE A. *Applets* Java são um excelente exemplo da aplicação deste paradigma.

Agente móvel

O paradigma de agentes móveis, ilustrado na Figura 2.5, envolve a migração de toda uma unidade de execução que encapsula o *know-how* para a realização de uma determinada tarefa. Este fato o diferencia dos demais paradigmas que simplesmente transferem código entre unidades de execução estáticas com relação a cada ambiente computacional. Para completar a tarefa programada, uma unidade de execução migra para os ambientes de execução que forneçam os recursos necessários. No ambiente remoto, a UE pode acessar um recurso desejado ou interagir com uma UE local, num estilo cliente-servidor.

2.2 Sistemas de agentes móveis

Um sistema de agentes móveis é uma infra-estrutura que suporta o paradigma de agentes móveis [45, 116]. Esta infra-estrutura é composta por diversos elementos como servidores

de agentes, agentes móveis e estacionários e infra-estrutura de comunicação. Veremos a seguir os componentes principais que são os agentes e os servidores.

2.2.1 Agentes

Um agente de *software* é qualquer programa (código, dados e estado de execução) que realize uma tarefa em nome de uma pessoa, da mesma forma que um agente, na vida real, representa seu cliente (agente de viagens, despachante, etc.) na execução de um serviço. Um agente pode apresentar diversas propriedades [16, 21, 50], algumas das quais estão abaixo listadas:

- **reatividade** – um agente percebe o ambiente em que se encontra e responde a mudanças que ocorram nele.
- **autonomia** – um agente opera sem intervenção humana ou de outro agente, tendo controle sobre suas ações.
- **orientação a objetivo** – um agente pode tomar a iniciativa para alcançar seus objetivos, sem depender apenas de estímulos do ambiente.
- **continuidade temporal** – um agente é um processo que é executado continuamente, ao invés de uma tarefa que aceita uma entrada, gera uma saída e termina.
- **adaptabilidade** – o comportamento do agente muda com base em experiências anteriores.
- **mobilidade** – um agente pode migrar de um ambiente para outro.
- **comunicabilidade** – um agente se comunica com outros agentes ou com usuários.

Dessas propriedades, a única que é comumente citada na literatura como necessária para a caracterização de um agente é a autonomia. Alguns autores definem um agente como um programa que apresenta no mínimo as quatro primeiras propriedades [21, 50].

Considerando a propriedade de mobilidade, podemos classificar os agentes nos sistemas de agentes móveis em estacionários e móveis. Um **agente estacionário** é executado somente no servidor onde foi iniciado. Se necessitar acessar um recurso remoto (uma base de dados ou outro agente), um mecanismo de comunicação como, por exemplo, RPC é utilizado. Por outro lado, **um agente móvel** tem a habilidade de mover-se autonomamente para os servidores que desejar e interagir localmente com os agentes e demais recursos ali presentes.

2.2.2 Servidores de agentes

Um servidor de agentes é um ambiente de execução para agentes móveis e estacionários que oferece recursos necessários à realização das tarefas delegadas aos agentes. Como recursos podemos citar bancos de dados, recursos de aplicação ou simplesmente poder de processamento. Servidores devem prover uma série de serviços aos agentes para possibilitar migração, comunicação, acesso aos recursos, tolerância a falhas e segurança. Para permitir que agentes móveis possam migrar para um dado servidor, este deve ser unicamente identificado e possuir um endereço de rede válido.

O termo servidor de agentes [32, 113, 45] é o que usamos ao longo desta dissertação. Na literatura, muitos outros termos são utilizados para designar este mesmo componente como: plataforma de agentes [116], *place* [80, 2], agência [34] e sistema de agentes [74], entre outros.

2.2.3 Modelo

A Figura 2.6 ilustra um sistema de agentes móveis com seus componentes e algumas interações envolvidas. O sistema é composto de vários *hosts*, cada um deles executando um ou mais servidores de agentes. Agentes podem migrar de forma autônoma entre os servidores em busca de recursos para completar suas tarefas. Além dos servidores e agentes, a infra-estrutura envolve elementos auxiliares como um diretório de nomes e uma terceira parte confiável. O diretório de nomes pode ser usado para a localização de recursos e servidores no sistema. Já a terceira parte confiável, nem sempre presente, pode ser requerida por um protocolo criptográfico. Geralmente, uma infra-estrutura de chaves públicas (ICP) também é necessária devido aos mecanismos de segurança empregados no sistema. Pode-se utilizar uma ICP pré-existente ou fornecer uma como parte do sistema.

2.3 Modelo conceitual comum da OMG

O modelo conceitual comum da OMG descrito no documento "Mobile Agent Facility Specification" (MAF) [74] está ilustrado na Figura 2.7. A especificação MAF trata da interoperabilidade de sistemas de agentes móveis e é uma atualização do documento "Mobile Agent System Interoperability Facility" (MASIF). Como muitos dos conceitos desse modelo são os mesmos que os do modelo da seção anterior, vamos apenas descrever os conceitos adicionais e diferenças entre os modelos.

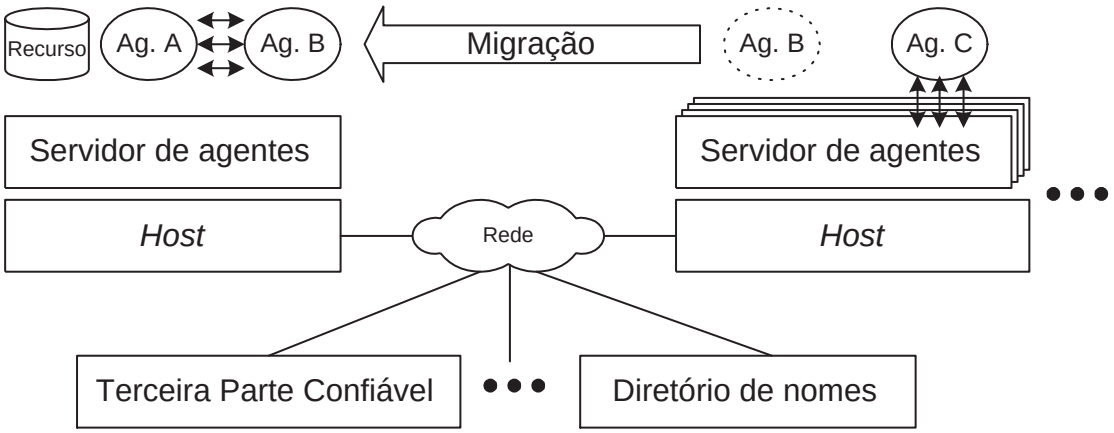

Figura 2.6: Um sistema de agentes móveis.

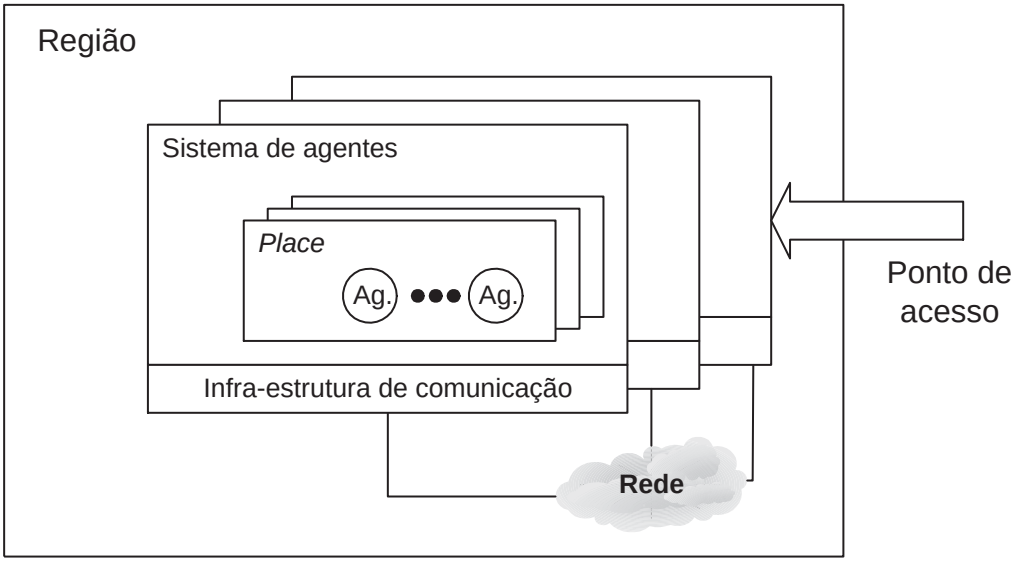

Figura 2.7: Modelo conceitual comum da OMG.

2.3.1 Agente

Cada agente está associado a uma **autoridade de agente** que identifica a pessoa ou organização que o agente representa. Um agente requer um **nome** que seja globalmente único e que é composto pela autoridade, identidade e tipo de sistema de agentes. Uma identidade de agente é um identificador único dentro do escopo de uma autoridade que identifica uma instância de agente em particular. Como o nome de um agente é um valor globalmente único, ele pode ser usado em operações de gerenciamento e serviços de nome para referir-se a um agente específico. A **localização de um agente** é dada pelo nome do sistema de agentes onde o agente reside e por um nome de *place* (descrito na Seção 2.3.3) contido neste sistema.

2.3.2 Sistema de agentes

Um sistema de agentes, na especificação MAF, é semelhante ao servidor de agentes apresentado na Seção 2.2.2. Portanto, este componente é uma plataforma que serve de ambiente de execução para agentes e que pode criá-los, transferi-los e terminá-los. Vale a pena mencionar que usamos o termo sistema de agentes móveis, nesta dissertação, para nos referirmos ao conceito apresentado na Seção 2.2, em detrimento deste aqui.

Cada *host* que queira suportar agentes móveis deve executar pelo menos um sistema de agentes. Da mesma forma que um agente, um sistema de agentes está associado a uma autoridade que é a entidade responsável pelo sistema. A política de segurança implementada por um sistema de agentes é aquela definida pela autoridade responsável. Um sistema de agentes é unicamente identificado por seu nome e endereço do *host* em que é executado. O **tipo de sistema de agentes** determina o perfil de um agente móvel. Assim, por exemplo, se o tipo for Aglet, o sistema de agentes suporta agentes escritos em Java e conseqüentemente utiliza o mecanismo de serialização de objetos daquela linguagem para a migração de agentes. Sistemas de agentes podem se comunicar com outros sistemas de agentes utilizando os serviços da infra-estrutura de comunicação.

2.3.3 *Place*

O ambiente de execução provido pelo sistema de agentes pode ser dividido em diversos contextos denominados *places*. Um *place* pode ser usado para fornecer um agrupamento lógico de funcionalidades dentro de um sistema de agentes como no sistema Grasshopper [34]. Por exemplo, pode haver um *place* de negócios, onde agentes anunciem ou comprem produtos, e outro que forneça informações sobre os serviços dos demais *places* existentes no mesmo sistema. Se o sistema não implementar *places*, o *place* padrão é definido como o próprio sistema de agentes.

2.3.4 Região

Uma região é um agrupamento de sistemas de agentes sob mesma autoridade com tipos potencialmente diferentes. Com isso, uma mesma autoridade pode ser representada por diversos sistemas de agentes. Uma região possui um ou mais sistemas de agentes que são designados como pontos de acesso para a região. Os endereços destes sistemas específicos representam os possíveis endereços que podem ser usados para que clientes de outras regiões comuniquem-se com a região em questão. Portanto, é por meio desses pontos de acesso que é realizada a interconexão entre regiões.

2.3.5 *Codebase*

O *codebase* especifica as localizações em que as classes utilizadas por um agente podem ser encontradas. As classes são disponibilizadas por uma entidade chamada de **provedor de classes** (*class provider*) que pode ser um sistema de agentes ou um servidor Web, entre outros.

2.3.6 Infra-estrutura de comunicação

A infra-estrutura de comunicação provê serviços de comunicação como RPC, serviços de nomes e mecanismos de segurança para comunicação.

2.4 Vantagens de agentes móveis

Nesta seção apresentamos algumas vantagens proporcionadas por sistemas de agentes móveis que são identificadas em diversos trabalhos na literatura [9, 38, 50, 54, 28, 31]. Estas vantagens não são exclusivas de tais sistemas e também estão presentes individualmente em soluções alternativas [9, 38, 28]. O que então torna o uso de agentes móveis atraente é a combinação de todas estas vantagens em um único *framework*, que pode ser usado para desenvolver diversos tipos de aplicações distribuídas de forma fácil, eficiente e uniforme. Com os paradigmas tradicionais, diferentes aplicações requerem uma combinação diferente de técnicas para se alcançar os mesmos objetivos.

A título de exemplo, citamos um experimento realizado na Dartmouth College com alunos de graduação de primeiro e segundo anos que consistia no desenvolvimento de aplicações distribuídas utilizando-se agentes móveis [31]. Mesmo sem conhecimentos de sistemas distribuídos, os alunos conseguiram desenvolver muitas das aplicações propostas, o mesmo não ocorrendo com tecnologia cliente-servidor. Esse resultado sugere que agentes móveis são mais fáceis de se entender do que outros paradigmas como RPC, por exemplo.

Isso se deve, ao tratamento uniformizado que o paradigma de agentes móveis proporciona ao desenvolvimento de aplicações distribuídas.

2.4.1 Redução do tráfego de rede

Em sistemas baseados no modelo cliente/servidor, como visto na Seção 2.1.4, o servidor fornece um conjunto fixo de serviços que podem ser acessados pelos clientes. Para se realizar uma tarefa, normalmente é necessário efetuar várias chamadas aos serviços disponibilizados pelo servidor. Com a adição de mecanismos de segurança, o fluxo de dados entre as máquinas aumenta ainda mais. Isso acontece porque geralmente o conjunto de serviços fornecidos não atende exatamente a cada necessidade específica dos clientes e, assim, dados intermediários são enviados para se obter o resultado desejado. Para se ter uma idéia da quantidade de tráfego de rede que pode ser gerada, basta imaginarmos que o cliente acessa um conjunto muito grande de dados não estruturados presente em diversos servidores e que todos esses dados são transferidos pela rede ao cliente.

Um agente móvel, por sua vez, reduz o tráfego de rede pois é transferido e executado no mesmo local em que estão os recursos necessários à realização de sua tarefa. A filtragem e o processamento dos dados são feitos nos próprios servidores e, dessa forma, o agente retorna apenas o resultado final para o usuário. Assim, aplicações que requeiram o processamento de um conjunto muito grande de dados localizados remotamente e cujos resultados não sejam fornecidos por serviços especializados nos servidores podem se beneficiar do uso de agentes móveis. Estudos comparativos do tráfego de rede gerado por soluções baseadas em agentes móveis e tecnologia cliente-servidor podem ser encontrados em [38, 28, 31].

2.4.2 Redução na latência

Em sistemas de tempo-real, que devem reagir instantaneamente aos eventos ocorridos, muitas vezes a latência da rede é alta se comparada às restrições de tempo-real impostas pelos sistemas. Dessa forma, pode ser problemático que o programa de controle se situe remotamente aos sensores e atuadores. Implementar a lógica de controle como um agente móvel diminui a latência, pois o agente pode se posicionar proximamente aos equipamentos de monitorização e controle e responder aos eventos localmente. Além disso, o agente pode se adaptar dinamicamente conforme mudem as condições da rede ou o conjunto de sensores/atuadores, de forma a posicionar-se sempre em uma melhor localização. Um exemplo extremo dado em [9] é o controle de sondas espaciais para exploração do sistema solar. Neste tipo de ambiente, a latência se torna evidente bem como a necessidade de respostas rápidas aos eventos ocorridos. Por exemplo, a sonda pode ter se desviado de sua rota e precisar de um ajuste imediato.

2.4.3 Execução assíncrona e autônoma

Um agente móvel, após despachado do servidor de origem, trabalha de forma assíncrona e autônoma em relação à aplicação que o criou. Isso significa que durante a realização da tarefa delegada, o agente móvel não interage com a aplicação-mãe, até o seu retorno ao servidor inicial. Essa característica torna o uso de agentes móveis atraente em ambientes de computação móvel que acessam a rede por meio de conexões muitas vezes não confiáveis ou caras. Pode-se encapsular a tarefa desejada em um agente móvel e liberá-lo na rede enquanto houver conexão disponível. Enquanto o agente móvel completa a tarefa confiada, o dispositivo móvel pode ser desligado ou ser desconectado da rede, sem prejudicar o funcionamento do agente. Com uma solução tradicional, baseada no modelo cliente-servidor, seria necessário bloquear a aplicação toda vez que a conexão se tornasse indisponível.

2.4.4 Instalação dinâmica de *software*

Conforme visto anteriormente, em um ambiente cliente/servidor, cada servidor provê um conjunto fixo de operações. Quando um cliente deseja realizar uma tarefa que não possui uma operação correspondente no conjunto de operações do servidor, diversas chamadas devem ser efetuadas, aumentando-se o tráfego da rede. Se um servidor for adicionar uma nova rotina para atender as necessidades específicas de cada cliente, ele se tornará extremamente complexo, contrariando as boas práticas de engenharia de *software* [28].

Agentes móveis fornecem uma forma simples de resolver esse problema: uma nova operação do servidor é encapsulada por um agente e executada localmente no servidor, com base nas operações básicas disponibilizadas. Se desejável, o agente pode ser executado como um serviço e atender requisições de outros clientes que possuam as mesmas necessidades específicas.

Outro exemplo de instalação dinâmica de *software*, dado por Kotz [67], é o de uma grande companhia que deseja atualizar e reconfigurar os *softwares* de diversos PDAs utilizados por seus funcionários. O agente móvel pode carregar os novos arquivos, copiá-los para cada PDA e realizar as reconfigurações de forma personalizada.

Johansen conclui em [38] que o melhor argumento para o uso de agentes móveis neste contexto é a facilidade de uso, pois o paradigma oferece uma maneira simples de instalar e relocar código em aplicações distribuídas.

2.4.5 Adaptação dinâmica

Agentes móveis podem monitorar o ambiente e se adaptar autonomamente a mudanças ali ocorridas. Dessa forma, um agente móvel pode, se necessário, migrar para outro servidor,

seja para reduzir a latência, seja para balancear a carga entre os servidores. Igualmente, um conjunto de agentes pode se reposicionar, de tempos em tempos, pelos servidores da rede de forma a melhorar suas localizações para solucionar um dado problema.

2.4.6 Robustez e tolerância a falhas

A capacidade de agentes móveis de reagir dinamicamente a mudanças ocorridas no ambiente, muitas vezes desfavoráveis, facilita a criação de sistemas distribuídos robustos e tolerantes a falhas. Por exemplo, se for necessário desligar um servidor para realizar sua manutenção, os agentes que estão sendo executados por ele podem ser avisados do evento e ter a chance de migrar para um outro servidor, antes que aquele se torne indisponível.

2.5 Exemplos de aplicações

Atualmente há um consenso entre os pesquisadores de que toda aplicação que pode ser desenvolvida utilizando-se agentes móveis também possui uma solução igualmente eficiente baseada em métodos tradicionais [38, 67, 28]. Dessa forma, as opiniões convergem para a inexistência de uma *killer application* [67] para o paradigma. Ainda assim, o uso de agentes móveis para o desenvolvimento de sistemas distribuídos é atraente, pois provê um *framework* geral e uniforme para a implementação de uma diversidade de sistemas. A seguir, estão alguns exemplos de aplicações desenvolvidas para sistemas de agentes móveis.

2.5.1 Busca e filtragem de informação

A quantidade de informação disponível na Internet e em outras redes é imensa e normalmente precisamos acessar uma grande quantidade de dados até encontrarmos o que estamos procurando. Esta tarefa de busca e filtragem pode consumir bastante tempo, além de desperdiçar a banda-passante da rede com dados irrelevantes, principalmente quando o conjunto de dados a ser pesquisado for extenso. Um agente móvel pode realizar esta tarefa de forma autônoma visitando os servidores de dados e efetuando a filtragem localmente, nos próprios servidores. Dessa forma, o agente retorna ao servidor de origem apenas com as informações de interesse.

Nesta linha de aplicação, Johansen descreve um sistema que recebe e processa dados gerados por satélite [38]. O conjunto de dados é da ordem de Terabytes e as imagens são disponibilizadas a clientes espalhados pelo mundo inteiro. A solução é construída utilizando-se o sistema de agentes móveis TACOMA (Seção 5.9.1). Um servidor TACOMA recebe novas imagens de satélite diariamente, as quais são disponibilizadas aos agentes

móveis visitantes. Um agente é personalizado de acordo com as necessidades do usuário a quem representa. Assim, por exemplo, um agente pode percorrer apenas as novas imagens de uma determinada área geográfica, analisá-las e selecionar apenas uma delas conforme um critério definido pelo usuário. Em termos de tempo de resposta e tráfego de rede, o experimento mostrou que a solução baseada em agentes móveis superava uma solução tradicional baseada em tecnologia cliente/servidor.

Outro exemplo é o de uma aplicação, baseada no sistema D'Agents (Seção 5.5), que realiza buscas de relatórios técnicos disponíveis no Departamento de Ciência da Computação da Dartmouth College [28]. Os relatórios técnicos estão distribuídos em diversas máquinas, cada uma executando um servidor D'Agents. Um agente estacionário, presente em cada servidor, fornece serviços para a obtenção de relatórios e para a pesquisa de documentos a partir de palavras-chaves. Quando um usuário deseja procurar um relatório, ele utiliza a aplicação cliente para especificar algumas palavras que devem estar presentes nos documentos. Um agente móvel é criado para realizar a pesquisa e como primeiro passo, ele verifica a qualidade da conexão de rede para decidir se continua na máquina cliente ou se migra para um servidor *proxy*. Após esse primeiro passo, o agente móvel consulta um diretório de serviços para localizar as coleções de documentos. Dependendo do número de operações requeridas por relatório, o agente pode decidir entre efetuar chamadas RPC aos servidores ou gerar agentes filhos para pesquisar as coleções. Ao final, uma lista de documentos é gerada e exibida ao usuário. Comparações de desempenho entre esta solução e outra baseada no modelo cliente/servidor são encontradas em [28].

2.5.2 Monitorização

Este tipo de aplicação aproveita-se da natureza assíncrona e autônoma dos agentes móveis, que podem ser criados para monitorar a ocorrência de certos eventos. Por exemplo, um agente móvel pode ser despachado para um servidor de venda e compra de ações, esperar que uma ação alcance um determinado valor e então comprar uma certa quantidade em nome do usuário [107]. Outro exemplo interessante é o emprego de agentes móveis para monitorar redes e sistemas distribuídos.

O uso potencial de agentes móveis para o gerenciamento de redes e as vantagens e desvantagens da aplicação do paradigma neste tipo de aplicação são discutidos em [5]. O modelo de gerenciamento OSI define cinco áreas funcionais que são: gerenciamento de falhas, gerenciamento de configuração, gerenciamento de desempenho, gerenciamento de segurança e gerenciamento contábil (*accounting*). Os autores analisam várias aplicações em cada uma dessas áreas.

Em [57, 58] é descrito um sistema que realiza monitorização ativa e distribuída de redes, utilizando o paradigma de agentes móveis. A idéia é utilizar agentes móveis como

monitores de áreas e explorar sua mobilidade para construir um sistema que se reconfigure à medida que o estado dos objetos monitorados mude. O sistema é descentralizado porque a rede monitorada é dividida em partições disjuntas, tal que cada partição é associada a um agente dinamicamente. A localização inicial de cada agente e o particionamento da rede é realizado por meio de um algoritmo distribuído que utiliza as informações geradas pelos protocolos de roteamento. Após o posicionamento inicial, cada agente móvel verifica periodicamente as condições da rede por meio das tabelas de rotas e avalia sua localização atual. No caso de falha em algum nó da rede ou queda de um *link*, o agente migra para uma nova máquina, de onde continua sua tarefa. Por meio de simulação, os autores demonstraram a viabilidade da abordagem e os benefícios obtidos [57].

O trabalho apresentado em [101] descreve um sistema de detecção de intrusão baseado no sistema de agentes móveis Ajanta (Seção 5.7). Cada nó do ambiente monitorado deve executar um servidor Ajanta para receber e executar agentes móveis. Estes são encarregados de monitorar alguns eventos e realizar a correlação dos dados coletados. Um agente carrega diversos **detectores** que encapsulam a lógica de detecção e que podem ser atualizados dinamicamente. A tarefa de detecção envolve a verificação de arquivos de *log*, processos e sistema de arquivos, entre outros recursos. Os experimentos realizados com o sistema utilizaram diversos cenários como: monitorização de atividade do usuário `root` e monitorização de requisições não autorizadas a serviços.

Capítulo 3

Segurança de Sistemas de Agentes Móveis

As seções deste capítulo estão assim organizadas: a Seção 3.1 apresenta um modelo simples de sistema de agentes móveis com base no qual são listadas as classes de ataques possíveis. A Seção 3.2 introduz brevemente os requisitos de segurança desejáveis nestes sistemas. Os mecanismos de proteção do servidor são apresentados na Seção 3.3 e as técnicas de proteção de agentes móveis, na Seção 3.4. Por fim, a Seção 3.5 resume e classifica os mecanismos de proteção descritos nas seções anteriores.

3.1 Tipos de ameaças e ataques

Para apresentar os principais tipos de ataques e ameaças aos sistemas de agentes móveis vamos utilizar o modelo ilustrado na Figura 3.1. Este modelo é bastante simples e nele o sistema de agentes é composto apenas por dois elementos: agentes móveis e servidores de agentes. Os agentes possuem código e dados e realizam alguma tarefa em nome de um usuário. Para isso, eles podem migrar de forma autônoma para os servidores que ofereçam os recursos necessários à sua execução. O servidor onde o agente é criado é chamado de servidor de origem e corresponde à entidade na qual o agente deposita maior confiança. A Figura 3.1 ilustra também as terceiras partes, que são todas as entidades externas ao sistema de agentes que efetuem algum ataque. Os ataques são representados por meio de setas como a seguir:

$$Atacante \longrightarrow Atacado$$

Cada seta foi rotulada com um número que representa uma das classes de ataques e ameaças possíveis, de acordo com classificação feita pelo NIST [36, 37]:

1. ameaças do servidor contra o agente;

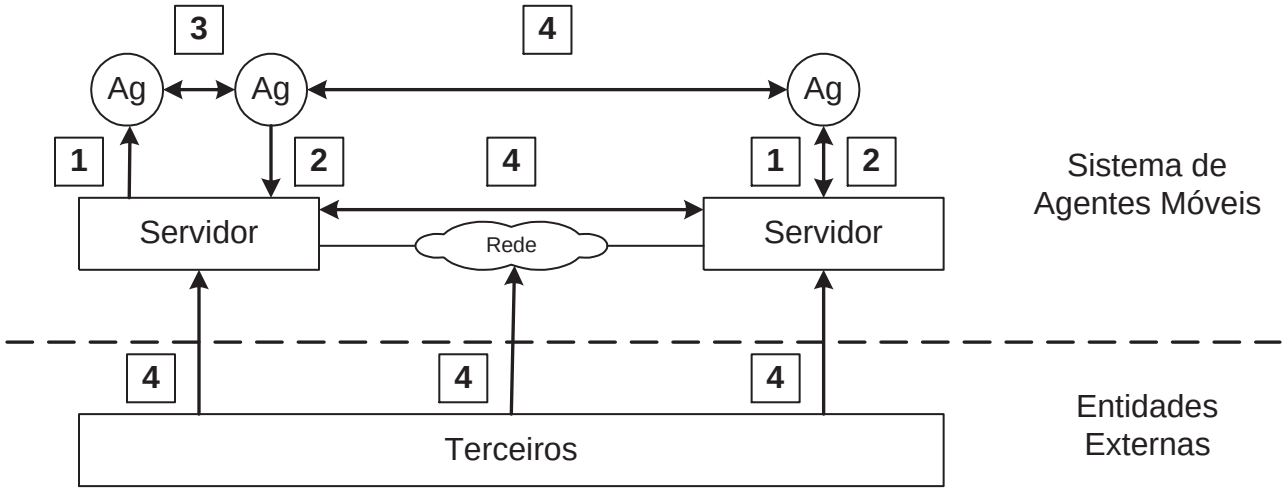

Figura 3.1: Ataques a um sistema de agentes móveis.

2. ameaças do agente contra o servidor;

3. ameaças do agente contra outro agente sendo executado no mesmo servidor; e

4. outras ameaças contra o sistema de agentes.

Grande parte dos sistemas analisados no Capítulo 5 tem boas respostas apenas para as três últimas classes de ameaças. Já o problema de proteção do agente contra servidores maliciosos ainda possui muitos aspectos não resolvidos e é considerado por apenas alguns dos sistemas existentes e de forma parcial. Nas próximas seções detalhamos e exemplificamos cada uma das classes de ameaças.

3.1.1 Ameaças do servidor contra o agente

Um agente que é executado por um servidor está completamente exposto ao mesmo. Se nenhum mecanismo de proteção for utilizado pelo agente, tanto o seu código como os seus dados serão completamente acessíveis ao servidor, que pode, então, afetar o agente de diversas maneiras. Como, geralmente, parte do estado de um agente precisa ser alterado para armazenar resultados de cálculos ou dados coletados, acredita-se não ser possível garantir, de uma forma geral, que um agente não seja alterado de forma maliciosa [18]. Essa classe de ameaças compreende os problemas mais difíceis de serem tratados.

Personificação

Um servidor malicioso pode personificar um outro servidor confiável, para extrair informações de agentes móveis que tenham como destino o servidor original. Como o ser-

vidor malicioso, em caso de um ataque bem sucedido, terá a posse do agente enganado, outros ataques, como os descritos a seguir, poderão suceder a personificação.

Negação de serviço

Quando um agente migra para um servidor, ele espera que o serviço seja executado corretamente e no tempo esperado. Um servidor malicioso pode não atender à solicitação feita, realizar a tarefa delegada em um tempo inaceitável ou finalizar o agente abruptamente. Este último caso pode fazer a aplicação que enviou os agentes entrar em um estado de *deadlock*, caso ela fique esperando indefinidamente pelo retorno do agente. Uma outra situação possível é o *livelock* na qual o agente nunca consegue terminar uma tarefa porque o servidor não pára de lhe fornecer serviço.

Eavesdropping

No cenário clássico de "escuta", o atacante intercepta o canal de comunicação estabelecido entre duas partes, A e B, normalmente chamadas pela literatura de Alice e Beto, respectivamente. No caso de agentes móveis, além desta possibilidade, há um problema maior: os servidores tem acesso a toda informação não cifrada contida nos agentes visitantes como instruções, dados e resultados. Dessa forma, um servidor malicioso poderia obter algoritmos proprietários, números de cartão de crédito ou qualquer outra informação contida no agente. Além disso, é possível que um servidor faça inferências pela observação dos tipos de serviços utilizados por um agente e das entidades com as quais ele se comunica. Por exemplo, um agente P que só troque mensagens com agentes de livrarias virtuais é um bom indicativo de que o usuário proprietário de P deseja comprar um livro.

Alteração de dados ou código

Um agente pode passar por vários servidores, dentre os quais alguns maliciosos; assim, mecanismos que permitam a verificação da integridade do código e dos dados carregados pelo agente devem ser providos aos servidores honestos. Segundo [36], não existe ainda uma solução geral para se detectar a alteração maliciosa do estado de um agente durante sua execução. Soluções calcadas em linguagens de programação não são suficientes, pois o servidor malicioso pode estar executando uma máquina virtual modificada, por exemplo.

É importante verificar a integridade do agente quando ele retorna ao servidor de origem. Embora um servidor tenha confiança em seus próprios agentes, estes podem ter sido alterados por algum servidor malicioso, ao longo do itinerário percorrido, com o intuito de comprometer a segurança do servidor de origem.

3.1.2 Ameaças do agente contra o servidor

Nesta categoria de ataques, os agentes procuram explorar fraquezas existentes nos servidores.

Personificação

É o ataque no qual um agente tenta se passar por um outro para utilizar recursos aos quais não tenha acesso ou para não ser responsabilizado por ações que venha a realizar.

Negação de serviço

Este cenário pode ocorrer quando os agentes consomem excessivamente os recursos do servidor. Isto pode acontecer por *bugs* nos agentes ou propositadamente. Enquanto a primeira causa pode ser atacada por técnicas de engenharia de software, a segunda requer o uso de mecanismos de segurança. Dependendo dos tipos de recursos que são oferecidos aos agentes, é possível, em alguns casos, tornar o sistema completamente indisponível.

Acesso não autorizado

O acesso não autorizado a recursos pode comprometer o funcionamento e a segurança de servidores e dos agentes sendo executados nos mesmos. Por exemplo, em uma aplicação bancária sem o devido controle de acesso, seria possível que um agente malicioso movimentasse o dinheiro de contas de terceiros. É necessário, então, estabelecer uma política de segurança e aplicá-la, por meio de mecanismos de controle de acesso, a todo agente que queira ser executado no servidor em questão.

3.1.3 Ameaças do agente contra outro agente

Esta categoria compreende os ataques em que um agente tenta explorar fraquezas na segurança de outros agentes presentes no mesmo servidor. Os casos de agentes atacando agentes em outras plataformas estão descritos na Seção 3.1.4 e estão baseados nos serviços de comunicação dos servidores.

Personificação

Um agente pode tentar se passar por outro agente para obter informações confidenciais de um terceiro ou para prejudicar o funcionamento deste último. Pode-se realizar este ataque também com o intuito de manchar a reputação do agente personificado. Um cenário típico deste ataque seria o de um agente personificando um agente de alguma aplicação de comércio eletrônico para obter números de cartão de crédito.

Negação de serviço

Agentes maliciosos podem realizar um ataque de negação de serviço contra outros agentes, enviando-lhes uma grande quantidade de mensagens desnecessárias. Independentemente de se processar ou bloquear as mensagens, tempo de processamento será desperdiçado pela entidade atacada. Com isso, uma tarefa sendo realizada por esta entidade pode não ser finalizada no prazo estabelecido. Outra forma deste ataque consiste em fornecer informações incorretas para que um agente não funcione apropriadamente.

Repúdio

Quando um agente participante de uma transação ou comunicação nega ter realizado as ações que foram efetuadas tem-se um repúdio. Este pode ocorrer devido à ação de um agente malicioso ou acidentalmente, por perda de uma mensagem. De qualquer forma, grandes disputas podem se originar e, por isso, é importante que a infra-estrutura do sistema de agentes móveis tenha mecanismos para poder auxiliar na resolução de tais situações.

Acesso não autorizado

Se não houver controle de acesso a recursos, um agente malicioso pode alterar dados ou mesmo o código de um outro agente residente no mesmo servidor. O último caso é mais grave ainda pois uma alteração deste tipo provoca uma mudança de comportamento do agente afetado.

3.1.4 Outras ameaças contra o sistema de agentes

Nesta categoria estão os ataques perpetrados por agentes contra agentes situados em outros servidores, ataques de servidores contra servidores e ataques tradicionais em ambiente de rede explorando falhas de segurança em sistemas operacionais. Um ponto comum nestes ataques é que eles estão baseados nos serviços de comunicação dos servidores e serviços de rede.

Personificação

Um agente pode personificar outro agente para conseguir acesso a um recurso ou a um serviço oferecido por um servidor remoto. Um servidor pode também assumir uma outra identidade para enganar agentes e outros servidores.

Acesso não autorizado

Agentes remotos ou outras entidades externas podem tentar acessar recursos para os quais eles não possuam os devidos privilégios. Com o acesso indevido, um atacante pode destruir dados e comprometer o funcionamento de um servidor por exemplo.

Negação de serviço

Como os serviços dos servidores podem ser acessados remotamente, é possível realizar ataques de negação de serviço comuns contra eles. Pode-se direcionar estes ataques também aos sistemas operacionais e aos protocolos de comunicação.

Eavesdropping

O atacante pode "escutar"o canal de comunicação entre servidores para obter informações de agentes migrantes ou dados por eles carregados. Outra possibilidade seria interceptar um agente ou alguma mensagem para efetuar um ataque por repetição.

Ataques por repetição

Neste ataque, uma terceira parte maliciosa intercepta um agente ou uma mensagem, realiza uma cópia e depois, clona o agente ou retransmite a mensagem, respectivamente. A interceptação acontece "escutando-se"o canal de comunicação ou por meio de um servidor malicioso, quando este recebe um agente migrante ou uma mensagem.

3.2 Requisitos de segurança

Sistemas de agentes móveis devem atender a alguns requisitos de segurança como confidencialidade e integridade. Nesta seção, damos uma breve introdução a esses requisitos e como eles se aplicam aos sistemas de agentes móveis.

3.2.1 Confidencialidade

Muitas vezes os agentes possuem código e/ou dados sigilosos e, assim, devem ser providos meios para evitar que servidores e outros agentes não autorizados tenham acesso a essas informações. A mesma consideração é válida para o caso dos servidores. A proteção de dados de servidores e partes do estado do agente que não se destinam a seu próprio uso (por exemplo, dados coletados para uso no servidor origem ou destinados a servidores específicos) é facilmente obtida por métodos criptográficos. A situação torna-se mais difícil quando se deseja proteger um dado carregado pelo agente que será usado pelo

mesmo antes de seu retorno ao servidor de origem. Igualmente difícil é a proteção de código, mas que pode ser obtida para algumas classes de funções (veja Seção 3.4.6).

É desejável que a confidencialidade seja mantida por toda a infra-estrutura de agentes móveis, não se limitando apenas à relação agente-servidor. Assim, os canais de comunicação pelos quais trafegam agentes e suas mensagens também devem atender a este requisito. É importante notar que informações sobre a atividade de um agente podem ser obtidas por outros meios que a simples inspeção do conteúdo de variáveis e mensagens trocadas entre as partes do sistema. Um exemplo disso é a análise de tráfego: o reconhecimento de padrões na troca de mensagens entre um agente, uma loja virtual e algumas entidades financeiras pode bastar para inferir que uma compra ou outra transação foi realizada.

Finalmente, agentes móveis podem desejar manter sua localização confidencial [36]. Isso pode ser obtido utilizando-se um *proxy* para comunicação cuja localização é publicamente conhecida e que esconde a localização real do agente.

3.2.2 Integridade

Um servidor precisa proteger os agentes contra modificações em seu código e estado e garantir que somente entidades autorizadas possam alterar dados compartilhados. Porém, se o servidor for a entidade maliciosa, não há meios de se impedir que o agente seja modificado. Neste caso, ao menos, deve-se prover algum mecanismo que permita servidores e agentes honestos detectar a modificação indesejada.

A integridade dos servidores também é um ponto importante a ser considerado. Servidores podem ser atacados e modificados para que passem a se comportar maliciosamente. Modificações válidas não intencionais também podem, às vezes, fazer o servidor ter um comportamento inadequado. Por exemplo, pode-se remover, acidentalmente, uma permissão de um arquivo de configuração, fazendo com que o servidor negue autorização incorretamente a uma entidade.

Igualmente, qualquer comunicação entre servidores e agentes deve possuir mecanismos para verificação de integridade, para evitar que mensagens sejam modificadas, substituídas ou removidas ou que tenham o remetente ou destinatário trocados, de forma imperceptível.

3.2.3 Autenticação

A autenticação permite identificar unicamente cada entidade ou informação no sistema. Por exemplo, quando se estabelece uma comunicação, as partes envolvidas devem ser identificadas. Pode-se dividir a autenticação em autenticação de entidades (também chamada de identificação) e autenticação da origem dos dados. Diversos protocolos criptográficos são projetados para esse fim [61, 95, 91, 94]. A autenticação é importante em sistemas

de agentes móveis para que seja possível manter um registro das atividades de agentes e servidores, conceder autorizações e efetuar cobrança pelo uso de recursos.

3.2.4 Não-repúdio

Não-repúdio é um requisito de segurança que impede que entidades neguem as ações por elas realizadas em uma transação ou comunicação. Para isso, a infra-estrutura de agentes móveis deve coletar e registrar informações que comprovem a ocorrência de transações e eventos importantes. É fácil observar como isso é importante, por exemplo, no contexto de comércio eletrônico.

3.2.5 Disponibilidade

Os servidores devem garantir disponibilidade de dados e serviços aos agentes locais e remotos, prover acesso concorrente ou exclusivo aos recursos e gerenciar *deadlocks*. Além disso, os servidores devem ser capazes de lidar com a quantidade de requisições realizadas pelos agentes para não causar negações de serviço não intencionais. Caso os servidores não possam atender a toda sua carga, devem degradar de forma graciosa e sinalizar essa situação às partes interessadas.

Os riscos de ataques de negação de serviço deixam clara a necessidade de controle na alocação dos recursos dos servidores. A dificuldade neste controle é diferenciar alocações válidas de maliciosas. E a imposição de limites no uso de recursos, muitas vezes impede agentes bem intencionados de realizarem sua tarefa. Um ataque de negação de serviço bem sucedido afeta todos os agentes e servidores que dependam da máquina atingida.

3.2.6 Anonimato

Em alguns casos é desejável que a identidade da pessoa responsável pelo agente móvel permaneça no anonimato. Por exemplo, uma pessoa pode querer não se identificar ao responder um questionário de avaliação; ou então ao adquirir algum bem ou ao utilizar um serviço. Há muitas situações, porém, em que os participantes podem relutar em negociar com uma parte anônima. Por exemplo, uma entidade financeira necessita da identidade de uma pessoa interessada em um empréstimo para poder verificar seu histórico financeiro e avaliar o montante a ser liberado.

A necessidade de privacidade de um agente móvel deve ser cuidadosamente balanceada com relação à necessidade dos servidores de manterem os agentes responsáveis pelas ações realizadas.

3.2.7 Responsabilização (*Accountability*)

Cada processo, usuário ou agente em um dado servidor devem ser responsabilizados pelas ações que venham a realizar [36]. Como exemplos dessas ações estão acessos a um arquivo ou mudanças administrativas nos mecanismos de segurança do servidor. Para que haja responsabilização, cada entidade no sistema deve ser unicamente identificada e autenticada. Sem responsabilização, mecanismos de proteção baseados em detecção *a posteriori* seriam completamente ineficazes: um servidor poderia, por exemplo, modificar um agente e não ser punido pelos seus atos.

3.3 Mecanismos de proteção do servidor

Como visto na Seção 3.1, os servidores de agentes estão sujeitos a ataques perpetrados por agentes móveis, outros servidores e entidades externas ao sistema. Felizmente, muitas técnicas tradicionais, como o monitor de referências [1], podem ser aplicadas neste paradigma, para proteção dos servidores. Nas seções seguintes, são apresentadas algumas das técnicas propostas na literatura para esse fim.

3.3.1 Monitor de referências

Uma das abordagens para a proteção de servidores é aplicar o conceito de monitor de referências [1], cuja função é autorizar ou proibir o uso de um recurso, toda vez que um acesso é realizado por uma entidade do sistema. Com esse mecanismo, é possível impedir que agentes interfiram na execução de outros agentes e do próprio servidor. Basta para isso isolar os agentes e servidores em domínios separados e controlar todo acesso entre domínios por meio do monitor de referências.

Uma implementação de um monitor de referências deve possuir as seguintes características:

- deve ser inviolável (*tamper-proof*);
- deve ser invocado em todo e qualquer acesso a um recurso; e
- deve ser suficientemente pequeno para que possa ser analisado e provada sua corretude.

3.3.2 Isolamento de falhas baseado em *software*

Muitas vezes, programas utilizam módulos desenvolvidos por terceiros. Uma falha em um desses módulos pode corromper dados da aplicação, tornando-a instável. Uma maneira de

evitar esse problema é prover isolamento de falhas entre módulos de um programa por meio da alocação de um espaço de endereçamento diferente para cada módulo. Módulos podem chamar outros módulos em espaços de endereçamento diferentes por meio de RPC, por exemplo. Implementações por *hardware* dessa abordagem têm um custo elevado quando há muita comunicação entre módulos em espaços diferentes [112].

Wahbe *et al.*, em [112], descrevem uma forma eficiente de implementar isolamento de falhas por *software* dentro de um único espaço de endereçamento. Inicialmente, cada módulo não confiável é isolado em seu próprio domínio de falhas, que contém um segmento de código e outro de dados. Um segmento é definido como uma região contígua do espaço de endereçamento virtual em que todos os endereços virtuais apresentam um mesmo valor para os k bits de mais alta ordem. Por exemplo, a faixa de endereços 0xab000000–0xabffffff é um segmento, com $k = 8$.

Em seguida, as instruções dos módulos são modificadas para prevenir acessos (escrita/desvio) fora de seu domínio de falhas. As modificações visam as instruções inseguras que são aquelas de escrita e desvio cujos endereços utilizados não podem ser determinados estaticamente e, portanto, comparados para determinar se estão dentro dos segmentos corretos. As instruções que apresentam essa característica são as que usam o conteúdo de registradores como endereço destino. Duas técnicas são apresentadas pelos autores para evitar que tais instruções resultem em endereços inválidos.

A primeira técnica, chamada de ***matching* de segmento**, insere código para verificação de endereços em tempo de execução antes de cada instrução insegura. A outra técnica, chamada de ***sandboxing* de endereços**, insere código antes das instruções inseguras para setar os k bits de mais alta ordem do endereço destino para o segmento correto. Considerando nosso exemplo anterior, seria armazenado o valor 0xab. Esta última técnica é mais eficiente que a primeira porque necessita de menos código adicional.

3.3.3 Interpretação segura de código

Sistemas de agentes móveis geralmente são desenvolvidos utilizando-se linguagens de programação ou *scripting* interpretadas. Isso se deve à necessidade de os agentes serem executados em ambientes heterogêneos. Como os comandos de um programa escrito em linguagem interpretada já são mediados, fica fácil adicionar controles de segurança a eles. A interpretação segura de código consiste em impedir a execução de comandos considerados perigosos ou torná-los inofensivos. Como exemplo de comandos perigosos podemos dar aqueles para remoção de arquivos e os que permitem executar comandos e programas externos.

Uma das linguagens de programação interpretadas mais utilizadas hoje em dia é a linguagem Java. Seu modelo de segurança é conhecido como *sandbox* e é analisado em

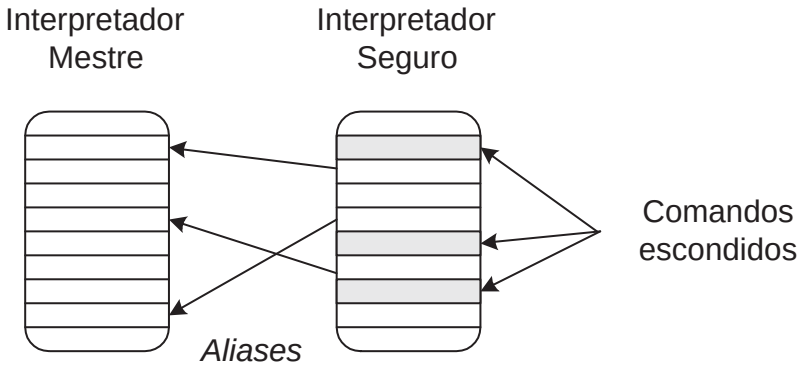

Figura 3.2: Mecanismos de segurança do Safe-Tcl.

detalhes no Capítulo 4. Características da linguagem como *type-safety* e mecanismos como o gerenciador de segurança e o carregador de classes são responsáveis pela implementação da *sandbox*. Diversos sistemas analisados no Capítulo 5 são baseados na linguagem Java como Concordia, D'Agents, SOMA e Ajanta, entre outros.

Tcl é uma linguagem de *scripting* interpretada cujos comandos são implementados em linguagem C. *Scripts* escritos em Tcl podem ser executados de forma segura por meio de um mecanimo chamado de Safe-Tcl [77]. Safe-Tcl utiliza uma abordagem conhecida por ***padded cell*** na qual os *applets* são executados em ambientes isolados e com menos privilégios. Os mecanismos responsáveis pela implementação da *padded cell* estão ilustrados na Figura 3.2.

O **interpretador mestre** retém toda a funcionalidade da linguagem, com todos os privilégios, e nele são executados apenas os *scripts* confiáveis. Por sua vez, o **interpretador seguro** é usado para executar *applets* e mantém apenas um conjunto de comandos, chamado de **base segura**, incapazes de causar qualquer dano ao sistema. Assim, comandos para acesso ao sistema de arquivos e de rede, entre outros, ficam escondidos e não podem ser chamados a partir do interpretador seguro. Um ***alias*** é uma associação de um comando (origem) no interpretador seguro com um comando (destino) no interpretador mestre. Quando o comando origem é chamado o comando destino é executado.

O interpretador mestre tem controle total dos interpretadores seguros e pode chamar os comandos escondidos destes últimos. Isso é importante porque muitos comandos geram resultados que só podem ser usados pelos interpretadores nos quais aqueles foram executados. Assim, por exemplo, imaginemos um *script* destino, usado para limitar os endereços para os quais um *applet* pode conectar, que estabelece uma conexão a um servidor por meio de um comando `socket`. Este comando deve ser executado pelo mesmo interpretador seguro no qual o *applet* que fará uso do canal resultante é executado. De outro modo, o canal ficaria indisponível.

3.3.4 Código assinado

A assinatura digital de código é uma técnica que permite confirmar a autenticidade de um agente móvel e sua integridade. Conseqüentemente é um meio de servidores de agentes verificarem se o agente recebido teve seu código alterado ao longo de suas migrações e, portanto, se pode ser perigoso executá-lo. Por outro lado, note-se que a correta verificação da assinatura não implica que a execução do agente seja segura. A identidade autenticada é utilizada pelos mecanismos de autorização dos servidores para a concessão de permissões aos agentes móveis. Normalmente, o agente é assinado por uma ou mais entidades como o autor do agente, seu proprietário ou uma terceira parte responsável pela revisão do agente.

Diversos sistemas de agentes móveis como Ara, SOMA e Ajanta, analisados no Capítulo 5, utilizam assinaturas digitais com a finalidade descrita. De modo semelhante, a arquitetura de segurança de Java utiliza assinaturas digitais para relaxar as restrições da *sandbox* e conceder mais privilégios a códigos trazidos pela rede (Veja Capítulo 4).

3.3.5 Avaliação de estado (*State appraisal*)

O mecanismo de avaliação de estado proposto por Farmer *et al.* [17] enfoca as informações de estado carregadas pelo agente móvel. A idéia é detectar modificações maliciosas no estado do agente por meio de verificações de invariantes de estado e de restrições como o domínio de uma variável. Estas verificações são feitas por **funções de avaliação** que são usadas para solicitar permissões aos servidores com base no estado atual do agente. Um exemplo de permissão necessária é a de execução em um servidor; assim, se um agente teve seu estado maliciosamente modificado de forma detectável, ele não será executado por qualquer servidor porque sua função de avaliação não requisitará permissão de execução.

As funções de avaliação são criadas pelo autor e pelo proprietário do agente móvel, incluídas no código do agente e protegidas contra modificações por meio de assinaturas digitais. Como os próprios autores de [17] indicam, nem toda modificação maliciosa de estado pode ser detectada pelas funções de avaliação. É possível em muitos casos alterar o estado do agente para refletir um resultado possível, mas forjado.

Como exemplo de uma modificação detectável, podemos imaginar um agente móvel programado para reservar dois lugares em um vôo para Paris na companhia aérea com a melhor oferta. Uma companhia maliciosa pode aumentar o número de reservas a serem realizadas para que as vagas de uma companhia concorrente sejam esgotadas. A função de avaliação detectaria essa modificação do estado e impediria que o agente fosse executado em servidores visitados após a alteração maliciosa. Agora, se considerarmos três variáveis `x`, `y` e `z` que satisfazem a relação `x = y * z`, uma modificação maliciosa de apenas uma variável também é facilmente detectada; porém o atacante pode modificar os valores das

três variáveis de modo a manter a relação de igualdade e enganar assim, a função de avaliação.

3.3.6 Histórico de caminho

A idéia de histórico de caminho [8, 75] é manter um registro dos servidores visitados pelo agente móvel. Quando um servidor recebe um agente, aquele verifica os servidores que este visitou e, dependendo do nível de confiança que deposita nestes servidores, pode recusar o agente recebido ou impor restrições em sua execução. Para que o registro seja feito de forma segura e verificável, quando um agente migrar de um servidor S_i para um servidor S_j, S_i deve incluir uma entrada digitalmente assinada no registro indicando as identidades de S_i (servidor atual) e de S_j (servidor destino). Um problema desta técnica é que os históricos aumentam de tamanho conforme novos servidores são visitados pelo agente móvel. Em decorrência disso, o processo de verificação do histórico fica mais custoso a cada nova migração do agente devido ao aumento de assinaturas digitais para serem verificadas.

3.3.7 *Proof-carrying code*

Proof-carrying code [71, 70, 72] é um mecanismo introduzido por Necula e Lee que permite determinar de forma automática se é seguro executar um programa fornecido por uma parte qualquer, possivelmente maliciosa. Este mecanismo dispensa verificações em tempo de execução, requerendo para tanto que o autor do código forneça juntamente com o programa (em nosso contexto, um agente móvel) uma prova formal de que o agente atende a uma política de segurança (*safety policy*) estabelecida pelo consumidor do código (no caso, um servidor). Antes de executar o agente, o servidor deve então verificar a validade da prova fornecida. Modificações maliciosas no código ou na prova acarretam erros na validação. Se na presença dessas modificações a validação não aponta nenhum erro, temos que o código sofreu uma transformação válida que não infringe a política de segurança especificada [71].

Os autores apontam em [72] características vantajosas de *proof-carrying code* como: eficiência na verificação das provas, que é realizada de forma automática, independência de linguagem de programação, podendo ser aplicado tanto a linguagens de máquina e de alto nível e desnecessidade de se estabelecer relações de confiança com o autor do código.

O artigo [10] descreve ferramentas para utilização de *proof-carrying code* com a linguagem Java, que permitem provar se um dado programa não viola a característica de *type-safety* da linguagem. Características como a carga dinâmica de classes e o uso de *threads* ainda não haviam sido contempladas pelas ferramentas apresentadas.

3.4 Mecanismos de proteção do agente

Enquanto o problema de proteção do servidor contra agentes móveis maliciosos já possui muitas soluções satisfatórias, a proteção dos agentes móveis ainda apresenta muitas lacunas. O problema deriva da dificuldade em se estender o ambiente confiável do servidor origem para outros servidores visitados pelo agente, sem a utilização de *hardware* seguro especializado. Uma primeira impressão que se tem é de que soluções completamente baseadas em *software* são impossíveis porque se entra em um ciclo infinito: os mecanismos providos para se detectar execução incorreta ou modificação do agente podem ser eles próprios violados. Porém, alguns mecanismos mostram que isso não é de todo verdadeiro. Por exemplo, a técnica chamada de computação com funções cifradas (Seção 3.4.6) permite proteger agentes que implementem funções polinomiais e racionais, provendo confidencialidade e integridade de código.

Nas seções seguintes apresentamos diversas técnicas desenvolvidas para a proteção de agentes móveis.

3.4.1 Ciframento deslizante (*Sliding encryption*)

Antes de descrevermos esta técnica, deixemos claro que o real mecanismo de proteção sendo empregado é o ciframento; a técnica em si é utilizada para diminuir os requisitos de espaço no uso de cifras assimétricas. No entanto, como a destinação deste mecanismo era o uso em sistemas de agentes móveis, achamos importante incluí-lo nesta seção.

Ciframento deslizante [120] é um modo de operação para criptossistemas de chave pública proposto por Young e Yung. O objetivo deste modo de operação é permitir cifrar dados muito menores que o tamanho da chave pública utilizada de modo a economizar espaço. Como exemplo, imaginemos um agente móvel que deve coletar dados de 4 bytes em 1024 servidores, cifrando-os com RSA com uma chave de 1024 bits. Cada ciframento resulta em um texto cifrado de 128 bytes e, assim, após visitados todos os servidores, teremos um total de 128 Kbytes de dados cifrados, a partir de apenas 4 Kbytes de texto em claro. O desperdício de espaço é evidente.

Vejamos como funciona o ciframento deslizante com RSA. Vamos supor uma chave de m bytes (m potência de 2), dados coletados a_i de u bytes e $t = u + v$ tal que t divide m e $t \ll m$. O modo utiliza um acumulador A e uma janela W ambos com m bytes divididos em m/t items de t bytes cada, numerados de 1 a m/t. Há também uma pilha S cujos elementos possuem m bytes de tamanho. Tudo isso está ilustrado na Figura 3.3.

Inicialmente, o acumulador A recebe um valor aleatório em $\mathbb{Z}_n^*$. Vamos supor que o agente móvel colete N dados a_i, $i = 0, ..., N-1$. O modo funciona conforme o algoritmo abaixo:

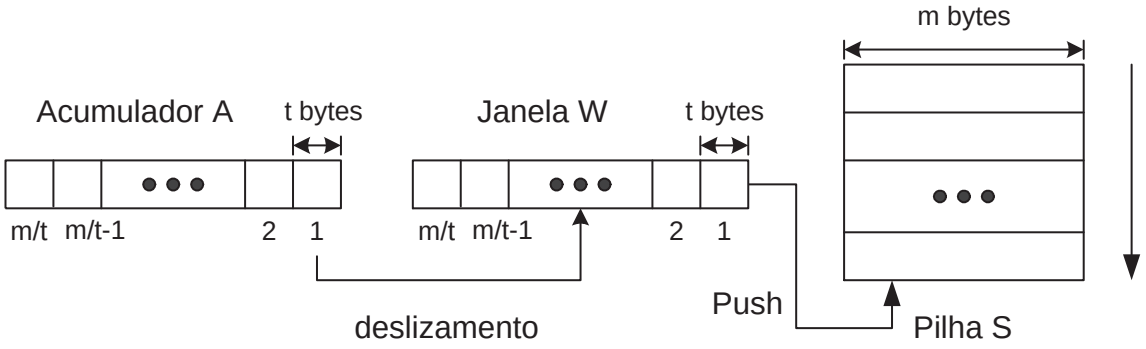

Figura 3.3: Elementos utilizados no modo de operação ciframento deslizante.

```
for i := 0 to N - 1 do {
  A[1] := v bytes aleatórios || a_i;
  A := E(A);     /* aplica RSA em A */
  indice := i mod m/t;
  W[m/t - indice] := A[1];     /* desliza A[1] em W */
  if (m/t - indice == 1)     /* preencheu W, então empilha */
    push(W);
}
```

Para realizar o deciframento basta inverter os passos do algoritmo. Inicialmente, desliza-se os elementos que sobraram em W, um a um a partir do menor índice, para a primeira posição do acumulador, realizando um deciframento a cada iteração. Depois, em cada passo, desempilha-se um elemento que é armazenado em W e procede-se como no passo inicial.

Realizando o ciframento desta maneira, o espaço necessário para N entradas é $2 \times m + N \times t$ bytes. Do modo convencional, seriam necessários $N \times m$ bytes. Para N suficientemente grande e lembrando que $t \ll m$, este modo de operação se apresenta muito mais eficiente em termos do espaço ocupado pelos textos cifrados.

3.4.2 Objetos de detecção

Catherine Meadows descreve em [60] o uso de "objetos de detecção" para permitir detectar modificações não autorizadas em dados coletados por agentes móveis. Tais objetos são dados falsos que são carregados pelos agentes e que não são alterados ou removidos por eles durante sua execução. Se estes dados permanecerem intactos até o momento em que o agente retornar a seu servidor origem, teríamos um indicativo de que os dados legítimos não foram corrompidos também.

O uso de "objetos de detecção" possui muitas desvantagens como: são muito específicos a cada aplicação; é necessário camuflá-los para que não se possa diferenciá-los dos dados verdadeiros; não podem modificar o resultado da tarefa do agente e, a mais grave, apontada por Sander e Tschudin em [88], não possuem a "força criptográfica" necessária para servir como prova em julgamento.

3.4.3 Código de autenticação de resultado parcial

A técnica chamada de Código de Autenticação de Resultado Parcial (PRAC, do inglês Partial Result Authentication Code) foi proposta por Yee [118, 119] e visa proporcionar *forward integrity*. Esta propriedade é definida da seguinte maneira: se um agente móvel visita uma seqüência de servidores $S_1, S_2, ..., S_n$ e o primeiro servidor malicioso é S_c, então nenhum dos resultados obtidos em servidores anteriormente visitados, i.e. S_i com $i < c$, pode ser forjado.

O autor apresenta várias maneiras de se usar PRACs. Na primeira forma, o criador do agente móvel inclui uma lista contendo uma chave para cada um dos servidores a serem visitados. Esta lista é mantida no servidor também até o retorno do agente. Antes de um agente migrar de um servidor S_i para um servidor S_{i+1}, ele deve usar a i-ésima chave, K_i, da lista, para calcular um MAC sobre o dado obtido em S_i. Em seguida a chave K_i é eliminada do estado do agente e só então ocorre a migração para S_{i+1}. Quando o agente retornar ao servidor origem, este pode verificar a integridade dos dados coletados pois possui todas as chaves utilizadas. Por outro lado, servidores maliciosos não podem modificar dados previamente obtidos, porque as chaves necessárias para o cálculo dos MACs são removidas do agente.

Uma outra maneira de usar PRACs é por meio de funções unidirecionais para a geração de chaves. O agente móvel inicia com uma chave K_1 que também é mantida no servidor origem. Quando o agente for migrar de um servidor S_i para outro S_{i+1}, ele gera uma chave $K_{i+1} = f(K_i)$, onde $f(x)$ é uma função unidirecional, e elimina K_i. Dada a unidirecionalidade de $f(x)$, servidores não conseguem recuperar chaves K_i anteriormente utilizadas e assim, não conseguem forjar PRACs de servidores anteriores.

Pode-se observar que nas abordagens anteriores somente o servidor origem é capaz de verificar os PRACs, porque esse processo requer alguma informação que somente ele detém (estamos desconsiderando o caso em que um mesmo servidor é visitado duas vezes). Para possibilitar a verificação em servidores intermediários, Yee propõe o uso de uma lista contendo um conjunto de funções de assinatura digital, $Sig_i(x)$, e outra contendo as respectivas funções de verificação, $Ver_i(x)$. O agente carrega essas duas listas e em um servidor S_i utiliza a função $Sig_i(x)$ para gerar o PRAC do dado ali coletado, eliminando-a antes de visitar S_{i+1}.

Um problema claro deste mecanismo é que todos os dados coletados entre duas visitas a um servidor malicioso podem ser modificados de forma indetectável. O ataque também é possível se um servidor malicioso S_i auxilia um outro servidor malicioso posteriormente visitado, informando-lhe os dados necessários para a geração de PRACs como por exemplo K_i.

3.4.4 Replicação e votação

Servidores de agentes que possuam falhas podem apresentar um comportamento semelhante ao de servidores maliciosos. Assim, Schneider em [90] propõe o uso de técnicas de tolerância a falhas para atenuar os efeitos de servidores maliciosos. A idéia é utilizar diversas réplicas de um agente móvel para a realização de uma tarefa. Uma suposição adotada é que a maioria dos servidores é bem comportada e assim, por meio de votação, é possível desconsiderar a pequena parcela de agentes modificados maliciosamente.

O autor considera a execução de um agente móvel em termos de uma trajetória, que é a seqüência de servidores visitados pelo agente para realização de sua tarefa. O primeiro e o último servidores da seqüência correspondem a um mesmo servidor. Os demais servidores pertencem cada um a um estágio que é replicado para prover a tolerância a falhas. Cada servidor do estágio i recebe/envia de/para cada servidor do estágio $i-1/i+1$ uma réplica do agente correspondente escolhido por meio da votação.

É importante que os mecanismos de votação dos servidores em um dado estágio i saibam quais são os servidores do estágio $i-1$. De outro modo, servidores maliciosos poderiam se passar como pertencentes ao penúltimo estágio, por exemplo, e enviar diversos agentes móveis para o último estágio. Uma solução proposta pelo autor a esse problema é um protocolo que realiza o registro de itinerário dos agentes utilizando assinaturas digitais.

Uma desvantagem natural desta técnica é a carga adicional do sistema decorrente da replicação de servidores e agentes móveis.

3.4.5 Rastros criptográficos (*Cryptographic traces*)

Rastro criptográfico [110, 111] é um mecanismo de detecção proposto por Vigna que permite verificar se a execução de um programa foi corrompida de alguma maneira. A técnica permite detectar modificações maliciosas nos valores de variáveis e no código do programa, desvio de fluxo de execução e cobrança indevida de serviços.

Um agente móvel possui um segmento de código p e um estado, no momento t, $S(t)$. A parte de código é composta por comandos de dois tipos: comandos brancos e comandos negros. Um **comando branco** é aquele que ou não altera o estado do agente ou o altera com base somente em valores de variáveis internas ao agente. Por exemplo, se `x` e `y` são variáveis declaradas pelo agente, `x := y + 2` é um comando branco. Já um **comando**

negro é aquele que altera o estado do agente utilizando informações recebidas do ambiente de execução como, por exemplo, um comando `read(x)`.

Um rastro T é uma seqüência de pares $< n, s >$, onde n é um identificador único de um comando e s é o que Vigna chama de assinatura. No caso de comandos brancos, s é nula e no caso de comandos negros, s assumirá o valor recebido do ambiente de execução. Em [110, 111], temos alguns protocolos baseados no uso de rastros. Como a seqüência de pares $< n, s >$ deve ser única para um programa determinístico a partir de um mesmo estado inicial, quando o resultado calculado por um agente móvel parecer suspeito, basta reexecutar o agente em um simulador e verificar se o mesmo rastro é obtido. Se os rastros diferirem, o protocolo permite determinar qual foi a entidade maliciosa.

Consideremos o seguinte trecho de programa no qual o número no início de cada linha corresponde ao identificador único de comando:

1: if (x < y)
2: z := x * x;
3: else
4: z := y * y;

Para um estado inicial $S(t) = \{x : 5;\ y : 3;\ z : 0\}$, o rastro T seria dado pela seqüência $< 1, \varnothing >< 3, \varnothing >< 4, \varnothing >$ e o estado em $t + 1$ seria $S(t + 1) = \{x : 5;\ y : 3;\ z : 9\}$. Um servidor malicioso pode executar o agente móvel incorretamente e gerar o estado $t + 1$ como $S(t + 1) = \{x : 5;\ y : 3;\ z : 25\}$. Se o proprietário do agente desconfia do resultado obtido, ele poderá verificar o rastro e encontrar a insconsistência. Como o rastro e o estado resultante são assinados digitalmente durante a execução do protocolo, a parte maliciosa não poderá negar que trapaceou.

Uma questão que pode ser percebida a partir do exemplo é como saber se o resultado obtido pelo agente foi calculado corretamente pelos servidores visitados. Para saber isso ou deve-se reexecutar o agente ou então deve-se saber do resultado esperado de antemão. Em ambos os casos, não faria sentido enviar o agente móvel para realizar a tarefa. Outro inconveniente é o tamanho dos rastros que pode se tornar grande, mesmo comprimido.

3.4.6 Computação com funções cifradas (*Encrypted functions*)

O trabalho de Sander e Tschudin [87, 88] foi o primeiro passo para desmistificar a idéia de que é impossível proteger agentes móveis sem o uso de *hardware* seguro, uma vez que os mesmos estão completamente expostos aos servidores. O problema que a técnica, chamada de Computação com Funções Cifradas ou de Criptografia Móvel, visa resolver é apresentado da seguinte maneira pelos autores:

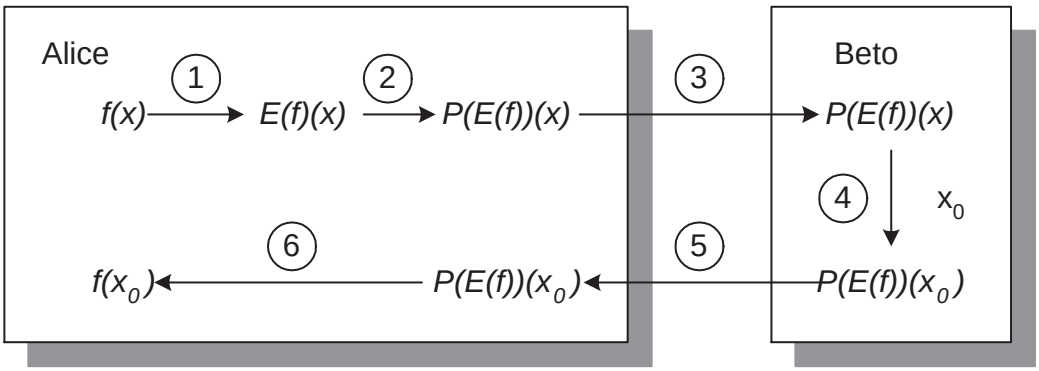

Figura 3.4: Protocolo não interativo para computação com funções cifradas.

> "Alice possui um algoritmo para computar uma função f. Beto possui uma entrada x e deseja computar $f(x)$ para Alice, mas ela não quer que Beto aprenda nada substancial sobre f. Mais ainda, Beto não deve interagir com Alice durante a computação de $f(x)$."

A abordagem proposta requer a diferenciação entre funções e os programas que as implementam. A idéia então é aplicar transformações nas funções antes que elas sejam implementadas, de modo que os servidores não possam obter nenhuma informação substancial sobre a função original. Isso pode ser obtido com o protocolo ilustrado na Figura 3.4, cujos passos são:

1. Alice cifra a função f e obtém $E(f)$;
2. Alice cria um programa $P(E(f))$ que implementa $E(f)$;
3. Alice envia o programa $P(E(f))$ para Beto;
4. Beto executa $P(E(f))$ com o valor x_0;
5. Beto envia $P(E(f))(x_0)$ para Alice; e
6. Alice decifra $P(E(f))(x_0)$ para obter $f(x_0)$.

Os autores conseguiram aplicar o protocolo com sucesso a funções racionais e polinomais usando técnicas de composição de funções e esquemas de ciframento contendo algumas propriedades homomórficas [87, 88]. A possibilidade de se aplicar a técnica a funções arbitrárias ainda é uma questão não resolvida.

Um exemplo simples é o de como proteger a matriz A de uma função $f(x) = Ax$ que calcula multiplicação de matrizes. A solução utiliza composição de funções e é realizada

da seguinte forma: Alice gera aleatoriamente uma matriz inversível S. Ela então calcula $E(f(x)) = SAx$ e envia $E(f(x))$ para Beto. Este avalia a função com sua entrada x_0 e retorna $E(f(x_0))$ para Alice, que obtém $f(x_0)$ calculando $S^{-1}E(f(x_0)) = S^{-1}SAx_0 = IAx_0 = Ax_0 = f(x_0)$.

Uma implicação direta do uso de funções cifradas que se pode perceber é que se obtém confidencialidade de código e integridade também, no sentido de não ser possível modificar o programa de modo a se conseguir um resultado específico. Com isso, seria possível a um agente móvel carregar, por exemplo, uma chave privada para geração de assinaturas digitais em servidores não confiáveis.

3.4.7 Contêiner somente-para-inclusão e estado direcionado

Estes mecanismos propostos por Karnik e Tripathi [45, 46, 47] fazem parte da arquitetura de segurança do sistema de agentes móveis Ajanta (Seção 5.7). O contêiner somente-para-inclusão (*append-only container*) permite incluir dados obtidos em servidores e protegê-los contra modificações e remoções. Já o estado direcionado (*targeted state*) pode ser usado pelo proprietário do agente para encapsular, de forma segura, dados destinados a servidores específicos.

Um objeto `AppendOnlyContainer` contém três vetores que armazenam os objetos incluídos e as respectivas assinaturas digitais e identidades dos assinantes. Além disso, possui um *checksum* que permite que o servidor de origem detecte modificações indesejáveis no contêiner. Ao criar um objeto deste tipo, o servidor de origem A gera um *nonce* N_A que deve ser mantido em segredo e que é utilizado para calcular o *checksum* inicial da maneira a seguir:

$$checkSum := E_{K_A}(N_A)$$

Para incluir um novo elemento X no contêiner, em um servidor B, o agente móvel utiliza o método `checkIn()`. Esta rotina solicita inicialmente que o servidor B assine digitalmente o objeto X e, em seguida, armazena X, $Sig_B(X)$ e a identidade B nos vetores correspondentes. O novo *checksum* é calculado da seguinte forma:

$$checkSum := E_{K_A}(checkSum \mid\mid Sig_B(X) \mid\mid B)$$

Quando o agente móvel retorna ao servidor de origem, pode-se verificar, por meio do método `verify()`, se o contêiner não foi violado. A cada iteração do processo de verificação, o *checkSum* é decifrado, obtendo-se como resultado o *checkSum* anterior, $Sig_B(X)$ e a identidade B. Estes últimos são comparados com os objetos contidos nos vetores de

assinaturas e de identidades, no índice relacionado à iteração atual. Se forem iguais, a assinatura $Sig_B(X)$ é verificada. O processo é repetido até que alguma inconsistência seja encontrada ou até que se obtenha o *checkSum* original. Neste ponto, este é decifrado e o valor resultante deve ser igual a N_A.

Como já mencionado, o estado direcionado permite a um agente móvel carregar objetos que são destinados ao uso em servidores específicos. Como exemplo, podemos citar um serviço de conteúdo personalizado destinado apenas aos usuários assinantes. Este mecanismo é implementado pela classe `TargetedState` cujas instâncias são compostas de um vetor de objetos direcionados, `objs`, e outro de identidades de servidores, `servers`. Para cada índice i possível, `objs[i]` contém um objeto cifrado com a chave pública da entidade representada por `servers[i]`.

Cada objeto `TargetedState` possui também uma assinatura digital gerada pelo proprietário do agente sobre a concatenação do vetor de objetos direcionados com o vetor de identidades de servidores. Cada servidor visitado, antes de decifrar um objeto a ele destinado, pode verificar a assinatura digital sobre os objetos para certificar-se de que nenhuma alteração maliciosa ocorreu.

3.4.8 Geração ambiental de chave (*Environmental key generation*)

Esta técnica descrita por Riordan e Schneier em [82] permite construir o que os autores chamam de agentes sem pista (*clueless agents*). Um agente sem pista não permite que a função que executa seja determinada pela simples análise de seu código. Isso é obtido cifrando-se a parte de código que se deseja proteger com uma chave que só pode ser construída sob a presença de determinados dados ou condições do ambiente de execução. As condições necessárias para a construção da chave são protegidas por meio do uso de funções de *hashing* criptográficas de modo semelhante ao usado para a proteção de senhas em sistemas Unix.

Os autores citam como exemplos de dados que podem ser coletados nos ambientes para a geração de chaves: palavras ou mensagens em correio eletrônico ou grupos Usenet, nomes de arquivos, nomes de domínio DNS e dados temporais. Uma preocupação necessária é que não se deve utilizar um domínio de dados com poucos elementos o que possibilitaria realizar ataques de dicionário.

Para um melhor entendimento, vejamos um dos exemplos dados em [82]. Alice teve uma idéia a qual ela gostaria de patentear. Mas para verificar a originalidade de sua idéia ela precisa pesquisar um banco de dados de patentes, sem contudo revelar que sua idéia é construir um "detector de fumaça com alarme". Alice inicialmente calcula:

1. $N :=$ um *nonce* aleatório

2. $K := H(\text{“detector de fumaça com alarme”})$

3. $M := E_K(\text{“notificar resultado para alice@weaseldyne.com”})$

4. $O := H(N \texttt{ xor } \text{“detector de fumaça com alarme”})$

Em seguida ela cria um agente móvel que realiza a pesquisa no(s) banco(s) de dados de patentes da seguinte maneira, em pseudocódigo:

for toda seqüência de cinco palavras (x) **do**
  if $(H(N \texttt{ xor } x) = O)$ {
    $P := D_{H(x)}(M)$;
    execute P;
  }

Dessa forma, o agente não carrega pistas da patente que está procurando e o dono do banco de dados de patentes só terá conhecimento da idéia de Alice se já possuir a idéia cadastrada.

3.4.9 Agentes cooperantes

Roth introduz em [83, 84] o conceito de agentes cooperantes e mostra protocolos utilizando esses agentes. Tais agentes são definidos em termos dos servidores a serem visitados: sejam H_A e H_B dois conjuntos de servidores não vazios e disjuntos. Um agente cooperante A deve ter em seu itinerário somente servidores contidos em H_A; enquanto isso, o agente B, par de A, somente deve visitar servidores contidos em H_B. Assim, A e B nunca visitam ambos um mesmo servidor S qualquer.

As seguintes suposições são feitas pelos protocolos descritos:

- para quaisquer pares (S_P, S_Q), tal que $S_P \in H_A$ e $S_Q \in H_B$, temos que o servidor S_P não coopera com S_Q na realização de um ataque;
- servidores provêem um canal autenticado para os agentes cooperantes;
- as identidades autenticadas dos servidores atual, destino e origem são disponibilizadas ao agente hospedado.

Roth diz que um ataque simples a um agente móvel por um servidor malicioso é impedir que o agente migre para servidores de concorrentes. Um dos protocolos propostos pelo autor trata desse problema realizando o registro do itinerário entre os agentes cooperantes. Antes de um agente móvel A realizar uma migração ele deve enviar as identidades dos servidores destino (N_i) e o anteriormente visitado (P_i) ao agente cooperante B, através

do canal autenticado. Com isso, B obtém a identidade do servidor atual H_i e verifica se $H_i = N_{i-1}$ e $P_i = H_{i-1}$. Uma forma simples de ataque que vemos é interferir nas saídas do algoritmo que decide qual o próximo servidor a ser visitado pelo agente. Assim, somente o itinerário é alterado, sem prejudicar a execução do protocolo.

O outro protocolo descrito permite utilizar agentes móveis para compras com dinheiro eletrônico. Cada servidor visitado fornece uma oferta para o produto procurado pelo agente de pesquisa. Essa oferta é assinada digitalmente pelo servidor e enviada pelo agente ao seu par cooperante que é quem mantém os valores coletados em cada loja virtual. Dessa forma, o agente cooperante fica responsável pela decisão de compra e os demais servidores não tem acesso aos preços dos concorrentes. Por fim, o montante de dinheiro eletrônico só é liberado por meio da cooperação de ambos os agentes.

Um grande problema na técnica dos agentes cooperantes são as suposições adotadas para os protocolos. Para podermos dizer com alto grau de certeza que não haverá cooperação em um ataque entre os servidores de um par (S_P, S_Q) anteriormente definido, seria necessário saber quais servidores se comportariam de forma maliciosa durante a execução do agente. E de posse dessa informação, parece mais fácil, simplesmente, impedir que os agentes passem por tais servidores. Além disso, servidores maliciosos podem não prover o canal autenticado entre os agentes cooperantes e fornecer identidades inválidas para o agente.

3.4.10 Caixa-preta com prazo de validade (*Time limited black-box*)

A abordagem de Hohl [33] para o problema de servidores maliciosos está centrada na idéia de um agente caixa-preta. Um agente deste tipo não possibilita que o atacante obtenha qualquer informação útil sobre a tarefa do agente e os dados por ele carregados. Além disso, não é possível modificar tal agente de forma controlada, restando apenas alterações aleatórias com resultados inesperados. O problema da abordagem é que não se conhece nenhum algoritmo capaz de gerar agentes que atendam à propriedade caixa preta dada uma especificação qualquer. O autor cita o uso de funções cifradas [88] como uma técnica para geração de agentes caixa-preta, mas que se aplicam somente a funções polinomiais e racionais.

Como a única técnica conhecida para se prover agentes caixa preta não se aplica a qualquer tipo de agente, Hohl fornece uma variação do mecanismo que é acrescido de um prazo de validade. Com essa variação, os agentes não ficam protegidos para sempre e sim, somente por um prazo pré-estabelecido. Ataques ao agente são possíveis após o prazo de proteção, mas são inofensivos dada a validade dos dados e código do agente. A vantagem obtida com a adição da data de expiração é que essa caixa preta relaxada pode ser obtida

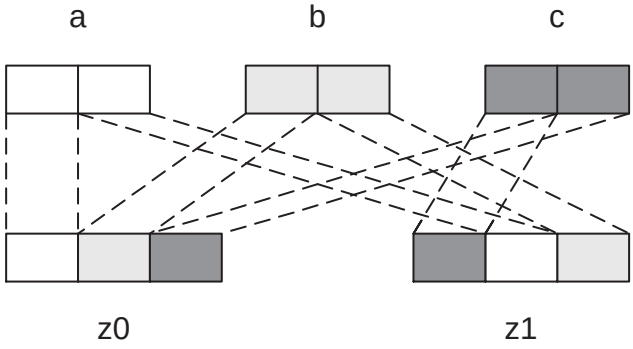

Figura 3.5: Recomposição de variáveis.

por meio de ofuscação de dados e código. Algoritmos de ofuscação recebem como entrada um programa P e produzem um novo programa $O(P)$ com mesma funcionalidade de P, mas que é ininteligível de alguma forma. Um exemplo desta classe de algoritmos seria a recomposição de variáveis que, por meio da segmentação e reorganização do conjunto original de variáveis do programa, gera um novo conjunto de variáveis substitutas. A Figura 3.5 ilustra esta idéia.

Alguns dos problemas das técnicas de ofuscação estão nas questões de como se determinar o tempo de proteção necessário para que um agente móvel realize alguma tarefa útil e como se calcular, a partir dos algoritmos de ofuscação, qual o intervalo de proteção mínimo garantido.

3.4.11 Protocolos KAG

Karjoth, Asokan e Gülcü estenderam o trabalho de Yee [118, 119] descrevendo uma família de protocolos destinados à proteção de dados coletados por agentes móveis em diversos servidores [43]. Os protocolos consideram um agente móvel que visita n servidores $S_1, ..., S_n$. O servidor de origem é chamado de S_0 e é igual a S_{n+1}, isto é, o agente retorna ao servidor origem após visitado o último servidor. Um dado o_i coletado em um servidor S_i é encapsulado por um mecanismo de proteção dependente do protocolo e representado por O_i, que é chamado de oferta encapsulada. Cada O_i por sua vez é relacionado a O_{i-1} e O_{i+1} por uma relação de encadeamento incluída em O_i.

Baseado na idéia de *forward integrity* de Yee, os autores definem *strong forward integrity* da seguinte forma: nenhuma oferta encapsulada O_k obtida em um servidor S_k pode ser modificada, para $k < m$. Uma suposição adotada é de que a oferta O_m não pode ser modificada ou excluída. Concordamos com Roth [85] que não há motivos para acreditar que um servidor malicioso não alteraria O_m, se necessário.

Vejamos o protocolo P2 descrito em [43], para um exemplo:

1. Oferta encapsulada:
   - $O_i = E_{K_{S_0}}(Sig_{S_i}(o_i) \mid\mid N_i), h_i,\ 0 \leq i \leq n$
2. Relação de encadeamento:
   - $h_0 = H(N_0 \mid\mid S_1)$
   - $h_i = H(O_{i-1} \mid\mid N_i \mid\mid S_{i+1}),\ 1 \leq i \leq n$
3. Mensagens do protocolo:
   - $S_i \longrightarrow S_{i+1} : \{O_k \mid 0 \leq k \leq i\}$

Analisando o protocolo, podemos verificar que, dada a suposição de que O_m não seja alterada, tem-se *strong forward integrity*. Digamos que O_k seja alterada, tal que $k = m-1$, gerando uma nova oferta O'_{m-1}. Para que a relação de encadeamento se mantenha em m, devemos ter $H(O_{m-1} \mid\mid N_m \mid\mid S_{m+1}) = H(O'_{m-1} \mid\mid N_m \mid\mid S_{m+1})$. Mas isso contradiz o fato da função $H()$ ser livre de colisões. Da mesma forma, não é possível alterar O_k, com $k < m-1$, sem modificar O_{k+1} e, assim, a propriedade segue por indução. Além disso, o valor N_m está cifrado com a chave pública do servidor de origem e, portanto, indisponível a um atacante. Se O_m puder ser modificada e o atacante, de alguma forma, tiver acesso a seqüência N_i, $k \leq i \leq m$, é possível modificar o k-ésimo elemento. Basta substituir os valores h_i, $k < i \leq m$, de forma a manter as relações de encadeamento. Isso pode ser facilmente corrigido, simplesmente modificando o protocolo para que cada servidor S_i assine também h_i.

3.4.12 *Hardware* seguro

Em [115, 116], Uwe Wilhelm propõe o uso de *hardware* seguro para garantir a privacidade e a correta execução de agentes. O elemento principal da abordagem apresentada é o Ambiente de Processamento Confiável (Trusted Processing Environment - TPE). O TPE é um microcomputador completo com CPU, RAM, ROM e armazenamento não-volátil e é implementado como um *hardware* seguro. A função principal do TPE é prover um ambiente de execução para os agentes móveis que garante uma determinada política de segurança. Cada TPE possui uma chave privada única de conhecimento somente do TPE.

Um servidor de agentes é composto por um *host* conectado a um TPE. O TPE só pode ser acessado por meio de uma interface bem definida que permite, entre outras coisas, a carga e a remoção de agentes. Assim, não é possível acessar dados contidos no TPE diretamente.

O uso do ambiente descrito é especificado pelo protocolo CryPO (Cryptographically Protected Objects) que se preocupa com a transferência segura de agentes para o TPE.

Antes de se realizar uma migração, verifica-se inicialmente se o TPE destino atende aos requisitos de segurança do agente móvel. Em caso positivo, o agente é cifrado com a chave pública do TPE destino e então transferido para o *host* associado. Este então carrega o agente no TPE.

Em [116], o autor diz que para garantir a integridade do agente em trânsito, basta concatená-lo com um *hash* dos dados e códigos do agente antes do ciframento. Na verdade, isso não é suficiente, pois um atacante poderia simplesmente forjar um agente, recalcular o *hash* e cifrá-lo novamente antes de transferi-lo para o servidor destino. Para obter o resultado desejado, é necessário utilizar assinaturas digitais.

Uma abordagem semelhante é apresentada em [22] por Funfrocken. O autor descreve o projeto WASP (Web Agent Based Service Providing) que integra as funcionalidades de sistemas de agentes móveis em servidores Web. A proteção de agentes é obtida por meio do uso de Java *cards*, que são mais simples e menos poderosos que os *hardwares* usados na proposta de Wilhelm.

Os Java *cards* são utilizados para autenticação e assinatura dos agentes e como uma base computacional confiável. Dadas as limitações do Java *card*, ele comporta apenas parte do estado e do código dos agentes hospedados. Esta parte é chamada de ***code parts*** e é protegida de terceiros sendo cifrada com a chave pública dos Java *cards* destinos.

Um ponto a se observar no uso de *hardware* seguro é que transfere-se a confiança que se tem nos servidores de agentes para o fabricante do *hardware* seguro. Uma coisa que se pode perguntar é o que se ganha com essa transferência. Wilhelm em [115, 116] cita como vantagens, entre outras, o número reduzido de fabricantes de *hardware* seguro em relação ao número de servidores de agentes e a alta experiência em segurança dos fabricantes, por serem provedores de serviço especializado.

3.4.13 Protocolos SOMA para integridade

O sistema de agentes móveis SOMA (Seção 5.6) apresenta duas soluções para proteger a integridade do estado de um agente móvel contra ataques realizados por entidades maliciosas: a primeira depende de uma terceira parte confiável (TTP) que valida os dados coletados pelo agente a cada migração; a segunda consiste de um protocolo distribuído, chamado de **múltiplos-saltos** (*multiple-hops*), que dispensa o uso da TTP. Ambas as soluções esperam que o agente seja composto de três partes: (i) código e dados de inicialização (CID), que correspondem à parte imutável do agente; (ii) dados de aplicação (AD) coletados pelo agente nos servidores visitados; e (iii) dados do protocolo (PD) gerados para a verificação da integridade do estado do agente. Nos protocolos, os servidores visitados são identificados por S_i, $1 \leq i \leq n$. S_0 corresponde ao servidor origem.

Na solução baseada na terceira parte confiável, o agente móvel sempre carrega um

MDC calculado pela TTP sobre os dados coletados. Toda vez que um agente desejar migrar de um servidor S_i para S_{i+1}, ele deve primeiramente passar pela TTP. Esta verifica o MDC previamente calculado para assegurar que nenhum dado coletado anteriormente foi modificado maliciosamente pelo último servidor. Depois disso, um novo MDC é gerado e o agente é enviado para S_{i+1}. O protocolo é ilustrado a seguir:

1. Inicialização:
   - $PD_1 = MDC_1 = H(CID)$
   - $S_0 \longrightarrow S_1 : (CID,\ PD_1)$
2. Em cada servidor $S_i,\ i \geq 1$:
   - $AD_i =$ dado obtido em S_i
   - $S_i \longrightarrow$ TTP $: (CID,\ PD_i,\ AD_i)$
3. Na TTP:
   - Verificação de MDC_i
   - $MDC_{i+1} = H(CID \mid\mid (PD_i,\ AD_i))$
   - $PD_{i+1} = (PD_i,\ AD_i,\ MDC_{i+1})$
   - TTP $\longrightarrow S_{i+1} : (CID,\ PD_{i+1})$

No protocolo múltiplos-saltos, o agente pode migrar para quaisquer servidores sem a necessidade de utilizar uma TTP. Durante a inicialização do protocolo, o servidor de origem do agente móvel gera um valor C_0 aleatoriamente, que deve ser mantido em segredo. Um servidor S_i, $i \geq 0$, calcula $C_{i+1} = H(C_i)$, que é cifrado com a chave pública do próximo servidor, S_{i+1}. O agente móvel carrega para S_{i+1} somente o valor C_{i+1} e não o valor C_i. Cada servidor S_i visitado pelo agente móvel calcula um MDC sobre o MDC anterior, os dados D_i incluídos por S_i, C_i e sobre a identidade S_{i+1}. O MDC é digitalmente assinado por S_i e armazenado com a assinatura pelo agente que é então enviado a S_{i+1}. Os passos do protocolo são abaixo ilustrados:

1. Inicialização:
   - $C_1 = H(C_0)$, C_0 gerado aleatoriamente
   - $MDC_0 =$ void
   - $AD_0 =$ void
   - $PD_0 = E_{K_{S_1}}(C_1)$

- $S_0 \longrightarrow S_1 : (CID,\ AD_0,\ PD_0)$

2. Em cada servidor $S_i,\ i \geq 1$:

   - $C_{i+1} = H(C_i)$
   - $MDC_i = H(D_i \mid\mid C_i \mid\mid MDC_{i-1} \mid\mid S_{i+1})$
   - $AD_i = (AD_{i-1}, D_i, Sig_{S_i}(D_i))$
   - $PD_i = (E_{K_{S_{i+1}}}(C_{i+1}),\ MDC_1, Sig_{S_1}(MDC_1), ..., MDC_i, Sig_{S_i}(MDC_i))$
   - $S_i \longrightarrow S_{i+1} : (CID,\ AD_i,\ PD_i)$

Quando o agente retorna ao servidor origem, este pode verificar se o estado do agente sofreu alguma modificação não autorizada. Para isso, basta que S_0 reconstrua a cadeia de MDCs calculados em cada servidor S_i a partir do segredo C_0 de conhecimento apenas de S_0. A cada MDC recalculado, S_0 verifica a assinatura de S_i sobre MDC_i. O processo prossegue até que se obtenha MDC_n ou que uma assinatura digital seja inválida.

Algumas análises sobre esses protocolos podem ser encontradas na Seção 5.6.

3.4.14 Assinatura com chave blindada (*Blinded-key signature*)

Assinatura com chave blindada é uma técnica proposta por Ferreira e Dahab [20, 19] que permite que um agente móvel gere assinaturas em nome de seu proprietário sem a necessidade de interação com o mesmo. O agente móvel gera assinaturas usando uma chave blindada resultante de uma transformação aplicada à chave privada por meio de um fator de blindagem (*blinding factor*). Com isso, os servidores e demais entidades não conseguem recuperar a chave privada do usuário, a partir do código e estado do agente. O fator de blindagem deve ser registrado por um notário juntamente com o identificador do agente móvel que usará a respectiva chave blindada e a política de uso desta chave. O notário é uma terceira parte confiável que é responsável pelo processo de verificação de assinaturas.

Para maior clareza das idéias apresentadas, vejamos como Alice pode criar um agente móvel que utilize uma chave RSA blindada. Inicialmente, ela deve blindar o expoente privado d da seguinte maneira:

1. Escolha um inteiro b (o fator de blindagem) aleatório, com $1 < b < \phi(n)$.

2. O expoente blindado d_b é calculado como $d_b = db \bmod \phi(n)$.

O expoente d_b será utilizado pelo agente móvel para gerar as assinaturas RSA para Alice. Digamos que este agente A seja unicamente identificado por ID_A. Alice envia uma mensagem ao notário contendo a política de uso de d_b, o identificador ID_A e o fator de

blindagem b. A política pode determinar, entre outras coisas, o tempo de validade da chave blindada e o número de assinaturas que podem ser geradas com ela.

Alice libera o agente A para percorrer os servidores em busca de um produto que ela deseja adquirir. Após visitar vários servidores, A determina que a melhor oferta foi dada por um servidor S_M. O agente migra para S_M onde então deve assinar digitalmente uma ordem de pagamento M. A assinatura gerada é

$$Sig = M^{d_b} \bmod n$$

A loja virtual solicita que o notário verifique a assinatura Sig, o que é feito seguindo-se os passos abaixo:

1. Calcule $T_1 = Sig^e \bmod n$ e $T_2 = M^b \bmod n$.
2. Aceite a assinatura se e somente se $T_1 = T_2$ e se a mesma não viola a política de uso estabelecida para a chave blindada em questão.

Em caso de uma verificação positiva, o servidor S_M aceita a ordem de pagamento e notifica o agente, que retorna ao servidor origem.

A prova de corretude do esquema apresentado é similar à prova de corretude do criptossistema RSA e esta última pode ser encontrada com riqueza de detalhes em [13]. Uma generalização das assinaturas com chaves blindadas e exemplos da aplicação da técnica a outros esquemas de assinatura como ElGamal e DSA podem ser encontrados em [20].

Um ponto importante a ser considerado é o de políticas de uso, pois é o que limita o forjamento de assinaturas. Se por um lado as políticas devem ser restritas o suficiente para impedir que um servidor malicioso gere uma assinatura válida mas resultante de uma incorreta execução do agente, elas devem ser flexíveis o bastante para não descaracterizar o paradigma de agentes móveis. Meios para se especificar políticas e verificar a conformidade de assinaturas a elas não são abordados pelos autores em [20, 19].

3.5 Sumário dos mecanismos de proteção

Nas Seções 3.3 e 3.4, foram analisados diversos mecanismos para proteção de servidores e agentes, que estão resumidos na Tabela 3.1. Estes mecanismos podem ser classificados conforme sirvam para detecção ou prevenção de ataques. **Mecanismos de prevenção** visam impedir que o atacante possa modificar o código ou os dados de uma forma controlada, restando-lhe apenas efetuar alterações aleatórias, que o impossibilita de conseguir um resultado específico. Por sua vez, **mecanismos de detecção** permitem que se possa verificar a ocorrência de um dado ataque. Como a detecção ocorre sempre após a realização do ataque, é importante que haja meios de se identificar o infrator ou então de

se reduzir a probabilidade de que novos ataques sejam perpetrados. De outra forma, tais mecanismos teriam valor prático muito pequeno. A Tabela 3.2 classifica os mecanismos analisados nas seções anteriores de acordo com essas duas categorias.

Mecanismo	Utilização
Proteção do servidor	
Monitor de referências	controle de acesso
Isolamento de falhas baseado em SW	isolamento de falhas entre módulos
Interpretação segura de código	controle sobre a execução de comandos considerados perigosos
Código assinado	autenticidade e integridade do código
Avaliação de estado	detecção de modificações maliciosas do estado
Histórico de caminho	autorização baseada nos servidores visitados
Proof-carrying code	verificação automática de se é seguro executar um dado programa
Proteção do agente	
Ciframento deslizante	confidencialidade de pequenos dados coletados nos servidores visitados
Objetos de detecção	integridade de dados coletados em servidores visitados pelo agente
Código de autenticação de res. parcial	integridade de dados coletados em servidores visitados pelo agente
Replicação e votação	integridade da execução de um agente
Rastros criptográficos	auditoria da execução do agente
Computação com funções cifradas	confidencialidade de código e dados do agente
Contêiner somente-para-inclusão	integridade de dados coletados em servidores visitados pelo agente
Estado direcionado	confidencialidade, integridade e autenticidade de dados destinados a entidades específicas
Geração ambiental de chave	sigilo de um trecho de código até que uma dada condição ambiental seja verdadeira
Agentes cooperantes	execução correta de uma dada tarefa, sob certas suposições
Caixa-preta com prazo de validade	confidencialidade de código e dados do agente em um certo intervalo de tempo
Protocolos KAG	integridade de dados coletados em servidores visitados pelo agente
Hardware seguro	confidencialidade e integridade dos dados e código do agente
Protocolos SOMA	integridade de dados coletados em servidores visitados pelo agente
Assinatura com chave blindada	proteção de chaves privadas para assinaturas

Tabela 3.1: Resumo dos mecanismos de proteção de servidores e agentes.

Mecanismo	Tipo
Proteção do servidor	
Monitor de referências	prevenção
Isolamento de falhas baseado em SW	prevenção
Interpretação segura de código	prevenção
Código assinado	detecção
Avaliação de estado	detecção
Histórico de caminho	detecção
Proof-carrying code	prevenção
Proteção do agente	
Ciframento deslizante	prevenção
Objetos de detecção	detecção
Código de autenticação de res. parcial	detecção
Replicação e votação	prevenção
Rastros criptográficos	detecção
Computação com funções cifradas	prevenção
Contêiner somente-para-inclusão	detecção
Estado direcionado	prevenção
Geração ambiental de chave	prevenção
Agentes cooperantes	prevenção
Caixa-preta com prazo de validade	prevenção
Protocolos KAG	detecção
Hardware seguro	prevenção
Protocolos SOMA	detecção
Assinatura com chave blindada	prevenção

Tabela 3.2: Classificação dos mecanismos de proteção.

Capítulo 4

Arquitetura de Segurança de Java

Quando a linguagem Java foi introduzida em 1995, ela atraiu a atenção de desenvolvedores no mundo inteiro, devido à facilidade de programação, independência de plataforma, robustez e segurança. Tais características eram resultado da linguagem ter sido projetada desde o início para atender às necessidades de ambientes interconectados por rede. Desde então, ela se tornou uma das plataformas de desenvolvimento mais populares do planeta e deixou de ser utilizada apenas para a criação de *applets*. Atualmente, Java é usada como uma linguagem de programação de propósito geral para o desenvolvimento das mais diversas aplicações para os mais diversos dispositivos e computadores, de *smart cards* a servidores.

O modelo de segurança original de Java, chamado de *sandbox*, oferecia um ambiente restrito para a execução de *applets* recebidos pela rede. Enquanto isso, código Java originário da própria máquina era considerado confiável e, portanto, tinha acesso total aos recursos do sistema. Essa política tudo ou nada provida pela *sandbox* muitas vezes apresentava-se excessivamente restritiva e, até então, a implementação de uma política particular mais flexível exigia uma trabalhosa tarefa de programação, sempre susceptível a erros de codificação ou lógica. Nas versões seguintes da plataforma Java, o modelo foi sendo modificado para facilitar a criação de políticas de segurança, com o mínimo de programação necessário.

As características acima citadas da linguagem Java a tornam uma candidata adequada ao uso em sistemas de agentes móveis, nos quais a mobilidade de código e a execução em diferentes plataformas são a tônica. De fato, Java é muito utilizada na implementação desses sistemas, que aproveitam-se de vários mecanismos da arquitetura de segurança para barrar algumas das ameaças descritas nas Seções 3.1.2 e 3.1.3.

O restante deste capítulo está organizado da seguinte maneira: a evolução do modelo de segurança de Java [14, 23, 25, 108, 59] é apresentada na Seção 4.1. Nas seções 4.2 a 4.5 discutimos os elementos que compõem a *sandbox* básica e nas Seções 4.6 a 4.8,

as adições realizadas na plataforma Java 2. Na Seção 4.9, mostramos os pontos não contemplados pelo modelo de segurança atual e alguns *bugs* relacionados à segurança encontrados em versões mais antigas da plataforma. Por fim, na Seção 4.10 descrevemos como os mecanismos de Java podem ser empregados na segurança de sistemas de agentes móveis. Boas referências para segurança de Java são o *site* da Sun [62] e os livros [59, 108, 73].

4.1 Evolução do modelo de segurança

O modelo de segurança do Java Development Kit (JDK) 1.0 é conhecido como modelo *sandbox*. Neste modelo, classes residentes[1] são consideradas confiáveis e possuem privilégios para acessar recursos vitais do sistema como sistema de arquivos, serviços de rede e impressora. Enquanto isso, códigos provenientes da rede não são considerados confiáveis e são executados dentro de uma *sandbox* que limita os recursos que podem ser utilizados. A *sandbox* original era bastante restritiva e impedia até programas bem intencionados, mas provenientes de fontes não confiáveis, de realizar qualquer tarefa útil. O modelo é ilustrado na Figura 4.1.

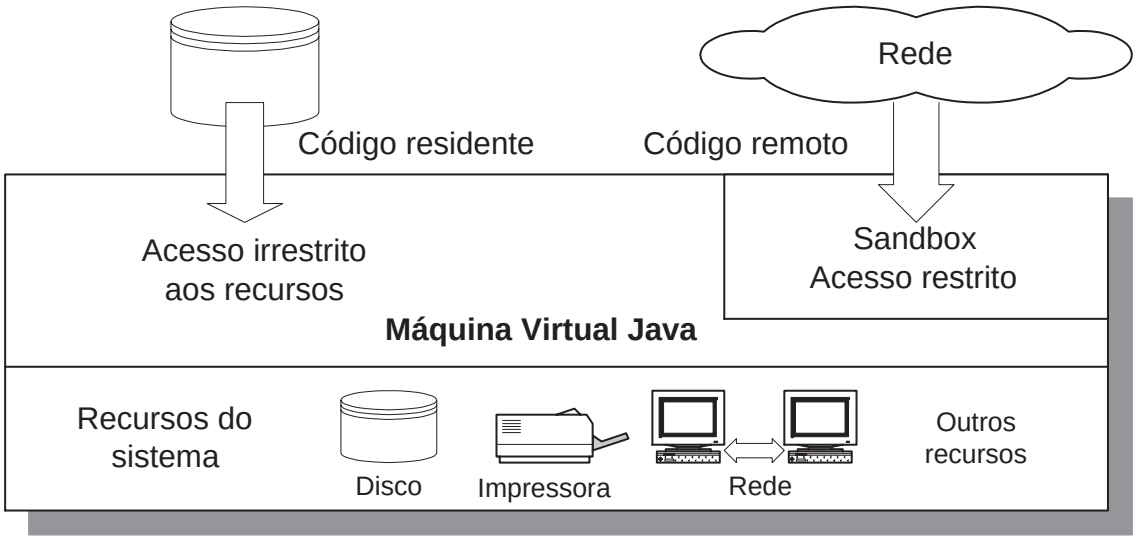

Figura 4.1: Modelo de segurança do JDK 1.0.

A *sandbox* básica é implementada por uma série de mecanismos:

- características da linguagem e da máquina virtual;

[1] Usamos esse termo para nos referirmos às classes encontradas na própria máquina.

- verificador de arquivos `class`;
- carregador de classes; e
- gerenciador de segurança.

No JDK 1.1, introduziu-se o uso de código digitalmente assinado. Neste modelo estendido, apresentado na Figura 4.2, uma classe remota pode ser executada com acesso total aos recursos do sistema, como se fosse uma classe residente (e portanto confiável), se for digitalmente assinada por uma entidade na qual se tem confiança. Por outro lado, classes remotas não assinadas continuam a ser executadas dentro da *sandbox*. Um conjunto de classes e sua assinatura são agrupados em um arquivo JAR.

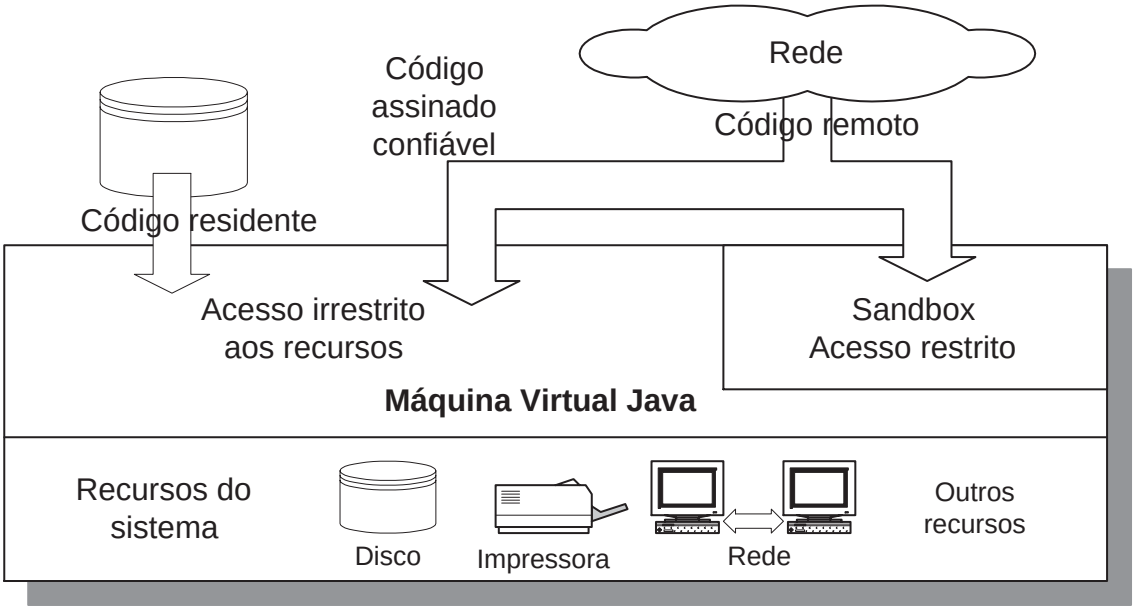

Figura 4.2: Modelo de segurança do JDK 1.1.

A Figura 4.3 mostra a arquitetura de segurança da plataforma Java 2. As principais características focadas nesta nova versão são:

- controle de acesso granular - permite relaxar as restrições da *sandbox* de forma gradual, por meio do controle de acesso a recursos mais específicos. Com isso, não é necessário, por exemplo, dar a uma classe que apenas necessita trabalhar com o arquivo `qualquer.txt` permissão para escrever no sistema de arquivos inteiro. A nova arquitetura torna esta tarefa mais simples e segura que anteriormente, quando era necessário estender as classes `SecurityManager` e `ClassLoader` escrevendo código muito complexo e susceptível a falhas de segurança.

- política de segurança configurável - embora também existente nas versões anteriores, a criação de uma política de segurança requeria muita programação. Em Java 2, isso é realizado mais facilmente, por meio de permissões concedidas em arquivos.

- estrutura de controle de acesso extensível - facilita a adição e verificação de novas permissões de acesso através do uso de permissões tipadas, cada uma representando o acesso a um determinado recurso do sistema. Nas versões anteriores isso era feito adicionando-se um método `check` na classe `SecurityManager` para cada nova permissão.

- controle de segurança para todos os programas Java - eliminação do conceito de que todas as classes residentes são confiáveis. A política de segurança passa a ser aplicada a toda e qualquer classe.

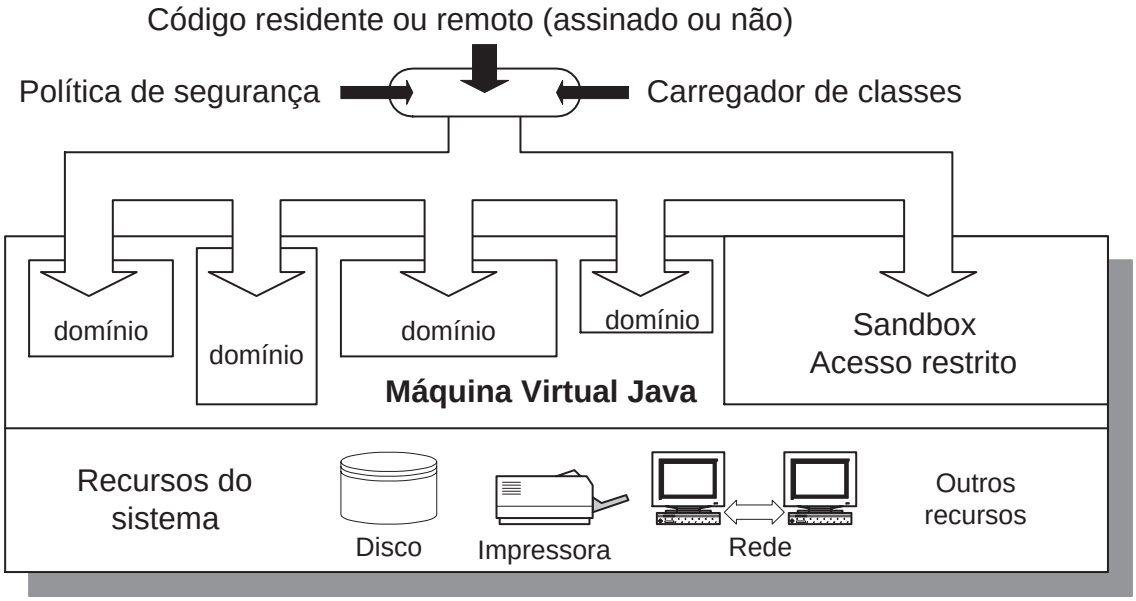

Figura 4.3: Modelo de segurança da plataforma Java 2.

Nas Seções 4.2 a 4.5 detalhamos os componentes do modelo de segurança de Java responsáveis por implementar a *sandbox* básica. Já os novos mecanismos de proteção adicionados à plataforma Java 2 são abordados nas Seções 4.6 a 4.8.

4.2 Características da linguagem e da máquina virtual Java

Muitas características da linguagem Java são importantes para o modelo de segurança utilizado. Elas tornam os programas robustos e são asseguradas pela máquina virtual Java durante a execução dos *bytecodes* de um programa [59, 108, 73].

A linguagem Java é projetada para garantir *type safety*, isto é, um programa só pode efetuar uma operação em um objeto se ela pertencer ao conjunto de operações definidas pelo tipo do mesmo. *Type safety* é um elemento essencial da segurança de Java, cuja importância fica clara ao notar-se que o modelo de segurança é composto por uma coleção de classes de determinados tipos. Sem esse mecanismo seria possível alterar dados da máquina virtual comprometendo completamente a segurança da arquitetura.

Um elemento importante para garantir *type safety* é a restrição imposta ao uso de adequação de tipos (*type casting*). Assim, em Java, uma referência a um objeto do tipo T_1 só pode ser manipulada como T_1, não sendo possível adequar a referência a um tipo T_2 arbitrário. Há somente dois casos em que uma referência pode ser explicitamente adequada a um tipo diferente. Vamos inicialmente supor a existência das classes `Automovel` e `Gol`, sendo a última uma especialização da primeira. Com base nesta hierarquia simples, no trecho de programa a seguir, mostramos as adequações válidas:

```
Automovel a = new Automovel();
Gol g = new Gol();

g = (Gol)a;       // supertipo para subtipo;
                  // inválida, pois a referência é para Automovel
                  // esta atribuição gera uma exceção
a = (Automovel)g; // subtipo para supertipo; válida, mas desnecessária
g = (Gol)a;       // agora esta adequação é válida, pois a referência é
                  // para uma instância da classe Gol
```

A inexistência de ponteiros em Java em conjunto com *type safety* são características que protegem a memória contra acessos arbitrários. Essa proteção é complementada por outros mecanismos como:

- garantia da semântica de `final` – classes e membros declarados dessa maneira não podem ser alterados ou redefinidos. Quase todas as verificações de segurança em Java são realizadas por métodos `final`. Se Java não impedisse que tais métodos fossem sobrescritos em uma subclasse, os mecanismos de segurança poderiam ser comprometidos;

- variáveis não podem ser usadas antes de inicializadas – se o contrário fosse possível poder-se-ia declarar uma grande quantidade de variáveis não inicializadas para ler posições de memória aleatórias em procura de alguma informação interessante. Para prevenir esse problema Java obriga que as variáveis locais sejam inicializadas antes de utilizadas e atribui um valor padrão durante a criação de cada variável de instância e de classe;

- coleta de lixo automática – Java se encarrega exclusivamente de liberar toda a memória alocada a objetos não mais referenciados. Desse modo, evita-se desperdiçar memória com objetos não utilizados e que se tente liberar a memória alocada a um objeto múltiplas vezes, causando o término inesperado do programa. Outro problema evitado é o da remoção de um objeto ainda referenciado por outro(s) objeto(s), que cria assim referência(s) pendente(s) (*dangling*). Este tipo de referência compromete *type safety* uma vez que a memória referenciada, agora considerada livre, pode ser alocada a um novo objeto completamente diferente, que pode ser tratado como o original;

- verificação de limites de vetores – toda vez que um vetor é acessado ocorre uma verificação para determinar se o item indexado está dentro dos limites do vetor. Dessa forma, não é possível armazenar um valor, por exemplo, na nona posição de um vetor com sete elementos como em C++.

4.3 Verificador de arquivos `class`

O verificador de arquivos `class` [56, 59, 108] faz parte da máquina virtual Java e não pode ser acessado pelos programas. Este módulo é acionado automaticamente e sua função é verificar que as classes carregadas possuem uma estrutura interna correta e que elas são consistentes umas com as outras. Muitas das verificações ocorrem antes que os *bytecodes* da classe sejam executados. Se for encontrado algum problema com a classe, uma exceção é gerada. Embora um compilador Java normal não deva gerar classes mal formadas, um *cracker* pode criar uma seqüência de *bytecodes* inválidos com o intuito de violar as regras da máquina virtual.

O processo de verificação é realizado em quatro passos:

1. Verificações estruturais no arquivo `class` – neste primeiro passo, verifica-se se o formato do arquivo de classe está correto. O arquivo deve começar com o número mágico `0xCAFEBABE` e deve possuir tamanho consistente com seu conteúdo.

2. Verificações semânticas – nesta etapa, são feitas verificações que não dependem dos *bytecodes* da classe. O verificador garante que toda classe possui uma superclasse

(com exceção da classe `Object`), que classes `final` não são estendidas e que métodos `final` não são sobrescritos. Métodos e atributos referenciados de dentro da classe devem possuir nomes, classes e tipos válidos.

3. Verificação de *bytecodes* – neste passo são verificados os *bytecodes* dos métodos da classe. Entre outras coisas, deve-se garantir que variáveis locais sejam usadas somente após serem inicializadas, que métodos sejam invocados com argumentos válidos, que atributos recebam apenas valores do tipo apropriado e que os *opcodes* sejam válidos e possuam operandos válidos.

4. Verificação de referências simbólicas – as três primeiras fases da verificação ocorrem durante o processo de carga e ligação da classe. Já este quarto passo é realizado em tempo de execução durante a ligação dinâmica e garante que referências simbólicas para classes, atributos e métodos estão corretas e referenciam itens válidos. Verifica-se também se o método em execução possui acesso ao item referenciado.

4.4 Carregador de classes

Este componente [24, 55, 56, 108] é responsável pela carga dinâmica de classes na máquina virtual Java. Classes são carregadas sob demanda a partir de representações binárias independentes de plataforma conhecidas como formato de arquivo `class`. A representação de uma única classe é denominada arquivo `class`, embora não seja necessário que ela esteja armazenada em um arquivo; pode-se, por exemplo, gerá-la em tempo de execução. Carregadores de classes podem ser definidos pelos programadores para atender às necessidades da aplicação. Isto é feito estendendo-se a classe abstrata `java.lang.ClassLoader` ou uma de suas subclasses, como `java.security.SecureClassLoader`. Por fim, cada carregador define um espaço de nomes separado, que é um conjunto de nomes únicos, um para cada classe por ele carregada.

Um carregador de classes `L` define, para cada classe `C` que carregar, um tipo qualificado por `<C,L>`. Neste caso, diz-se que `L` define `C` ou que `L` é o definidor de `C`. Dois tipos em Java são iguais se os nomes completamente qualificados das classes forem iguais e estas forem definidas pelo mesmo carregador de classes. Uma mesma classe `X` pode ser carregada em espaços de nomes diferentes (definindo, portanto, tipos diferentes). Porém, uma vez que a classe `X` seja carregada no espaço `Y`, nenhuma outra classe de mesmo nome poderá coexistir em `Y` ou substituir `X` em `Y`. Isto é crucial para garantir *type safety* [55, 24]. A Figura 4.4 mostra dois espaços de nomes distintos gerados pelos carregadores `L1` e `L2`. A classe `Recurso` está carregada nos dois espaços e resulta, então, em dois tipos diferentes.

A contribuição da separação de espaços de nomes para segurança reside no fato de que classes em espaços de nomes distintos não conseguem enxergar a presença umas das

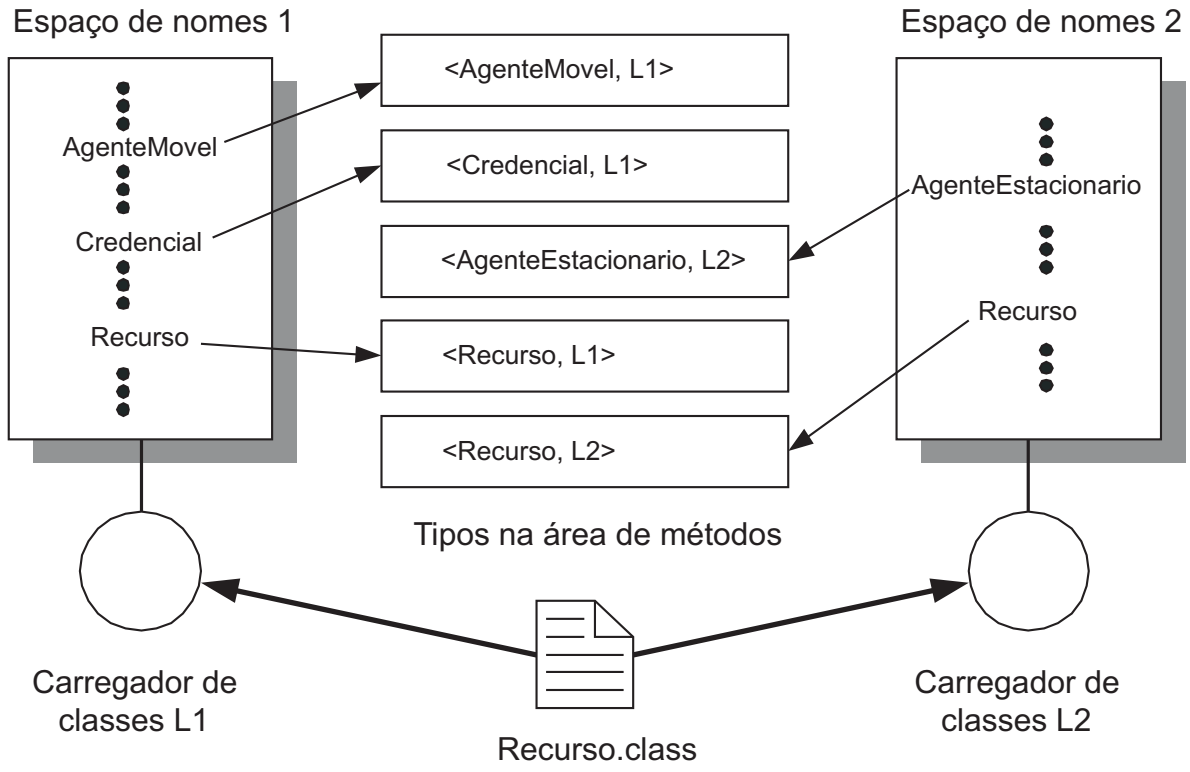

Figura 4.4: Separação de espaços de nomes pelos carregadores de classes.

outras, salvo se for provido um mecanismo explícito para esta finalidade. Com isso, para prevenir que códigos maliciosos interfiram na execução de códigos confiáveis, basta utilizar carregadores de classes distintos para carregar o código de cada aplicação.

Em Java 2, os carregadores de classes passaram a ser organizados de forma hierárquica em um modelo de delegação. No topo da hierarquia está o carregador de classes *bootstrap* [56, 108], também chamado de carregador de classes primordial, nulo ou do sistema [24, 55, 73] e que é responsável por carregar as classes do núcleo da API Java. Excetuando-se o *bootstrap*, todo carregador de classes possui um carregador pai na hierarquia de delegação. Quando uma classe `C` definida por um carregador `L` necessitar de uma classe `D` ainda não carregada, a máquina virtual solicita que `L` (o definidor de `C`) efetue essa tarefa . Essa solicitação é repassada nível a nível pela hierarquia até o carregador *bootstrap* e a classe é fornecida pelo nível mais alto que souber carregá-la.

Para exemplificar como funciona a delegação, vamos considerar a hierarquia ilustrada na Figura 4.5. No topo está o *bootstrap* ($\mathtt{L_B}$) que é pai do carregador de classes do *classpath* ($\mathtt{L_C}$). Este por sua vez é pai do carregador de classes da aplicação ($\mathtt{L_A}$). O carregador *bootstrap* está desenhado com linhas tracejadas porque ele é parte da máquina virtual Java e não uma instância de uma classe que estende `ClassLoader`, como os demais

carregadores.

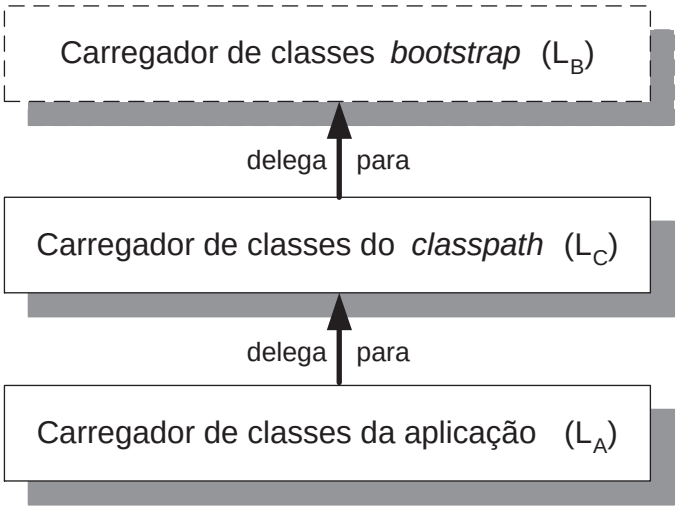

Figura 4.5: Exemplo de hierarquia de delegação de carregadores de classes.

Imagine que a aplicação necessite da classe `AgenteMovel` pertencente a ela. Primeiramente, a máquina virtual verifica se a classe já não está carregada. Se não estiver, aquela solicita a classe ao carregador L_A. Este delega a tarefa a L_C que repassa a L_B. Como a classe não pertence ao núcleo da API Java e nem está no *classpath*, L_B e L_C retornam sem carregar a classe `AgenteMovel`, o que é finalmente realizado por L_A. Neste caso, L_A iniciou a carga da classe solicitada e a definiu: por isso, L_A é chamado de carregador iniciador e definidor de `AgenteMovel`.

Agora suponha que a classe `AgenteMovel` utilize a classe `java.lang.String` e que esta é a primeira referência à classe no programa. A máquina virtual solicita que ela seja carregada por L_A. A delegação ocorre como no exemplo anterior e agora, como a classe faz parte do núcleo da API Java, ela é carregada por L_B. Como no exemplo anterior, L_A iniciou a carga da classe requisitada, mas desta vez ela foi definida por L_B. Então, neste cenário, dizemos que L_A é o iniciador de `String` e L_B seu definidor.

Com este modelo de delegação, impede-se que uma aplicação maliciosa tente carregar uma classe falsa substituindo uma classe do núcleo da API Java, por exemplo. Como uma classe é sempre fornecida pelo nível mais alto na hierarquia que souber carregá-la, classes da API Java serão sempre carregadas pelo *bootstrap*. Também não será possível criar uma classe como `java.lang.AgenteMalicioso` para tentar fazer uso da visibilidade de pacote para obter acesso privilegiado às classes de `java.lang`. Como esta classe não será carregada pelo *bootstrap*, ela pertencerá a um *runtime package* diferente do de `java.lang`, o que impede a referida visibilidade. Em Java 2, a partir da versão 1.3, é impossível definir uma classe como pertencente ao pacote `java`, o que acarretaria uma exceção de segurança

em tempo de execução.

Por fim, a partir da versão 1.2 de Java, cada classe carregada é associada a um domínio de proteção, tema da Seção 4.7

4.5 Gerenciador de segurança

O gerenciador de segurança [108, 73] determina quando operações potencialmente perigosas, como a escrita no sistema de arquivos, podem ou não ser realizadas. Enquanto os componentes da arquitetura de segurança descritos nas seções anteriores protegem a integridade da máquina virtual Java, o gerenciador de segurança, além disso, define as bordas externas da *sandbox*, protegendo os recursos externos à maquina virtual.

O gerenciador de segurança é composto por diversos métodos de verificação como, por exemplo, `checkRead()`, que é invocado para determinar se o arquivo especificado pode ou não ser lido. Uma lista completa detalhando estes métodos pode ser encontrada em [73]. Quando um programa Java chama um método potencialmente perigoso na API Java, esta consulta o gerenciador de segurança, invocando o método de verificação apropriado, para saber se a operação é permitida. Em caso negativo, o gerenciador gera uma exceção de segurança que se propaga pela cadeia de métodos do *thread* que efetuou a operação proibida. Quando a exceção é recebida pelo primeiro método da cadeia, o *thread* é finalizado. De outro modo, a operação é realizada normalmente.

Nas versões 1.0 e 1.1, a única maneira de se estabelecer uma política de segurança era estender a classe abstrata `java.lang.SecurityManager` e implementar os métodos de verificação para refletir a política desejada. Isso é uma tarefa complicada que muitas vezes resultava em brechas na segurança. Para minimizar estes problemas, a plataforma Java 2 passou a fornecer uma implementação padrão para a classe `SecurityManager`. A política de segurança agora é definida em um arquivo de política em formato ASCII, sem a necessidade de programação. Quando um método de verificação do gerenciador de segurança é invocado, ele repassa a requisição ao controlador de acesso (Seção 4.8), que é, em Java 2, a entidade que realmente garante a execução da política de segurança.

4.6 Permissões e política de segurança

As classes de permissões [23, 108, 73] introduzidas na plataforma Java 2 representam acessos a recursos do sistema. Para que uma aplicação obtenha acesso a um determinado recurso é necessário que a ela seja concedida explicitamente a permissão correspondente.

Toda permissão em Java é subclasse da classe abstrata `java.security.Permission`. Novas permissões podem ser criadas estendendo-se esta classe ou uma de suas subclasses.

Tipicamente, uma permissão consiste de um alvo e uma ação, que variam de acordo com o tipo de permissão. A Tabela 4.1 mostra alguns exemplos de permissões com os respectivos alvos e ações.

Exemplo de permissão	Alvo	Ação
`p = new java.security.AllPermission()`	-	-
`p = new java.lang.RuntimePermission("stopThread");`	`stopThread`	-
`p = new java.io.FilePermission("/tmp/a.txt", "read");`	`/tmp/a.txt`	`read`

Tabela 4.1: Exemplos de permissões, alvos e ações.

Cada permissão deve implementar o método abstrato `implies()` da classe `Permission`. O código `x.implies(y)` deve retornar `true` se ter a permissão `x` implica ter a permissão `y`. Por exemplo, uma permissão `x` para ler todos os arquivos do diretório `/home/nelson` implica uma permissão `y` para ler o arquivo `/home/nelson/.myprofile`.

A política de segurança, em Java 2, consiste de permissões concedidas a *code sources*. Um *code source* corresponde ao local de onde uma classe foi carregada juntamente com a relação daqueles (eventualmente ninguém) que a assinaram digitalmente. Essas concessões são realizadas por meio de cláusulas `grant` enumeradas em arquivos de política Java, sendo um deles global e o restante específico a cada usuário. Quando a máquina virtual Java é iniciada, as cláusulas que formarão a política de segurança são lidas a partir dos arquivos específicos do usuário e do arquivo de política global. A política é representada, em tempo de execução, por um objeto `Policy`.

Permissões adicionais àquelas especificadas nos arquivos de política podem ser concedidas pelas aplicações às classes que elas carregarem. Além disso, as implementações padrões de carregadores de classe providas em Java dão permissões adicionais a toda classe carregada. Por exemplo, classes que são carregadas pelo protocolo HTTP podem estabelecer uma conexão de rede com o servidor origem; classes oriundas do sistema de arquivos local podem ler arquivos da estrutura de diretórios cuja raiz é o diretório do qual as classes foram carregadas.

A seguir, damos um exemplo de arquivo de política:

```
grant codeBase "http://siteamigo.com.br" {
   permission java.io.FilePermission "segredo.txt", "read";
   permission java.io.FilePermission "msg.txt", "read";
};

grant signedBy "amigo" {
   permission java.io.FilePermission "/home/artigos/*", "read";
};
```

A primeira cláusula concede às classes oriundas de `siteamigo`, assinadas ou não, permissão para ler os arquivos `segredo.txt` e `msg.txt`. Já a segunda cláusula permite que as classes assinadas por `amigo` possam ler qualquer arquivo localizado em `/home/artigos`. Se uma classe satisfizer mais de uma cláusula, ela receberá todas as permissões concedidas por cada uma delas. Assim, no exemplo anterior, uma classe proveniente de `siteamigo` e assinada por `amigo` recebe todas as três permissões acima listadas.

4.7 Domínios de proteção

Um domínio de proteção [23, 26, 108] define todas as permissões concedidas a um *code source* e conseqüentemente, a um conjunto de classes que devem ser tratadas com os mesmos privilégios. Um domínio é estabelecido através de uma ou mais cláusulas `grant` do arquivo de política. Considerando o exemplo da seção anterior, temos que classes de `siteamigo` assinadas por `amigo` pertencem todas a um mesmo domínio que possui as três permissões do exemplo.

Cada tipo definido na máquina virtual Java é associado, durante a carga da classe, a um único domínio de proteção, de acordo com seu *code source*. O carregador de classes obtém as permissões de um domínio chamando o método `getPermissions()` do objeto `Policy`, que retorna as permissões relacionadas ao *code source* com base nos arquivos de política. O carregador pode modificar o domínio removendo permissões ou adicionando outras mais. O mapeamento de classes para domínios e permissões é ilustrado na Figura 4.6. Como vemos, permissões não são concedidas diretamente a classes e sim, a domínios de proteção.

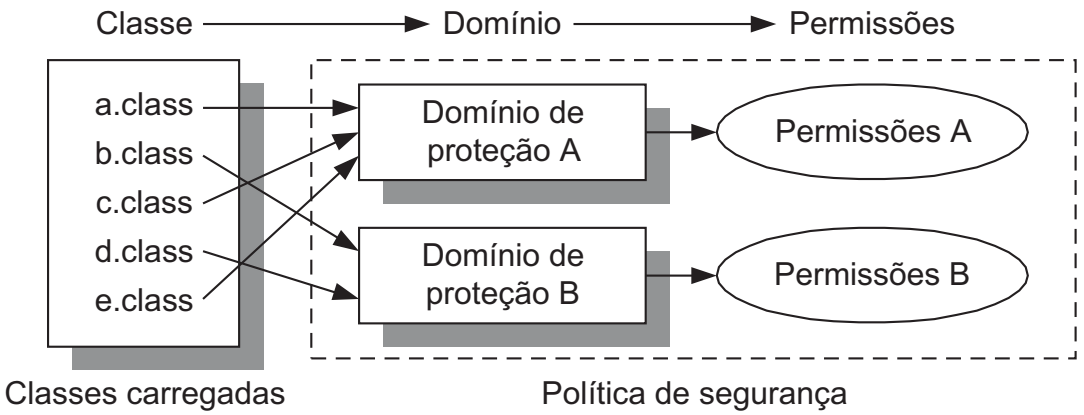

Figura 4.6: Domínios de proteção.

4.8 Controlador de acesso

O controlador de acesso [23, 108, 73] foi introduzido na plataforma Java 2 e é responsável por garantir a política de segurança segundo a nova arquitetura. O controlador decide quando um acesso a um recurso do sistema deve ser permitido ou não, tal qual o gerenciador de segurança. Apesar disso, ele não substitui este último para permitir que aplicações antigas, baseadas no modelo de segurança anterior, possam ser executadas sem modificações. Assim, a API Java continua chamando os métodos de verificação do gerenciador de segurança quando uma operação perigosa é realizada. O que muda é que estes métodos agora chamam o novo método `checkPermission()` do gerenciador que chama, por sua vez, o método de mesmo nome da classe `AccessController`.

A Figura 4.7 [73] ilustra a coordenação, acima descrita, entre a API Java, o gerenciador de segurança e o controlador de acesso na proteção de recursos. Percebe-se pela figura, que o controlador de acesso apenas suplementa o gerenciador de segurança e não o substitui. Além disso, vemos que bibliotecas nativas não passam pelo controle do modelo de segurança e que as aplicações podem acessar diretamente o controlador de acesso.

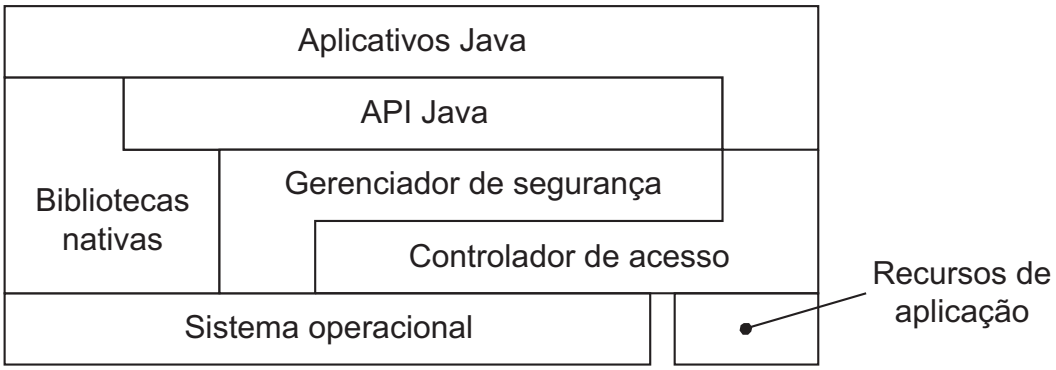

Figura 4.7: Relação entre o gerenciador de segurança e o controlador de acesso.

Para decidir se uma operação O pode ser realizada por um método `m()`, o controlador de acesso verifica se todos os métodos contidos na pilha de execução do *thread* corrente possuem permissão para realizar O. A verificação ocorre do topo para a base da pilha e uma exceção é gerada se um método não possuir os devidos privilégios. Cada método presente na pilha está associado a um domínio de proteção através da classe à qual pertence. Assim, para saber se um método $\texttt{m}_\texttt{i}\texttt{()}$ na pilha pode realizar O, basta checar se as permissões do domínio de $\texttt{m}_\texttt{i}\texttt{()}$ implicam as permissões necessárias. O processo descrito é chamado de inspeção de pilha e impede que um método de um domínio com menos privilégios obtenha permissões adicionais chamando um método de um domínio mais privilegiado. Há uma exceção a essa regra que ocorre quando o método `doPrivileged()` (Apêndice B.1) é utilizado. O Apêndice B.1 contém exemplos de inspeção de pilha.

4.9 Fraquezas do modelo e antigas vulnerabilidades

O modelo de segurança de Java, embora se preocupe com diversos problemas de segurança, ainda possui algumas fraquezas que podem ser exploradas por código malicioso. Essas fraquezas estão relacionadas ao uso indiscriminado de recursos do sistema como memória e CPU. Assim, é possível efetuar ataques de negação de serviço exaurindo-se toda a memória disponível ou criando-se inúmeros *threads* até que o sistema fique extremamente lento. A dificuldade em barrar esses tipos de ataques reside na questão de como diferenciar, por exemplo, alocações feitas por uma aplicação de edição de vídeo que requeira uma grande quantidade de memória das alocações realizadas por uma aplicação puramente maliciosa.

A Classe 4.1 implementa os ataques acima mencionados. Cada *thread* criado aloca 50000 *bytes* de memória, cria um novo *thread* e executa alguns cálculos matemáticos em um laço sem fim. Com isso, em pouco tempo, a máquina fica sem memória e muito lenta. O exemplo pode ser facilmente adaptado para atacar sistemas de agentes móveis, como mostra a Classe C.1 no Apêndice C.

Outros pontos que não são controlados pelo modelo de segurança incluem a criação de janelas e a finalização de *threads* [59]. Aproveitando-se disso, uma aplicação maliciosa pode abrir milhares de janelas que cubram a tela toda do computador. Com isso, além de se atrapalhar a visualização da área de trabalho, congestiona-se a fila de eventos do gerenciador de janelas o qual, normalmente, pára de responder. Quanto à finalização de *threads*, em [59] temos uma classe simples que recria o *thread*, usando um bloco `try/finally`, toda vez que o método `Thread.stop()`[2] é chamado. Adaptamos a idéia e criamos um agente que não pode ser finalizado pelos métodos da API (Classe C.2).

Classe 4.1: DoS.java - ataque de negação de serviço

```
import java.lang.Math;

class DoS extends Thread {
  Thread t;
  Byte[] ba = new Byte[50000];    // Aloca 50000 Bytes

  public void run() {
    t = new DoS();    // Cria novo thread
    t.start();

    // Executa indefinidamente uma tarefa
```

[2] O uso deste método está desaprovado (*deprecated*) atualmente.

```
    while (true)
      Math.acos(Math.atan(Math.asin(Math.random())));
  }

  public static void main(String[] args) {
    Thread t = new DoS();    // Cria thread inicial
    t.start();
  }
}
```

Para finalizar esta seção, vamos relatar dois ataques explorando vulnerabilidades descobertas em versões mais antigas de Java. O primeiro ataque, conhecido como **"Pulando o firewall"** [59], permitia que um *applet* proveniente da Internet fizesse conexões para máquinas dentro de uma rede privada protegida por um *firewall*. Já o segundo ataque explorava uma situação não prevista no uso dos carregadores de classes que possibilitava criar um cenário de **confusão de tipos**, no qual um objeto de um tipo qualquer pode ser manipulado como sendo de outro tipo [89, 55].

Uma cronologia completa dos *bugs* em Java relacionados à segurança pode ser encontrada em [66]. Em [59], alguns desses *bugs* e os ataques relacionados são apresentados de forma mais detalhada.

Pulando o firewall

Este ataque foi descrito por Drew Dean, Ed Felten e Dan Wallach da Universidade Princeton em 1996 [59] e afetava versões anteriores à 2.01 do Netscape Navigator e à 1.0.1 do JDK. O ataque explorava o modo como o gerenciador de segurança de *applets* utilizava DNS para a resolução de nomes, como veremos a seguir.

Uma restrição imposta a todo *applet* é que eles não podem estabelecer conexões de rede, exceto para o servidor Web de onde se originaram. Nas versões afetadas com o problema, quando um *applet* tentava se conectar a alguma máquina M, os seguintes passos eram realizados para determinar se M correspondia ao servidor origem:

- Por meio de uma consulta DNS, o nome do servidor de origem era traduzido para uma lista de endereços IP.
- Outra consulta DNS traduzia o nome da máquina destino M para uma segunda lista de endereçoes IP.
- Se a intersecção entre as duas listas fosse não nula, as máquinas eram declaradas iguais e era aberta uma conexão para o primeiro endereço IP da lista correspondente a M. De outro modo, a conexão não era permitida.

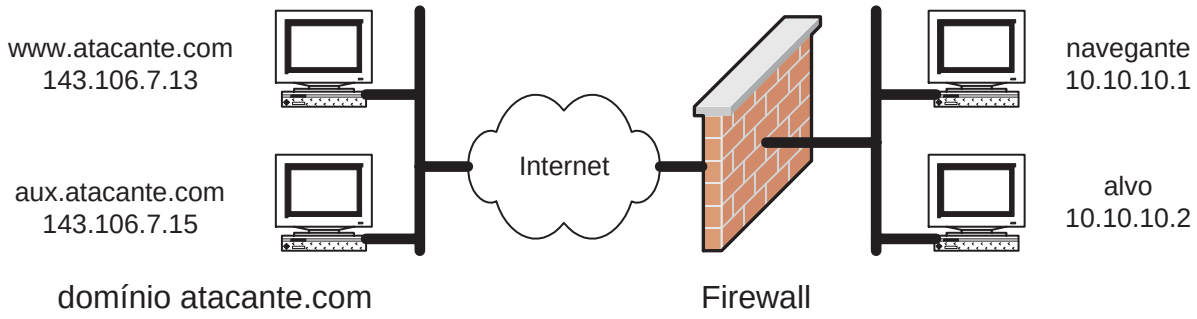

Figura 4.8: Exemplo para o ataque "Pulando o firewall".

Para explicarmos como o ataque podia ser realizado, vamos tomar como base a Figura 4.8, que ilustra uma rede privada protegida por um *firewall* e um domínio pertencente ao atacante. O servidor Web chamado `www.atacante.com` (143.106.7.13) contém em suas páginas um *applet* malicioso escrito para atacar a máquina `alvo` (10.10.10.2).

Eventualmente alguém usando a máquina `navegante`, ou outra máquina da sub-rede de `alvo`, pode acessar o *site* `www.atacante.com`. Com isso, o *applet* malicioso é trazido para dentre da rede privada, quando então solicita uma conexão de rede para `aux.atacante.com`. O servidor DNS do domínio `atacante.com` é consultado sobre o endereço IP da máquina em questão. Maliciosamente, o servidor DNS devolve o par de endereços (10.10.10.2, 143.106.7.13). Como o último endereço é o mesmo do servidor origem do *applet*, o gerenciador de segurança permite que uma conexão seja estabelecida para o primeiro endereço da lista (10.10.10.2) que é o IP da máquina `alvo`.

A solução para o problema foi simplesmente mudar o critério para permissão de conexões. Agora, o endereço IP de cada servidor origem é armazenado e um *applet* pode estabelecer conexões de rede somente para o IP de seu servidor.

Confusão de tipos

Este ataque foi descrito por Saraswat [89] em 1997 e podia ser executado nas versões 1.0 e 1.1 do JDK. A vulnerabilidade explorada residia no fato de que as implementações dos compiladores e da máquina virtual Java consideravam apenas o nome da classe para o conceito de tipo ao invés do par (classe, carregador) definido na especificação da máquina virtual. Graças a isso, dependendo de como eram organizados os carregadores de classes, era possível comprometer *type-safety* de Java. Felizmente, como os *applets* eram proibidos de criar carregadores de classes, eles não conseguiam realizar ataques explorando o problema.

Inicialmente, para mostrar o ataque, vamos listar algumas classes:

```
public class Alvo {
```

```
   private int segredo = 20;
}

public class Ponte {
   public Alvo obtemAlvo() {
      return new Alvo();
   }
}

public class Coordenador {
   public static void main(String[] args) {
      Ponte ponte  = new Ponte();
      Alvo alvo    = ponte.obtemAlvo();
      System.out.println(alvo.segredo);
   }
}
```

A classe a seguir será chamada, no restante do texto, de `Alvo` falso. Note que o nome completamente qualificado desta classe é igual ao nome da classe `Alvo` original.

```
public class Alvo {
   public int segredo;
}
```

Dadas as classes acima, vamos supor a existência de dois carregadores de classes `L1` e `L2`. O carregador `L1` é responsável pela carga de `Coordenador` e `Alvo` falso e delega a carga de `Ponte` a `L2`. Este por sua vez, responsabiliza-se pela carga das classes `Ponte` e de `Alvo`.

O ataque procede da seguinte maneira: a classe `Coordenador` é definida por `L1`. Em decorrência disso, `L1` é utilizado para iniciar a carga de `Ponte` e `Alvo` referenciados no método `main()`. Enquanto `Ponte` é definida por `L2`, por meio da delegação, `L1` define `Alvo` a partir de `Alvo` falso. Quando o método `obtemAlvo()` é chamado, `L2` inicia o processo de carga de `Alvo`, referenciado pelo método, e que ainda não foi definido no espaço de nomes gerado por `L2`. Este definirá a classe a partir da classe `Alvo` com o atributo `private` e será esse o tipo do objeto retornado pelo método `obtemAlvo()`. Porém, a variável `alvo` que referenciará o objeto retornado é do tipo `Alvo` falso. Isso causa uma inconsistência entre os espaços de nomes `L1` e `L2` que permite, no caso, que um atributo privado seja acessado como se fosse público. Este problema permaneceu um bom tempo sem solução, a qual foi proposta finalmente por Liang e Bracha [55] em 1998 e implementada nas versões mais recentes da plataforma Java.

4.10 Arquitetura de segurança de Java e segurança de sistemas de agentes móveis

Nesta seção, exemplificamos como alguns mecanismos da arquitetura de segurança de Java podem ser usados para solucionar problemas da segurança de sistemas de agentes móveis baseados nesta linguagem.

Um dos componentes mais importantes é o carregador de classes, que é responsável por obter as classes dos agentes recebidos por um servidor. Se é desejável que o código do agente trafegue apenas por canais seguros, o carregador deve ser programado para esse fim, utilizando algum mecanismo como SSL, por exemplo. Outra função do carregador é atribuir as permissões de acesso a um agente, associando-o a um domínio de proteção específico, com base em informações como o *codebase* e o proprietário do agente. Para proteger agentes contra outros agentes residentes no mesmo servidor, podemos carregar cada agente móvel, utilizando um carregador de classes separado. Dessa forma, um agente não poderá interagir diretamente com outro agente, o que deverá ocorrer por meio da infra-estrutura de comunicação dos servidores e de maneira protegida. Um ponto importante é que não pode ser concedida permissão para que agentes móveis criem um carregador de classes; senão, todo o esquema de segurança fica comprometido.

O gerenciador de segurança e o controlador de acesso funcionam como um monitor de referências e garantem a execução de uma política de segurança determinada pelo sistema. A política pode ser facilmente adaptada às necessidades específicas de cada sistema, por meio da criação de permissões, que estendam a classe `Permission`, para controlar funcionalidades como migração de agentes e comunicação, por exemplo.

Capítulo 5

Um *Survey* sobre Sistemas de Agentes Móveis

Durante os últimos anos, diversos sistemas de agentes móveis foram desenvolvidos por universidades e empresas. Muitos destes sistemas apresentam mecanismos de segurança para proteção do servidor contra ataques perpetrados por agentes móveis e outras entidades externas. Porém, o problema inverso, o da proteção do agente móvel contra servidores maliciosos, é tratado por apenas alguns dos sistemas e de forma bem superficial.

Este capítulo é uma extensão de um artigo publicado nos anais do 3º Simpósio Segurança em Informática [106] e de um relatório técnico [105], nos quais descrevemos e analisamos alguns sistemas de agentes móveis, com ênfase nos aspectos de segurança. Cada análise é dividida em quatro partes: (1) **informações gerais**, como organização desenvolvedora, linguagens suportadas e ano de criação; (2) **descrição do sistema**, que apresenta os conceitos do sistema analisado; (3) **aspectos de segurança**, que cobre os mecanismos de segurança presentes no sistema; e (4) **comentários** sobre os mecanismos de segurança utilizados, apontando eventuais fraquezas e problemas existentes.

O restante deste capítulo está estruturado da seguinte forma: da Seção 5.1 à Seção 5.8, analisamos de forma completa os seguintes sistemas de agentes móveis: Telescript, Aglets, Concordia, Ara, D'Agents, SOMA, Ajanta e Grasshopper. A Seção 5.9 apresenta uma rápida descrição de alguns outros sistemas. Finalmente, a Seção 5.10 resume os principais aspectos dos sistemas analisados, organizando-os em quadros sinóticos.

5.1 Telescript

Informações Gerais

Telescript [100, 114] é uma linguagem orientada a objetos para desenvolvimento de aplicações baseadas em agentes. Foi criada pela empresa General Magic, em 1995,

com objetivos comerciais. Como a programação de agentes em Telescript requeria o aprendizado de uma linguagem completamente nova, o projeto não obteve o sucesso esperado e foi abandonado. Abordamos Telescript neste *survey* apenas por razões históricas, uma vez que foi um dos primeiros projetos a abordar os problemas de segurança oriundos desta área.

Descrição do Sistema

A ***engine*** Telescript é um programa que implementa a linguagem Telescript e que executa os servidores de agentes, chamados de ***places***. Um agente estacionário é responsável por representar um *place* e prover os serviços oferecidos por este servidor. Todo agente que chega a um servidor pode verificar o conjunto de serviços oferecidos, interagindo com o agente representante. Um conjunto de *places* operados por uma mesma **autoridade** recebe o nome de **região**.

Os nomes das entidades em Telescript são dependentes de localização e baseados em nomes de domínio DNS. O modelo de mobilidade é forte permitindo a captura do estado de execução no nível de *thread*. Quando um agente migra para outro servidor, todas as classes relacionadas são transportadas para o novo local. Uma forma de causar a migração de um agente é por meio da primitiva `go`, que faz com que a próxima instrução do agente seja executada no *place* destino. Telescript possui também uma migração relativa baseada na primitiva `meet`. Com ela, um agente pode migrar para um servidor que esteja hospedando um determinado agente com o qual se queira interagir.

Aspectos de Segurança

Telescript é uma linguagem "segura"do ponto de vista de programação. Não possui ponteiros, como em Java, e qualquer interação com objetos é feita por meio da interface pública dos mesmos. O interpretador realiza verificação de tipos em tempo de execução, gerenciamento de memória e processamento de exceções. Por fim, o interpretador funciona como um monitor de referências mediando os acessos aos objetos.

Cada processo está associado a uma **autoridade** que é uma identificação única atribuída a um responsável (uma pessoa ou organização). Essa autoridade pode ser autenticada quando necessário, por exemplo, para permitir que um servidor decida se aceita ou não um agente migrante.

Utilizando-se **permissões**, pode-se limitar o consumo de recursos e restringir a execução de um agente. As permissões podem ser atribuídas pelo criador do agente, pela *engine* Telescript no momento de criação do processo ou pelo servidor ao aceitar um agente. A intersecção de todas estas permissões resulta nas permissões finais do

agente. Quando um processo viola um limite estabelecido, uma exceção é sinalizada que, geralmente, causa a finalização do código responsável.

A proteção de agentes durante uma migração é realizada por meio de canais seguros estabelecidos segundo um **regime de segurança**. Há seis regimes de segurança em Telescript e cada um deles especifica um conjunto de serviços e protocolos criptográficos a ser utilizado na criação do canal. O regime mais simples somente troca as informações das autoridades envolvidas, ao passo que os mais complexos utilizam autenticação por RSA, troca de chaves pelo protocolo Diffie-Helman e ciframento, para criar o canal seguro, por meio do algoritmo RC4. O uso de RSA requer uma infra-estrutura de chaves públicas que é provida pela **autoridade de nomes**. Seu funcionamento, porém, não é abordado em [100]

Comentários

Como Telescript pretendia ser utilizado globalmente, os algoritmos criptográficos foram obrigados a utilizar chaves menores, respeitando-se os limites impostos pela política de exportação dos Estados Unidos nesta área. Com isso, torna-se possível descobrir as chaves utilizadas nestes sistemas.

Todos esses mecanismos apresentados objetivam apenas a proteção do servidor e a proteção do agente contra outros agentes e terceiros. Telescript não possui qualquer recurso para a proteção do agente contra ataques realizados por um servidor.

5.2 Aglets

Informações Gerais

Aglets Software Development Kit (Aglets) [44, 51, 52, 53, 76], antigamente denominado de Aglets Workbench, é um sistema de agentes móveis desenvolvido pela IBM Tokyo Research Laboratory. O termo **aglet** é uma combinação das palavras *agent* e *applet*. Isso se deve ao fato de Aglets ter sido criado com base no modelo de *applets* de Java. O sistema é baseado na linguagem Java 1.1 e a primeira publicação data de 1996. Recentemente, a IBM disponibilizou o código fonte do sistema sob licença pública da IBM, a qual foi aprovada pela Open Source Initiative.

Descrição do Sistema

Os agentes neste sistema recebem o nome de **aglets** enquanto que os servidores de agentes são chamados de **Tahiti**. Um servidor contém um único **contexto** correspondente a um *place* na especificação MAF, embora a API suporte a existência de múltiplos contextos. O Tahiti possui uma interface gráfica para gerenciamento do contexto que permite criar e remover agentes, entre outras tarefas.

O modelo de programação de Aglets é orientado a eventos. Desse modo, para cada evento que possa ocorrer durante o ciclo de vida de um aglet, há um método correspondente que é ativado na ocorrência do mesmo. Por exemplo, quando um aglet é criado e quando chega a um contexto, os métodos `onCreation()` e `onArrival()` são executados, respectivamente. Todos os métodos tratadores de eventos podem ser modificados para atender às necessidades específicas dos agentes.

A mobilidade em Aglets é fraca e baseada na serialização de Java. Portanto, o estado de execução não é capturado no nível de *thread*. Quando o aglet deixa um contexto e é recriado na máquina destino, o método `run()` é chamado automaticamente. Uma migração ocorre quando um aglet invoca o método `dispatch()` ou quando alguma entidade solicita que o agente retorne a algum servidor específico. Aglets suporta transferência de código sob demanda e completa. Em alguns casos, uma combinação dos dois tipos é utilizada.

A comunicação entre agentes é realizada por meio da passagem de mensagens, de forma síncrona ou assíncrona. Cada aglet deve implementar um método para tratar as mensagens que recebe de outros agentes. O acesso a um agente não é feito por uma referência direta, mas sim por meio de um objeto ***proxy***. Um *proxy* representa um aglet específico e desempenha duas funções: proteger a interface pública do agente e prover transparência de localização. Quando um aglet migra de servidor, os *proxies* que o referenciavam deixam de ser válidos [76].

Aspectos de Segurança

Para descrever os aspectos de segurança de Aglets utilizamos o documento [76] e analisamos o código-fonte da versão 2.0.2. O modelo apresentado em [44] é apenas conceitual, não tendo sido implementado até a versão corrente do sistema.

Um **domínio**, em Aglets, corresponde a um conjunto de servidores que confiam uns nos outros. Todos os servidores pertencentes a um mesmo domínio compartilham uma chave secreta que é utilizada para a autenticação. Isto é realizado calculando-se um MAC sobre os dados a serem enviados concatenados com um *nonce*. Esta trinca de informações (dados, *nonce* e MAC) é enviada ao servidor destino que verifica o MAC para determinar se o servidor remetente pertence ao mesmo domínio. Uma desvantagem deste método é que não é possível saber exatamente a identidade do outro servidor. Todo agente recebido de um servidor no mesmo domínio é considerado confiável e aceito para execução. MACs também são utilizados para a verificação da integridade dos canais de comunicação.

A política de segurança utilizada no controle de acesso a recursos é especificada no arquivo `aglets.policy`, que contém **permissões** concedidas ao **proprietário** e/ou ***codebase*** do agente. O proprietário corresponde ao usuário que o agente representa

e o *codebase*, ao servidor que fornece as classes necessárias ao aglet. Ambas as informações são obtidas no objeto `AgletInfo` do agente. Além das permissões existentes na linguagem Java, Aglets fornece outras mais relacionadas à troca de mensagens e a funcionalidades dos contextos e agentes.

Comentários

Os mecanismos de segurança de Aglets são escassos e atendem a alguns requisitos apenas superficialmente. Muito pouco do modelo apresentado em [44] foi implementado e não há nenhuma preocupação quanto à segurança dos agentes contra servidores maliciosos.

A autenticação é realizada apenas para identificar se uma máquina pertence a um dado domínio. Assim, não é possível saber com que servidor a comunicação está sendo realizada. Se uma terceira parte mal intencionada obtiver a chave utilizada para cálculo do MAC, todo o esquema de autenticação fica comprometido. A solução também é muito pouco flexível e não poderia ser aplicada a redes abertas contendo diversos servidores como a Internet, por exemplo.

Os canais por onde trafegam mensagens e agentes não possuem mecanismos para garantir a confidencialidade da transmissão. A única proteção existente é em relação à integridade dos dados enviados.

Não existe controle de acesso sobre a primitiva `retract` e assim, qualquer agente pode solicitar que um agente localizado em um servidor remoto retorne/migre para o servidor no qual o primeiro é executado.

5.3 Concordia

Informações Gerais

Concordia [117, 48, 68, 113] é o sistema de agentes móveis desenvolvido pela Mitsubishi Electric ITA. É baseado na linguagem Java 1.1 e a primeira publicação sobre o sistema data de 1997.

Descrição do Sistema

Um sistema de agentes em Concordia é composto por uma série de servidores sendo executados sobre máquinas virtuais Java. Cada servidor Concordia possui uma série de componentes que são responsáveis por funcionalidades como mobilidade dos agentes, comunicação, administração, gerenciamento de segurança e persistência entre outros.

O componente **Service Bridge** permite a desenvolvedores de agentes/aplicações incluir serviços, em servidores Concordia, que possam ser utilizados pelos agen-

tes visitantes. Quando um agente deseja localizar um servidor ou um serviço ele consulta o **Directory Manager** que é responsável por manter um registro destas informações. Todo serviço que for disponibilizado a agentes visitantes deve ser registrado previamente por este módulo. No máximo, haverá uma unidade deste módulo por máquina. Os nomes de servidores e serviços são dependentes de localização e baseados em DNS.

Concordia suporta apenas mobilidade fraca valendo-se dos mecanismos de serialização de Java. A transferência de código pode ser feita sob demanda ou pelo envio de todas as classes necessárias ao agente de uma só vez. Para aumentar a confiabilidade na migração, Concordia utiliza uma fila de mensagens, que serve de *buffer* de transmissão, para armazenar os agentes migrantes. Assim, um agente pode permanecer na fila no caso do servidor destino estar indisponível no momento da transferência. Adicionalmente, uma cópia do agente é armazenada em mídia persistente e só é removida após o agente ter sido corretamente transferido para o servidor destino.

A comunicação entre agentes pode ocorrer por meio de **eventos** ou **colaboração**. No primeiro caso, os agentes podem definir quais tipos de eventos querem receber ou então podem juntar-se a um grupo e receber todos os eventos a este enviados. O tratamento dos eventos pode ser síncrono, a partir do *thread* principal do agente, ou assíncrono, utilizando-se um tratador de eventos. Já a colaboração permite que agentes formem grupos e correlacionem dados, em **pontos de colaboração**, obtidos individualmente por cada um deles.

O estado interno dos servidores Concordia bem como os estados dos agentes são armazenados em mídia persistente para possibilitar a recuperação do sistema em casos de queda. É permitido também a aplicações e agentes realizar *checkpoints* para, em caso de falhas, reiniciar sua execução a partir de um ponto desejado. Após uma falha do sistema ou do servidor, o módulo **Agent Manager** é responsável por reinicializar cada agente a partir da mídia persistente. Todo este aparato implica uma degradação do desempenho do sistema. Para garantir uma maior flexibilidade, Concordia permite que os administradores balanceiem desempenho e confiabilidade determinando quais módulos do servidor serão armazenados em mídia persistente.

Aspectos de Segurança

A proteção do agente, em Concordia, se preocupa com os problemas de segurança decorrentes da migração dos agentes e de seu armazenamento em disco. Concordia não implementa nenhuma funcionalidade adicional para proteger agentes enquanto estejam em memória, confiando esta tarefa, completamente, aos mecanismos de segurança de Java e do sistema operacional. Quanto à segurança do agente contra

servidores maliciosos, não há nada desenvolvido.

Concordia provê comunicação segura entre os servidores por meio do protocolo SSLv3 (Secure Sockets Layer version 3). Este protocolo fornece serviços de autenticação e ciframento para conexões TCP. Como Concordia realiza comunicação de rede utilizando RMI de Java, que é baseado em TCP/IP *sockets*, basta trocar as bibliotecas padrões de *sockets* por SSL, para se obter transmissão segura nos níveis superiores de rede, de forma transparente.

Um agente é armazenado em disco em decorrência da política adotada por Concordia para conseguir servidores mais confiáveis. Obviamente, isso representa uma brecha para ataques. Assim, para evitar o acesso não autorizado a um agente, as informações do mesmo são cifradas com o uso de um algoritmo simétrico, antes de serem armazenadas. Diversos algoritmos podem ser utilizados (IDEA, DES, RC4 entre outros) e a chave é gerada aleatoriamente para cada agente recebido pelo servidor. Esta chave é cifrada por um algoritmo assimétrico e armazenada junto com o agente.

A proteção de recursos dos servidores é baseada em extensões aplicadas ao modelo de segurança de Java. Este modelo é bastante restritivo e concede acesso somente a objetos classificados como confiáveis ou cujos códigos tenham sido escritos e digitalmente assinados por um autor considerado confiável. Em Concordia, a relação de confiança desejada não acontece com o autor e sim, com a pessoa pela qual o agente realiza alguma tarefa.

Todo agente em Concordia está associado a um usuário em particular por meio de uma **identidade de usuário**. Esta identidade corresponde a um objeto Java contendo um nome de usuário, um grupo de usuário e um valor *hash* calculado sobre uma senha (como em sistemas Unix). A identidade é carregada pelo agente durante todo o tempo e é verificada contra uma lista de usuários válidos em cada servidor visitado. A lista de usuários válidos é armazenada em um **arquivo de senhas** presente no disco local de cada servidor ou em um banco de dados JDBC centralizado. Este arquivo contém os nomes de usuários e os valores *hash* das senhas. Um *hash* é calculado sobre o arquivo e este é assinado digitalmente pelo servidor para evitar modificação não autorizada do arquivo. A validação de um agente se dá comparando o valor *hash* contido na identidade com o *hash* correspondente no arquivo de senhas.

O uso de recursos é condicionado por permissões concedidas de acordo com a identidade associada a um agente. Concordia possibilita que as permissões sejam dadas para aceitar ou negar acesso a recursos tão granulares como a leitura de um arquivo em particular. As permissões se aplicam a todos os recursos controlados pela classe

`SecurityManager` e algumas funcionalidades adicionais como criação e suspensão de agentes. Para isso, a classe `SecurityManager` é estendida por Concordia. Quando uma classe tenta utilizar um recurso, o sistema permite a operação se a classe for local ou a classe pertencer a um agente, sua identidade for validada e o mesmo possuir as permissões necessárias no **arquivo de permissões**. De outro modo, a classe é executada dentro de uma *sandbox*, que pode ser configurada para permitir o nível de acesso desejado.

Como já mencionado, as permissões sobre recursos são armazenadas em um arquivo de permissões. Este arquivo, igualmente ao arquivo de senhas, é protegido contra modificações indesejadas por meio de um valor *hash* assinado digitalmente pelo servidor.

Comentários

Concordia não possui nenhum mecanismo para proteger agentes contra servidores maliciosos. A proteção de agentes se preocupa somente com a migração e com a cópia que é armazenada em disco para aumentar a confiabilidade do sistema.

Em [113] comenta-se que a construção de um agente necessita da senha original, não sendo suficiente apenas o valor *hash* da mesma contido no objeto identidade. Não está claro como é este processo, mas nossa impressão é que a associação da identidade a um agente não é feita de forma segura. Considerando que a validação de um agente compara o valor *hash* presente na identidade com o *hash* pré-calculado contido no arquivo de senhas, conclui-se que o *hash* independe de qualquer código ou dado do agente. Embora seja difícil a uma terceira parte maliciosa obter o objeto identidade de um agente, uma vez que a transmissão de agentes entre servidores é feita por canais seguros e a cópia armazenada em mídia persistente é cifrada, um servidor malicioso tem acesso a essa informação. De posse deste dado, é possível forjar agentes e associá-los ao usuário relacionado, simplesmente adicionando-se o objeto identidade adquirido.

Para garantir a integridade dos arquivos de senha e de permissões poder-se-ia usar, no lugar do *hash* com a assinatura digital, um MAC, simplesmente.

5.4 Ara

Informações Gerais

O sistema de agentes móveis Ara (Agents for Remote Action) [80, 78, 79] é um trabalho desenvolvido na Universidade de Kaiserslautern e iniciado em 1997. A idéia principal do projeto é adicionar mobilidade às linguagens de programação

existentes. Ara suporta as linguagens Tcl, C e C++ e as plataformas Sparc Solaris, Intel Linux e Sparc SunOS. O código fonte é disponibilizado gratuitamente para uso não comercial.

Descrição do Sistema

Os servidores em Ara são chamados de ***places*** como em Telescript (Seção 5.1). A arquitetura do sistema é composta por um núcleo e por interpretadores para cada linguagem suportada. No núcleo, estão concentradas as funcionalidades independentes de linguagem, como mobilidade e comunicação. Já as questões dependentes de linguagem, como a captura e a restauração do estado de um agente, são resolvidas pelos interpretadores. Serviços de mais alto nível são disponibilizados por agentes estacionários.

Esta arquitetura torna o sistema flexível e permite adicionar suporte a novas linguagens. Adaptar um interpretador de uma dada linguagem para funcionar com o núcleo Ara requer duas tarefas: primeiro, é necessário criar interfaces (*stubs*) na linguagem em questão para que agentes que a utilizem possam efetuar chamadas às rotinas do núcleo; segundo, o interpretador deve fornecer rotinas ao núcleo (*upcalls*), que são chamadas em casos como a migração, por exemplo, em que a captura do estado de um agente é necessária.

O nome de um servidor é constituído de listas de URLs correspondentes aos protocolos de transporte que podem ser utilizados para se comunicar com o servidor. Já o nome de um agente consiste de um identificador único global, uma identificação do usuário responsável e um nome simbólico opcional.

Ara apresenta mobilidade forte em todas as linguagens atualmente suportadas. Quando um agente deseja migrar para um novo servidor, ele efetua uma chamada à primitiva `ara_go` do núcleo por meio da interface provida pela linguagem. Na migração, todas as classes necessárias ao agente são transferidas para o novo local.

A comunicação entre agentes é feita por troca de mensagens no estilo cliente-servidor. Ara encoraja o uso de comunicação local fornecendo, nos servidores, **pontos de encontro** onde esta interação entre agentes pode ocorrer. Comunicações globais não tem suporte do sistema e devem ser tratadas pelos próprios agentes utilizando-se os recursos de rede.

Por fim, Ara permite que os agentes criem um *checkpoint*, quando desejado, para armazenar seu estado interno atual em mídia persistente. Assim, se ocorrer algum problema, é possível restaurar o estado anterior do agente preservado pelo *checkpoint*.

Aspectos de Segurança

O modelo de segurança de Ara considera três entidades para serem autenticadas: **usuário do agente**, **fabricante do agente** e **máquina *host***. A primeira é o usuário do sistema que instancia o agente móvel e que é responsabilizado pelos atos do mesmo. Normalmente, os servidores se baseiam nesta identidade para conceder autorizações para a realização de operações. Por sua vez, o fabricante é a entidade que desenvolve o código do agente e a máquina *host* é uma máquina da rede que abriga um ou mais *places*.

Cada agente carrega consigo um **passaporte** que contém informações necessárias a sua autenticação. Entre estas informações estão um identificador, nome do agente e a assinatura digital do fabricante sobre o código. O passaporte nunca é alterado e é assinado digitalmente pelo usuário durante a criação do agente. Além desta parte imutável, há também atributos de segurança variáveis como o **rastro de servidores**. Este rastro é uma lista dos servidores visitados pelo agente e serve para que um servidor saiba por onde o agente sendo recebido andou previamente. A lista é assinada por cada servidor destino, após verificada a identidade do servidor remetente.

Durante a migração, o agente pode optar se deseja ou não realizar autenticação da origem e do destino bem como o ciframento dos dados transmitidos. Estes recursos são implementados com base no protocolo SSL e o agente deve balancear o desempenho e a segurança ao utilizá-los.

Uma **região** em Ara é composta por um conjunto de máquinas administradas por uma mesma autoridade e que portanto, pode ser considerada de forma atômica do ponto de vista de segurança. Dessa forma, quando um agente migra entre máquinas pertencentes a uma mesma região, os serviços de autenticação e ciframento são automaticamente desabilitados. Regiões são transparentes aos agentes e são conhecidas apenas pelo sistema de agentes móveis.

Cada *place* em Ara possui uma **função de admissão** que é responsável pelo processo de autorização. Estas funções recebem como parâmetros o passaporte do agente, atributos de segurança e a quantidade desejada de cada recurso. Com base nestas informações, aceita-se ou não a entrada do agente no servidor. No primeiro caso, o agente passa a ser executado e recebe um vetor contendo os limites no uso de cada recurso disponível. Os limites impostos são definidos com base nas quantidades solicitadas e na política de segurança adotada pelo servidor.

Comentários

Dos sistemas analisados, Ara foi o primeiro a oferecer alguma proteção ao agente contra servidores maliciosos. Como visto anteriormente, o passaporte contém a

assinatura digital do fabricante sobre o código do agente e com isso, é possível verificar a autenticidade e a integridade do código recebido.

Quanto ao uso de criptografia de chaves públicas, em [80], os autores relatam que Ara supõe a pré-existência de uma infra-estrutura de chaves públicas e, assim, não abordam esta questão. Para a verificação das assinaturas digitais do usuário e fabricante do agente, são utilizados os certificados digitais contidos no próprio passaporte.

5.5 D'Agents

Informações Gerais

D'Agents [29, 30, 32], antigamente denominado de Agent Tcl, tem sido desenvolvido na Dartmouth College desde 1995 e é patrocinado por diversas entidades como DARPA e Office of Naval Research. O objetivo principal do sistema é suportar aplicações de recuperação, organização e apresentação de informação distribuída em redes arbitrárias. Os agentes podem ser escritos em Tcl, Java e Scheme. O código fonte do sistema é disponível para uso não comercial.

Descrição do Sistema

A arquitetura do sistema D'Agents é composta pelos quatro níveis hierárquicos, ilustrados na Figura 5.1. O nível mais baixo é uma API para os mecanismos de transporte disponíveis. O segundo nível é o servidor D'Agents que executa em cada máquina. Ele é responsável por manter a lista dos agentes sendo executados na máquina, prover primitivas para comunicação entre agentes e receber um agente migrante, reinicializando-o, após autenticado, no interpretador apropriado. O terceiro nível contém os ambientes de execução para cada linguagem suportada e a interface deste nível com o inferior é feita por meio de bibliotecas escritas em C/C++. Por fim, a última camada é composta pelos agentes, que podem ser móveis ou estacionários.

O esquema é flexível e permite a inclusão de novas linguagens. Para isso basta criar um novo módulo, no terceiro nível, para a linguagem a ser incluída. Cada módulo é composto de um interpretador para a linguagem, funções para captura/recuperação do estado de um agente, módulo para evitar ações maliciosas dos agentes e uma API para acessar os serviços do servidor.

Quando um agente deseja migrar para um novo servidor, a primitiva `agent_jump` é utilizada. Como o tipo de mobilidade depende das capacidades do interpretador utilizado, D'Agents apresenta mobilidade fraca ou forte, dependendo da linguagem

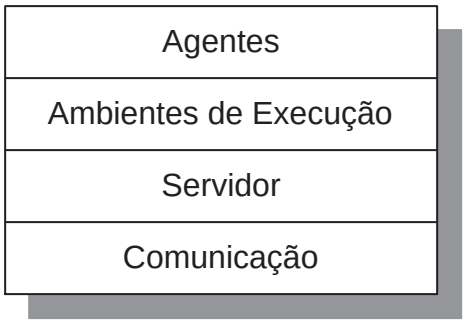

Figura 5.1: Arquitetura do sistema D'Agents.

com a qual o agente é escrito. Agentes escritos em Tcl e Java [1] têm seu estado capturado no nível de *thread* de execução, o que configura mobilidade forte. Já agentes escritos em Scheme apresentam apenas mobilidade fraca. Durante a migração, todo código necessário ao agente é enviado ao servidor destino.

A comunicação entre agentes é realizada de duas maneiras: passagem de mensagens e conexão direta. A primeira utiliza as primitivas `agent_send` e `agent_receive` que enviam e recebem uma mensagem, respectivamente. A segunda faz uso da primitiva `agent_meet` que estabelece uma conexão com o agente desejado. Esta forma é mais eficiente que a primeira para interações mais longas. As mensagens não possuem nenhum formato pré-definido e correspondem simplesmente a cadeias de caracteres. A semântica e a sintaxe das mensagens deve ser combinada pelas partes comunicantes. Agentes escritos em linguagens diferentes podem se comunicar normalmente.

Aspectos de Segurança

Os mecanismos de segurança destinados à proteção dos servidores em D'Agents envolvem, basicamente, as tarefas de autenticação e autorização. Estas tarefas são realizadas pelos subsistema de Criptografia, módulo de segurança dependente de linguagem e módulo de política de segurança independente de linguagem.

D'Agents classifica os agentes em ***owned*** e **anônimos**. Os primeiros são agentes cujos proprietários podem ser autenticados e constam da lista de usuários autorizados pelo servidor. Os segundos são os agentes para os quais qualquer uma das duas situações anteriores não é satisfeita. Agentes anônimos podem ou não ser aceitos por um servidor, dependendo da sua política de segurança.

Cada usuário e servidor do sistema possui um par de chaves RSA utilizadas para

[1] Uma versão modificada da máquina virtual Java 1.0 é utilizada

autenticação. D'Agents utiliza PGP para prover os serviços de ciframento e assinatura digital. Estes serviços podem ser usados pelos agentes durante migrações e troca de mensagens. Como é possível desabilitar estes serviços, os agentes devem considerar seu uso conforme cada caso. Uma vez que os algoritmos de chave pública são lentos, a desativação destes recursos implicaria um ganho de desempenho.

Para mostrar como um usuário é autenticado, vamos imaginar que um servidor X envia um agente pertencente ao usuário Y para o servidor Z. O proprietário do agente, Y, é autenticado se: (1) o agente é digitalmente assinado por Y; ou (2) o agente é digitalmente assinado por X, Z confia em X e X foi capaz de autenticar Y. É importante notar que X poderia ter autenticado Y da mesma forma que em (1) ou (2). Assim, temos que a confiança é transitiva.

Quando um agente registra-se em seu servidor base com o comando `begin` ou quando ele migra pela primeira vez com o comando `jump`, a requisição e o agente, respectivamente, são assinados digitalmente com a chave privada do proprietário do agente. Opcionalmente é possível cifrar, com a chave pública do servidor destino, os dados a serem enviados. Nas migrações seguintes, o agente é assinado com a chave privada do servidor atual e a autenticação se dá como explicado anteriormente.

Enquanto um agente migra por servidores dentro do conjunto considerado confiável, é possível autenticar seu proprietário direta ou indiretamente da maneira apresentada. Tão logo o agente deixe este conjunto de servidores, ele torna-se anônimo por todo o resto de sua existência.

Em D'Agents há dois tipos de recursos: **indiretos**, que podem ser utilizados somente por meio de um agente, e **nativos (*built-in*)**, que são acessíveis diretamente por meio de primitivas da linguagem. Entre exemplos deste último tipo estão recursos como memória e uso do processador.

O uso de recursos indiretos é controlado diretamente pelo agente responsável. Já os recursos nativos são controlados pelo servidor, pelos módulos de segurança específicos de linguagem e pelos **gerenciadores de recursos** independentes de linguagem. Estes são compostos por agentes estacionários que são consultados toda vez que um agente quiser acessar um recurso nativo. A consulta é realizada por intermédio dos módulos específicos de cada linguagem. Em ambos os casos, o acesso é concedido se o proprietário do agente possuir permissão para utilizar o recurso, conforme a política de segurança adotada. D'Agents possui seis gerenciadores de recursos distintos. Cada um deles controla uma classe de recursos diferente como o sistema de arquivos e serviços de rede.

Finalmente, D'Agents possui um mecanismo para proteção de um conjunto de máquinas quando as mesmas se encontram sob um mesmo domínio administra-

tivo. Cada agente carrega um vetor que determina, para cada recurso existente, o máximo que pode ser usado e o quanto já foi usado. Se o agente ultrapassar o limite estabelecido ele é finalizado. A atualização correta dos valores e o cumprimento dos limites são garantidos pelos módulos de segurança e gerenciadores de recursos.

Comentários

O artigo [32] cita diversas soluções para o problema de proteção do agente contra servidores maliciosos. Porém, como na maioria dos sistemas, nenhum destes mecanismos propostos foi implementado.

O modelo utilizado para autenticação, assim como em Aglets (Seção 5.2), depende de uma confiança estabelecida entre as máquinas. Isso torna o sistema pouco flexível e de difícil aplicação a sistemas que utilizam redes abertas como a Internet, por exemplo. Além disso, não existe uma infra-estrutura de chaves públicas. Os servidores são obrigados a saber previamente as chaves públicas de todos os servidores nos quais depositam confiança.

5.6 SOMA

Informações Gerais

O sistema de agentes móveis SOMA (Secure and Open Mobile Agent) [4, 12, 11] é um projeto em desenvolvimento no DEIS - Universidade de Bologna e iniciado em 1998. Os objetivos principais do projeto são segurança e interoperabilidade. SOMA suporta a linguagem Java (JDK 2) e as plataformas Win95/NT e Solaris. O binário e o código fonte do sistema estão disponíveis para uso não comercial e podem ser solicitados aos autores do projeto.

Descrição do Sistema

Um servidor de agentes em SOMA recebe o nome de ***place*** como ocorre em outros sistemas de agentes móveis. Cada servidor é composto por diversos módulos: o **Agent Manager**, responsável pela mobilidade e comunicação de agentes; o **Local Resource Manager**, que controla o acesso a recursos; o **Distributed Information Service**, que busca informações sobre servidores e agentes localizados remotamente; e o **CORBABridge**, para suporte a CORBA e MASIF.

O sistema SOMA utiliza uma hierarquia de abstrações de localidade para facilitar o uso do sistema em diferentes ambientes conectados por rede. Na base desta hierarquia estão os *places* que são agrupados em **domínios**. Cada domínio possui um servidor padrão que funciona como um ***gateway*** para comunicação com outros domínios. Estas abstrações podem ser diretamente mapeadas para recursos físicos:

um *place* corresponde a um nó da rede, enquanto que um domínio modela uma rede local.

SOMA suporta apenas mobilidade fraca obtida por meio da primitiva `go`. Quando chamada, esta primitiva faz com que o agente seja transferido para o novo servidor e que o método especificado como parâmetro seja executado após a migração. A transferência de código para o servidor destino é realizada sob demanda, conforme a necessidade do agente.

A comunicação entre agentes localizados em um mesmo servidor é feita por meio do compartilhamento de recursos comuns como um *buffer*, por exemplo. Já a comunicação com um agente remoto é realizada utilizando-se troca de mensagens. Mesmo que um agente migre para outro servidor, ele receberá eventualmente as mensagens que lhe são destinadas.

O objetivo de interoperabilidade é alcançado por meio de CORBA e MASIF: um agente pode atuar como um cliente CORBA e acessar objetos CORBA externos, ou se registrar como um servidor CORBA publicando sua interface em um ORB; além disso, entidades externas também podem acessar o sistema SOMA por meio da interface MASIF.

Aspectos de Segurança

SOMA estrutura as políticas de segurança de forma hierárquica baseando-se nas abstrações de localidade. Dessa forma, os domínios determinam autorizações e proibições gerais que podem ser restringidas por cada servidor de acordo com a política de segurança local adotada.

Cada agente carrega consigo **credenciais** que são informações não-forjáveis utilizadas no processo de autenticação. Dentre estas informações estão os nomes do domínio e do servidor origem, a classe que implementa o agente e o usuário do sistema por ele responsável. A autenticação ocorre tanto no domínio quanto no servidor e a aceitação de um agente migrante dependerá da política de segurança utilizada. SOMA integra uma infra-estrutura de chaves públicas da Entrust, Inc. para o gerenciamento de chaves e credenciais.

A autorização para utilizar um recurso depende dos **papéis (*roles*)** associados ao usuário responsável pelo agente. Os papéis definem quais tipos de operações podem ser realizadas sobre os recursos do servidor e sua utilização visa tornar o gerenciamento de permissões mais flexível. Os papéis e as políticas de controle de acesso são especificados na linguagem Ponder [15] e armazenados em arquivos cifrados. SOMA possui uma ferramenta gráfica para gerenciar estes arquivos em tempo de execução.

A confidencialidade de um agente durante a migração e de mensagens trocadas entre servidores é obtida com o uso de canais cifrados. É possível utilizar o algoritmo DES para cifrar as informações enviadas ou adotar o protocolo SSLv3. No caso do DES, a chave utilizada é transportada de um servidor para outro cifrada por RSA ou então estabelecida pelo protocolo Diffie-Hellman.

SOMA apresenta duas soluções para proteger o estado de um agente contra ataques perpetrados por servidores maliciosos: a primeira depende de uma terceira parte confiável (TTP) que valida os dados coletados pelo agente após cada visita a um servidor não confiável; e a segunda é um protocolo distribuído, chamado de ***Multiple-Hops***, que dispensa o uso da TTP. Estes protocolos são descritos em detalhes na Seção 3.4.13.

Comentários

O sistema SOMA apresenta uma grande diversidade de recursos de segurança abrangendo todas as classes de ataques listadas na Seção 3.1. Além disso, é um dos poucos sistemas a se preocupar com a segurança do agente móvel contra servidores maliciosos.

O protocolo baseado em TTP, como apontado pelos autores, não é adequado a redes abertas como a Internet devido a necessidade da existência de nós confiáveis que implementem a TTP. Além disso, o protocolo causa um impacto direto no desempenho do sistema ao obrigar que o agente passe pela TTP antes de migrar para o próximo servidor no itinerário.

A segurança do protocolo está baseada na suposição de que apenas as TTPs conhecem a função *hash* $h(x)$ utilizada para gerar a prova criptográfica de integridade do estado do agente. Isso é um problema, pois normalmente, a segurança de primitivas criptográficas não deve depender de tal suposição. Se uma entidade maliciosa descobre a função $h(x)$ empregada, ela pode gerar ou apagar itens do estado do agente sem que se possa detectar a modificação indesejada. Ao invés de um MDC, seria preferível o uso de um MAC cuja chave fosse compartilhada pelo conjunto de TTPs.

Outro problema encontrado é quando o agente visita um mesmo servidor malicioso S_M duas vezes. Durante a primeira visita, o servidor copia e armazena todo o estado do agente recebido. Como este estado, incluindo a prova criptográfica calculada pela TTP, é válido, basta que durante a segunda visita, S_M substitua o estado atual do agente por aquele armazenado anteriormente. Assim, todos os dados coletados entre as duas migrações são removidas com sucesso. Um ataque semelhante é possível se um servidor malicioso S_i envia o estado do agente recebido a um outro servidor, também malicioso, S_j. Se o agente visitar S_j, este pode substituir o estado daquele pelo enviado por S_i.

O protocolo *Multiple-Hops* é menos custoso que o baseado em TTP, mas ele também é vulnerável a ataques se um servidor puder ser visitado mais que uma vez por um mesmo agente. O servidor malicioso pode armazenar o segredo C_i recebido durante a primeira visita do agente e utilizá-lo, na segunda migração, para remover os dados inseridos pelos servidores S_j, com $j > i$. Um ataque similar pode ser realizado por dois servidores maliciosos S_i e S_k: S_i fornece a S_k o segredo C_i, o dado incluído, o MDC e a sua assinatura digital sobre o MDC. Com esses dados, S_k é capaz de remover os dados incluídos pelos servidores S_j, com $i < j < k$, quando o agente passar por S_k.

Finalmente, como não há controle sobre qual servidor está atualmente hospedando um agente, é possível a um servidor malicioso introduzir uma cópia de um agente simplesmente enviando-o a dois servidores ao mesmo tempo.

5.7 Ajanta

Informações Gerais

O sistema de agentes móveis Ajanta [45, 103, 46, 102, 47] é um trabalho desenvolvido na Universidade de Minnesota e iniciado em 1997. Suporta a linguagem Java 1.1.5 em plataformas Unix/Linux e necessita de Perl para ser instalado. O binário é disponibilizado gratuitamente para uso não comercial.

Descrição do Sistema

Em Ajanta, um servidor de agentes executa sobre uma máquina virtual Java e deve estar presente em todo nó da rede que deseje suportar aplicações baseadas em agentes. Cada servidor, além de executar agentes visitantes, oferece primitivas para comunicação, migração, controle e monitorização de agentes.

Ajanta apresenta mobilidade fraca como todos os sistemas baseados na máquina virtual Java original. Para realizar uma migração, o agente deve utilizar a primitiva `go`, que permite especificar qual método deve ser executado no momento que o agente chegar ao servidor destino. A transferência do código necessário à execução de um agente é feita sob demanda a partir de seu ***code base***. Este corresponde ao URN do servidor que é capaz de fornecer as classes do agente em questão.

Normalmente, o tratamento de exceções é uma tarefa que deve ser executada pelo próprio agente móvel. Quando, por algum motivo, o agente não consegue lidar com uma exceção, Ajanta o transporta até o **objeto guardião** responsável e notifica o problema ocorrido. Toda aplicação, de modo geral, cria um objeto guardião que é incumbido de tratar as exceções não resolvidas pelos agentes por ela utilizados.

Todas as entidades do sistema (como agentes, servidores e recursos) recebem um nome que as identificam unicamente. Estes nomes são independentes de localização e baseados no modelo URN (Uniform Resource Name) [93, 69]. Ajanta disponibiliza um **registro de nomes** responsável por fazer o mapeamento dos nomes das entidades para sua localização atual. Outra função deste serviço é atuar como uma entidade certificadora das chaves públicas utilizadas pelo sistema.

A comunicação entre agentes sendo executados no mesmo servidor é realizada por meio da invocação de métodos ou do acesso compartilhado a algum recurso. No primeiro caso, um agente se registra no servidor como um recurso e fornece *proxies* (explicado a seguir) aos agentes que desejem comunicar-se com ele. Além destes mecanismos de comunicação local, Ajanta permite que agentes interajam com pares localizados remotamente por meio de RMI autenticado.

Aspectos de Segurança

Conforme já mencionado, o registro de nomes atua também como uma entidade certificadora de chaves públicas para o sistema Ajanta. Agentes, servidores e usuários têm suas chaves públicas associadas a seus URNs, pelo registro de nomes. Criptossistemas de chaves públicas são utilizados em Ajanta para prover autenticação (DSA) e confidencialidade (ElGamal).

Cada agente possui, como parte de seu estado, **credenciais** contendo informações utilizadas para autenticá-lo. Entre essas informações estão o nome global do agente, o *code base* e as identidades (URNs) do **proprietário**, **criador** e objeto guardião. O proprietário é o usuário a quem o agente representa e esta identidade é utilizada para o controle de acesso. Por sua vez, o criador é a entidade que iniciou o agente, podendo ser outro agente ou uma aplicação.

A autenticação de entidades é feita com base em um mecanismo simples de desafio-resposta utilizando assinaturas digitais. Ao todo, são trocadas apenas três mensagens e a solicitação de serviço é enviada juntamente com a última mensagem. Este protocolo opera no nível de aplicação e permite autenticação mútua de URNs.

Em Ajanta, a migração de agentes entre servidores deve seguir o **protocolo de transferência de agentes**. A autenticação dos servidores envolvidos e o cifrамento dos dados transmitidos podem ser habilitados pelo agente migrante, conforme especificado em suas credenciais. O protocolo é realizado em três passos. No primeiro passo, o servidor atual envia a solicitação de transferência, juntamente com as credenciais do agente digitalmente assinadas pelo proprietário. Também é especificado o nome do método a ser executado pelo agente ao ser iniciado no servidor destino. Este verifica a assinatura sobre as credenciais e se pode ou não aceitar o agente conforme sua política de segurança. Dependendo da decisão, o agente é transferido

ou uma exceção é gerada. Finalmente, o servidor atual atualiza o registro de nomes para refletir a nova localização do agente.

O acesso a recursos de sistema é controlado pelo **Gerenciador de Segurança do Ajanta** que estende a classe `RMISecurityManager` de Java. O gerenciador utiliza listas de controle de acesso baseadas nos URNs de usuários para controlar o acesso a recursos como o sistema de arquivos e serviços de rede. Já a proteção de recursos de aplicação é realizada por meio da interposição de um objeto ***proxy*** entre o recurso e o cliente. O *proxy* possui a mesma interface que o recurso que ele protege, mas quaisquer métodos podem ser desabilitados, na construção do objeto, de acordo com a política de segurança adotada para aquele recurso. Os servidores também podem habilitar/desabilitar os métodos do *proxy* dinamicamente.

Para proteger os agentes contra ataques realizados por servidores maliciosos, Ajanta fornece três mecanismos que permitem detectar modificações indesejadas ao estado do agente. O primeiro consiste de **dados somente-para-leitura**, inicializados durante a criação do agente e que podem ter sua autenticidade e integridade verificadas. O segundo é um **contêiner somente-para-inclusão**, que permite incluir dados obtidos nos servidores visitados e protegê-los contra modificações e remoções. Por fim, o último mecanismo é chamado de **estado direcionado** e é composto de um vetor de dados em que cada posição tem servidores específicos como destinatários.

Os dados somente-para-leitura são a parte do estado do agente móvel que, após criada pelo proprietário do agente, não deve sofrer modificações. As credenciais de um agente são um exemplo de tais dados. Alterações indesejadas a esses dados podem ser detectadas se eles forem digitalmente assinados pelo proprietário. A assinatura é carregada pelo agente e pode ser verificada pelos servidores sendo visitados. Os outros dois mecanismos são analisados na Seção 3.4.7.

Comentários

O sistema Ajanta é o mais completo dos sistemas analisados do ponto de vista de segurança e possui ótima documentação dos mecanismos empregados. O sistema se preocupa com todas as classes de ataques apresentadas na Seção 3.1, inclusive ataques perpetrados por servidores maliciosos contra agentes móveis. Isso é um ponto positivo para Ajanta, uma vez que esta classe de ataques, nos outros sistemas, ou não é abordada ou é de forma bem superficial, com exceção do sistema SOMA.

Roth descreve em [85] um ataque que pode ser realizado contra o contêiner somente-para-inclusão. O ataque é possível quando um servidor malicioso S_M é visitado mais de uma vez pelo mesmo agente móvel ou quando dois servidores formam um conluio. Quando o agente alvo visita S_M pela primeira vez, o *checksum* do contêiner é copiado por S_M. Na segunda visita, o servidor malicioso é capaz de remover todos

os dados obtidos entre as duas visitas, simplesmente removendo os itens coletados e substituindo o *cheksum* atual pelo copiado anteriormente. Se repararmos bem, esse ataque é o mesmo que o descrito na Seção 5.6, contra os protocolos SOMA.

5.8 Grasshopper

Informações Gerais

Grasshopper [34, 35] é o sistema de agentes móveis desenvolvido pela IKV++ Technologies AG, em conformidade com a especificação MAF. O sistema suporta a plataforma Java 2 e pode ser executada em Windows e Unix. A primeira versão do sistema foi liberada em 1998. Os binários do Grasshopper são disponibilizados gratuitamente para uso não comercial.

Descrição do Sistema

Os elementos componentes de Grasshopper são modelados seguindo-se a especificação MAF. Os servidores são chamados de **agências** e são compostos de um **núcleo** e de um ou mais ***places***. O núcleo fornece serviços para comunicação, registro dos agentes e *places* localizados na agência, gerenciamento, migração, segurança e persistência. Um *place* é um agrupamento lógico de funcionalidades dentro de uma agência. Como exemplo, podemos citar um *place* que forneça serviços avançados de comunicação. O conceito de **região** também é implementado e consiste de um conjunto de agências gerenciadas por uma mesma organização.

Como o sistema é baseado na linguagem Java, ele apresenta apenas mobilidade fraca, baseando-se nos mecanismos de serialização da linguagem. A migração ocorre quando o método `move()` é chamado pelo agente ou quando o método `moveAgent()` é chamado por outro agente ou por uma agência. A transferência das classes necessárias é realizada sob demanda por meio de carregadores de classe Java.

Os serviços de comunicação de Grasshopper permitem interações transparentes de localização entre agentes e agências. Dessa forma, quando uma entidade deseja se comunicar com outra, não é necessário se preocupar com a localização atual de seu par. Isso é obtido por meio do uso de **objetos *proxies*** que são diretamente acessados por um cliente e se encarregam de repassar a mensagem ao objeto remoto. A comunicação entre entidades pode ser realizada de forma síncrona e assíncrona e utilizando-se diversos mecanismos como RMI Java e *sockets*.

Grasshopper permite armazenar agentes e *places* em mídia persistente e com isso, é possível restaurar o estado do sistema após uma eventual queda. Os mecanismos de persistência são classificados em **implícitos** e **explícitos**. Os primeiros não são

visíveis aos programadores e são empregados automaticamente, após configurados pelos administradores das agências. Por outro lado, os explícitos requerem que sejam ativados por um agente para funcionar. Por exemplo, um agente pode definir um intervalo de tempo para ser armazenado periodicamente, enquanto é executado; ou então, pode solicitar que seja suspenso, após um período de inatividade, e reiniciado, a partir da mídia persistente, quando acessado por outro agente.

Aspectos de Segurança

O controle de acesso a recursos das agências, em Grasshopper, está baseado no modelo de segurança da plataforma Java 2. Cada agente carrega consigo as identidades do **servidor de origem** e de seu **proprietário**, além da assinatura digital deste sobre o código do agente. Estas identidades são utilizadas durante a criação do agente recebido para associá-lo a um domínio de proteção. A política de segurança é definida em arquivos de políticas e os acessos são mediados pelo controlador de acesso de Java.

A proteção de dados transferidos entre servidores é feita por meio do protocolo SSL. Assim, cada agência possui uma chave privada e o certificado de chave pública correspondente que são utilizados no processo de autenticação de entidades do protocolo. Para prover comunicação segura, é necessário que a agência destino esteja executando servidores seguros (*socket*/SSL ou RMI/SSL) e que a agência cliente selecione o mecanismo apropriado para comunicar-se. Quando se tenta estabelecer uma conexão entre servidores, se somente uma das partes aceitar conexões seguras, uma exceção é gerada.

Comentários

Analisamos Grasshopper neste *survey* por ele ser considerado o sistema de referência da especificação MAF. Pode-se perceber, que o sistema não apresenta muitos mecanismos de segurança e se preocupa basicamente com a proteção de servidores e com a comunicação segura. Por meio da assinatura digital do proprietário, é possível verificar a integridade e autenticidade do código do agente. Nenhuma outra proteção é provida para resolver o problema do servidor malicioso.

5.9 Outros sistemas

5.9.1 TACOMA

O sistema de agentes móveis TACOMA (Troms∅ And COrnell Moving Agents) [40, 41, 39] é um projeto resultante da colaboração das Universidade de Troms∅, Universidade Cornell e Universidade da Califórnia de San Diego. TACOMA suporta agentes escritos

em C, C++, ML, Perl e Python, entre outras linguagens, e pode ser executado em Unix, Windows e PalmOS.

Diferentemente de outros sistemas, TACOMA não provê captura automática de estado. Assim, em uma migração, fica a encargo do agente decidir quais dados de seu estado serão transferidos para o servidor destino. Estes dados são gravados em uma ou mais **pastas (*folders*)**, sendo uma delas utilizada para armazenar o código do agente. Cada pasta contém um conjunto de itens que correspondem a cadeias de bits. A primitiva `meet` é utilizada para migrar o agente para outro servidor e requer um conjunto de pastas a ser transportado para o servidor destino.

TACOMA não implementa nenhum mecanismo de segurança para proteger agentes ou servidores. O controle de acesso aos recursos do sistema, dessa forma, é dependente dos mecanismos do sistema operacional.

5.9.2 Mole

O sistema de agentes móveis Mole [96, 3, 2] é um trabalho de mais de cinco anos desenvolvido no IPVR (Institute for Parallel and Distributed Computer Systems) da Universidade de Stuttgart. A última versão é a 3.0, cujo código fonte é disponibilizado gratuitamente para uso não comercial.

Como Mole utiliza a máquina virtual Java padrão, o sistema suporta apenas mobilidade fraca. Quando um agente migra, todas as classes utilizadas são transferidas de uma vez para o servidor destino. A comunicação entre agentes é feita estabelecendo-se uma sessão entre eles, após o que, a interação ocorre por meio de RMI ou passagem de mensagens.

Mole adota o modelo *sandbox* de segurança no qual os agentes de usuários não podem acessar diretamente nenhum recurso do sistema. O acesso é feito de forma controlada e segura por intermédio de agentes estacionários existentes nos servidores. Para evitar ataques de negação de serviço, o gerenciador de recursos controla o uso do processador, uso da rede, número de agentes filhos criados e tempo total no servidor. Além disso, cada servidor pode especificar o tipo de agente que aceita para execução.

5.9.3 NOMADS

O sistema NOMADS [97, 98, 99] é um projeto em andamento na Universidade de West Florida e suportado pelo DARPA. NOMADS tem como objetivo fornecer mobilidade forte e controle granular de recursos. Ele é baseado na lingugem Java e é compatível com as plataformas Windows NT, Solaris e Linux.

O sistema utiliza a máquina virtual Aroma que é compatível com a máquina virtual Java padrão e que foi projetada e implementada para atingir os objetivos listados ante-

riormente. Em uma migração, todas as classes necessárias são transferidas de uma vez só para o servidor destino. A comunicação entre agentes é feita por meio da troca de mensagens que podem conter qualquer objeto Java serializável.

A biblioteca de código nativo é capaz de limitar o uso de disco e de rede, evitando assim alguns ataques de negação de serviço. Três limites podem ser impostos: taxa de transferência, quantidade e espaço. Por exemplo, pode-se impor a um agente uma taxa de escrita na rede de 100KB/s e determinar que o espaço total em disco utilizado não ultrapasse 5MB.

A política de segurança pode ser estabelecida sobre um usuário individual ou sobre um grupo de usuários. Ela é responsabilidade do módulo **Policy Manager** do servidor (chamado de **Oasis**) que, na inicialização, lê um arquivo `policies` contendo a política sobre transferência de agentes, controle de execução, controle de acesso e uso de recursos.

5.10 Quadros sinóticos

Para finalizar este capítulo, resumimos, em forma de quadros sinóticos, os diversos aspectos dos sistemas analisados. A Tabela 5.1 se refere a características gerais dos sistemas de agentes móveis como linguagem empregada, mobilidade, transferência de código e comunicação. Na Tabela 5.2, fazemos uma correspondência entre os conceitos da especificação MAF e os conceitos empregados nos sistemas analisados. Por fim, as Tabelas 5.3 e 5.4 tratam dos mecanismos de segurança utilizados para proteção de agentes e servidores.

Sistema	Linguagem	Mobilidade	Transferência de código	Comunicação
Telescript	Telescript	forte	completa	invocação de método
Aglets	Java	fraca	sob demanda completa	passagem de mensagens
Concordia	Java	fraca	sob demanda completa	eventos colaboração
Ara	Tcl C C++	forte	completa	passagem de mensagens
D'Agents	Tcl Java Scheme	forte forte fraca	completa	passagem de mensagens
SOMA	Java	fraca	sob demanda	passagem de mensagens compartilhamento de recurso
Ajanta	Java	fraca	sob demanda	invocação de método compartilhamento de recurso RMI
Grasshopper	Java	fraca	sob demanda	invocação de método via *proxy*

Tabela 5.1: Resumo das características dos sistemas de agentes móveis analisados.

Sistema	Conceito MAF			
	Agente móvel	*Place*	Sistema de agentes	Região
Telescript	agente móvel	-	*place*	região
Aglets	aglet	contexto	Tahiti	-
Concordia	agente móvel	-	servidor Concordia	-
Ara	agente móvel	-	*place*	região
D'Agents	agente móvel	-	servidor	-
SOMA	agente móvel	-	*place*	domínio
Ajanta	agente móvel	-	servidor de agentes	-
Grasshopper	agente móvel	*place*	agência	região

Tabela 5.2: Correspondência entre os conceitos dos sistemas analisados e os conceitos da especificação MAF.

Sistema	Proteção do agente	
	x Agente	x Servidor
Telescript	Acesso a outro objeto somente por meio da interface pública	-
Aglets	Mecanismos Java; interposição *proxy*	-
Concordia	Mecanismos Java e ciframento dos agentes enquanto em disco	-
Ara	Cada agente é executado em um interpretador separado; comunicação entre agentes controlada e segura	Integridade e autenticidade do código por meio de assinaturas digitais; rastro dos servidores visitados
D'Agents	Acesso a outros agentes somente por meio de passagem de mensagens	-
SOMA	Cada agente utiliza um espaço de nomes separado	Integridade e autenticidade do código por assinatura digital; integridade do agente por meio dos protocolos MH e TTP
Ajanta	Mecanismos Java	Integridade do código e estado do agente
Grasshopper	Mecanismos Java	-

Tabela 5.3: Mecanismos para proteção do agente empregados nos sistemas analisados.

Sistema	Proteção do servidor		Comunicação segura
	x Agente	x Servidor	
Telescript	Limite no uso de recursos; autorização baseada na autoridade do agente	Autenticação de entidades	Autenticação com RSA e ciframento com RC4
Aglets	Autorização baseada no *code base* do aglet	Autenticação de um servidor como pertencente à teia de confiança	Autenticação e integridade via MAC em teia de confiança
Concordia	Autorização baseada na identidade do usuário responsável pelo agente	Autenticação de entidades	Autenticação e ciframento via SSL
Ara	Limitação no uso de recursos determinado por uma função de admissão	Autenticação dos servidores; regiões de confiança	Autenticação e ciframento via SSL
D'Agents	Autorização baseada no proprietário do agente; limite no uso de recursos em uma máquina ou em um conjunto de máquinas	Autenticação de entidades	Autenticação e ciframento via PGP
SOMA	Política de segurança especificada na linguagem Ponder e autorização baseada em *roles* associados a usuários	Autenticação de entidades	Ciframento por DES ou autenticação e ciframento por SSL
Ajanta	Autorização baseada no proprietário do agente; proteção de recursos por interposição *proxy*	Autenticação de entidades	Ciframento com ElGamal e autenticação com DSA
Grasshopper	Autorização baseada no proprietário e servidor origem do agente	Autenticação de entidades	Autenticação e ciframento via SSL

Tabela 5.4: Mecanismos para proteção do servidor e para comunicação segura empregados nos sistemas analisados.

Capítulo 6

Extensões aos Mecanismos de Segurança do Sistema Aglets

Este capítulo descreve os mecanismos de segurança adicionados ao sistema de agentes móveis Aglets, como resultado deste trabalho. O sistema Aglets foi escolhido por ser de código aberto e por possuir uma boa documentação da API e dos mecanismos de segurança empregados [51, 52, 76, 53]. Além disso, o sistema possui uma escassa quantidade de mecanismos de proteção, aumentando, dessa forma, o leque de mecanismos que podem ser implementados e incorporados. A versão utilizada foi a 2.0.2, que é composta por 365 classes, incluindo os exemplos. Uma dificuldade encontrada foi a análise do sistema pois o código está muito pouco comentado e muitas das classes são extremamente grandes e complexas.

Como visto na Seção 5.2, Aglets não possui nenhum mecanismo para proteção do agente contra servidores maliciosos. Assim, para uma maior usabilidade do sistema em ambientes reais, pareceu necessária a adição de alguns mecanismos de segurança à API de Aglets. Dessa forma, programadores poderiam oferecer algum nível de proteção aos aglets desenvolvidos, pelo menos contra alguns dos ataques e ameaças a que estão sujeitos.

Se observarmos os mecanismos de proteção analisados na Seção 3.4, vemos que muitos deles não são práticos por dependerem de suposições muito fortes (e.g., agentes cooperantes) ou pelo alto custo de implantação (e.g., *hardware* seguro, replicação); entretanto, nem todos os mecanismos apresentam essas restrições. Exemplos destes são os projetados para a verificação da integridade de dados coletados por um agente. Um dos problemas é que esta classe de mecanismos está sujeita a alguns ataques no caso de um servidor malicioso ser visitado mais de uma vez. Mesmo assim, em muitos contextos a proteção oferecida é suficiente e, por isso, incluímos um mecanismo deste tipo na lista de extensões. A escolha recaiu sobre o conjunto de mecanismos do sistema Ajanta por oferecer também outros tipos de proteção do estado do agente.

Outro mecanismo implementado foi o de assinaturas digitais com chaves blindadas, que é fruto de pesquisa realizada localmente [20, 19]. Por isso, as motivações principais para a inclusão desta técnica foram a criação de um protótipo que pudesse ser utilizado neste trabalho paralelo e a verificação da dificuldade de implementação do referido mecanismo.

As seções restantes deste capítulo descrevem as adições realizadas ao sistema Aglets.

6.1 Correção do método `getCertificate()`

O método `getCertificate()` da classe `com.ibm.aglets.AgletRuntime` é utilizado para obter um objeto `Certificate` a partir da forma codificada de um certificado digital. Cada tipo de certificado possui uma forma codificada específica que é utilizada para transmiti-lo como uma seqüência de bytes. Por exemplo, certificados X.509 são codificados utilizando-se ASN.1 DER.

Da maneira como a rotina estava implementada, procurava-se por um certificado, no *keystore* utilizado pelo sistema Aglets, que possuísse a mesma forma codificada que o do certificado desejado. Com isso, quando um agente migrava de um servidor para outro que não tinha o certicado correspondente em seu *keystore*, o aglet era identificado como anônimo, pois não era possível localizar o certificado de seu proprietário. Isso pode ser observado por meio da opção `AgletInfo` da interface gráfica do Tahiti, ao selecionar um aglet proveniente de outro servidor, desde que o *keystore* não possua o certificado necessário.

A correção consiste simplesmente de se construir o objeto `Certificate` a partir das próprias informações da forma codificada do certificado. Isso é facilmente conseguido por meio do método `generateCertificate()` da classe `CertificateFactory`.

6.2 Mecanismos Ajanta

Implementamos neste trabalho os três mecanismos para proteção de agentes fornecidos pelo sistema de agentes móveis Ajanta, que são descritos nas Seções 3.4.7 e 5.7. Embora utilizemos as mesmas técnicas descritas para prover a segurança de cada um dos mecanismos, utilizamos interfaces ligeiramente diferentes que as apresentadas em [45]. Em seguida, ilustramos estas interfaces e fazemos um breve comentário sobre cada método fornecido. Note porém, que as interfaces abaixo não são construções válidas da linguagem Java, mas apenas construções similares, objetivando ilustrar as operações fornecidas pelas classes reais.

6.2.1 `ReadOnlyContainer`

Esta classe, listada como Classe 6.1, permite utilizar um conjunto de atributos que são protegidos contra modificações e que podem ter sua autenticidade verificada, por meio de assinaturas digitais. Os atributos a serem protegidos, em cada caso, devem ser definidos, pelo programador do agente, em uma subclasse de `ReadOnlyContainer`. A classe possui apenas os métodos `sign()` e `verifySignature()`, que são utilizados para geração e verificação da assinatura, respectivamente.

Classe 6.1: ReadOnlyContainer - protege dados imutáveis

```
public class ReadOnlyContainer implements Serializable {
  public void sign(PrivateKey key);
  public boolean verifySignature(PublicKey key);
}
```

6.2.2 `AppendOnlyContainer`

Esta classe, listada como Classe 6.2, é utilizada para proteger dados coletados nos servidores visitados pelo agente móvel contra remoções e alterações. O contêiner permite apenas a inclusão de itens, não possibilitando que eles sejam posteriormente modificados. O método `add()` é usado para incluir no contêiner um objeto associado à uma entidade identificada pelo objeto `Principal`. Enquanto isso, os métodos `verify()` e `getObjects()` tem como função verificar a integridade do contêiner (e conseqüentemente de todo seu conteúdo) e obter os objetos incluídos, respectivamente.

O mecanismo está sujeito a alguns ataques que podem ocorrer no caso de um servidor malicioso ser visitado mais de uma vez. Para saber sobre esses ataques, consulte a parte de comentários da Seção 5.7.

Classe 6.2: AppendOnlyContainer - protege os dados incluídos

```
public class AppendOnlyContainer {
  public AppendOnlyContainer(PublicKey key, long nonce);
  public void add(Principal p, Object obj, byte[] signature);
  public boolean verify(PrivateKey key, long nonce);
  public Iterator getObjects();
}
```

6.2.3 `TargetedState`

A Classe 6.3 tem como objetivo possibilitar que o agente carregue objetos destinados a servidores específicos. Os diversos objetos direcionados são incluídos por meio do método `add()`, após o que, uma assinatura deve ser gerada chamando-se o método `sign()`. Uma entidade pode verificar a integridade e autenticidade do estado direcionado, conferindo a assinatura do proprietário do agente, com o método `verifySignature()`. Por fim, o método `getObjects()` devolve o conjunto de objetos endereçados à entidade especificada como parâmetro.

Classe 6.3: TargetedState - dados com destinatários específicos

```
public class TargetedState {
  public void add(Principal p, PublicKey key, Object obj);
  public void sign(PrivateKey key);
  public boolean verifySignature(PublicKey key);
  public Iterator getObjects(Principal p);
}
```

6.3 Assinaturas com chave blindada

Adicionamos o mecanismo de assinaturas com chaves blindadas ao sistema Aglets, definindo as interfaces dos componentes necessários para suportar esta primitiva e provendo implementações para as mesmas. A figura central é representada pelo **notário**, que corresponde a um servidor confiável cuja função é verificar assinaturas blindadas. Nosso servidor espera por conexões em uma porta especificada por uma propriedade do sistema e aceita somente conexões seguras, por meio do protocolo SSL. Somente o notário é autenticado, da mesma forma que ocorre na autenticação de servidores Web, pelos navegadores.

Embora, à primeira vista, pareça simples a criação de clientes e servidores SSL em Java, são necessárias diversas extensões à implementação fornecida, para se conseguir um maior controle e flexibilidade. Por exemplo, o *handshake* padrão apenas verifica que os certificados enviados possuem assinaturas válidas. Mas isso não significa muita coisa, pois um servidor `www.mentiroso.com` pode possuir um certificado assinado por uma autoridade certificadora válida e com isso, personificar um servidor, simplesmente apresentando este certificado durante o *handshake*. É necessário então, verificar que as informações contidas no certificado estão relacionadas à parte com a qual se está interagindo. Além disso, para controlar as chaves e certificados utilizados no estabelecimento da chave de

A	$\longrightarrow$	Notário:	`RegKey`	(1)
A	$\longleftarrow$	Notário:	desafio	(2)
A	$\longrightarrow$	Notário:	DN, fator de blindagem, resposta	(3)
A	$\longleftarrow$	Notário:	ID	(4)

Figura 6.1: Protocolo para registro de fator de blindagem.

sessão, é preciso estender e implementar diversas classes da API de Java relacionada a SSL (pacote `javax.net.ssl`).

Nossa implementação oferece suporte, por enquanto, apenas para assinaturas RSA. Quando se deseja incluir uma chave blindada em um agente, o primeiro passo consiste de registrar o fator de blindagem correspondente com o notário. Isso é realizado seguindo-se o protocolo ilustrado na Figura 6.1. A entidade A inicialmente indica que deseja registrar um fator de blindagem enviando a mensagem (1). O notário gera aleatoriamente um desafio e envia a mensagem (2). A então assina o desafio com a chave blindada e envia (3), que contém a parte DN de seu certificado de chave pública, o fator de blindagem correspondente e a resposta ao desafio. O notário então obtém a chave pública de A a partir de um repositório, verifica a assinatura sobre o desafio e, se esta estiver correta, ele gera o próximo identificador disponível para a entidade A (4).

De posse do identificador obtido no passo (4), a entidade A inclui essa informação e a chave blindada no agente móvel, que agora está capacitado a gerar assinaturas blindadas. Um agente pode assinar mensagens por meio do método `sign()` que devolve, além da assinatura, a mensagem codificada conforme o procedimento do documento [86]. A mensagem codificada, em geral, é muito menor que a mensagem original pois ela é resultado da compressão desta última por uma função *hash*. Devido a isso, utilizamos a forma codificada no processo de verificação de assinaturas, ao invés da mensagem original, para diminuir a quantidade de dados a serem enviados pela rede ao notário. É possível, porém, que a verificação de conformidade de uma assinatura blindada à política de uso, requeira a mensagem original. Neste caso, o protocolo deverá ser modificado.

Quando uma entidade B deseja verificar uma assinatura gerada por este agente móvel, ela deve utilizar o método `verifySig()`. Este método realiza os passos do protocolo ilustrado na Figura 6.2. Inicialmente, uma solicitação para a verificação de assinatura blindada é enviada ao notário (1). O notário responde com um `ACK` indicando que ele está pronto para receber os dados para verificação (2). A entidade B então envia o DN, ID carregado pelo agente, a mensagem codificada e a assinatura digital (3). O notário então obtém o fator de blindagem associado ao par (DN, ID) e a chave pública de B com base no valor de DN. Com esses dados ele verifica a assinatura e retorna o resultado a B (4).

B	$\longrightarrow$	Notário:	`VerSig`	(1)
B	$\longleftarrow$	Notário:	`ACK`	(2)
B	$\longrightarrow$	Notário:	DN, ID, mensagem codificada, assinatura	(3)
B	$\longleftarrow$	Notário:	`true` ou `false`	(4)

Figura 6.2: Protocolo para verificação de assinatura blindada.

Capítulo 7

Conclusões

Neste trabalho, introduzimos o conceito de sistemas de agentes móveis e de seus principais componentes. Os conceitos foram dados com base nas visões dos autores da maioria dos artigos estudados. Percebemos que não existe um consenso na nomenclatura utilizada. Por exemplo, o termo servidor de agentes, que usamos ao longo do texto, recebe os mais diferentes nomes como constatado na Tabela 5.2. E isso, porque a tabela não inclui os termos usados isoladamente em artigos que não estão relacionados a um sistema de agentes móveis específico, como aqueles que tratam de segurança. Por outro lado, verificamos que há uma grande concordância no que seja um sistema de agentes. Por isso, parece-nos estranho que a especificação MAF use esse termo para denominar um servidor de agentes.

Em nosso estudo sobre as classes de ameaças a que estão sujeitas as entidades de tais sistemas, ficou óbvio que o problema mais difícil de ser tratado é o da proteção do agente móvel contra servidores maliciosos. A dificuldade reside no fato do agente ter de ser executado pelos servidores, que então têm acesso a toda informação não cifrada contida no agente, além de controle sobre a execução do mesmo. E como eventuais mecanismos de segurança serão executados pelos próprios servidores, estes também podem ser subvertidos.

Muitas das soluções propostas na literatura para a proteção do agente móvel não são práticas, pois adotam suposições extremamente restritivas, que não seriam satisfeitas em sistemas reais. Por sua vez, os mecanismos desenvolvidos para proteger o estado do agente contra remoções e modificações não-autorizadas, podem sofrer o mesmo ataque que o descrito na Seção 5.6, que ocorre quando um agente visita mais de uma vez um mesmo servidor malicioso. Isso ocorre porque a prova de integridade sempre pode ser armazenada pelo atacante durante a primeira visita do agente. Uma técnica promissora, mas que precisa ser generalizada para ser útil, é a de computação com funções cifradas (Seção 3.4.6). Já o uso de *hardware* seguro é uma solução que funciona, mas que é muito custosa e de difícil implantação.

Do lado da proteção do servidor, o cenário é mais animador pois as técnicas existentes proporcionam um resultado satisfatório para conter as ameaças possíveis. Normalmente, o que se emprega são mecanismos de autenticação e autorização, sendo que todo acesso a algum recurso é controlado por um monitor de referências. Como a maioria dos sistemas de agentes móveis é construída em cima de linguagens interpretadas, a adição de verificações de segurança é relativamente simples, pois todo comando já é mediado pelo interpretador.

A plataforma Java, que é base para uma grande gama de sistemas de agentes móveis, possui uma arquitetura de segurança bastante elaborada e cuja compreensão é muito importante no desenvolvimento das camadas de segurança do tipo de sistemas que tratamos neste estudo. Por isso, dedicamos um capítulo abordando este tópico e mostrando a relação desta arquitetura com a segurança de sistemas de agentes móveis.

A maioria dos sistemas de agentes móveis que analisamos neste trabalho se preocupa apenas com a proteção dos servidores contra agentes móveis maliciosos. O conjunto de técnicas empregadas é muito similar, principalmente porque muitos destes sistemas são escritos na linguagem Java e então se baseiam no modelo de segurança da linguagem. Com relação à proteção de agentes, há pouca coisa desenvolvida. Basicamente, são utilizadas assinaturas digitais para prover integridade e autenticidade do código e das partes imutáveis do agente e algum mecanismo específico para a proteção de informações coletadas por eles. Encontramos algumas falhas nos mecanismos de segurança utilizados por alguns destes sistemas, as quais foram apontadas e discutidas nesta dissertação.

Resumindo tudo isso, temos que sistemas de agentes móveis estão sujeitos a uma grande variedade de ataques e ameaças, muitas das quais são satisfatoriamente tratadas por mecanismos tradicionais ou extensões. Por outro lado, apresenta-se extremamente desafiador e interessante o problema da segurança do agente móvel e que, no momento, carece de soluções gerais. Tentamos neste trabalho apresentar um estudo abrangente e detalhado de todos estes aspectos.

Como principais contribuições deste trabalho, ressaltamos:

- Estudo abrangente, auto-contido e com certa profundidade dos aspectos de segurança de sistemas de agentes móveis, cobrindo as ameaças, requisitos de segurança e mecanismos de proteção propostos na literatura.

- *Survey* detalhado dos principais sistemas de agentes móveis, com ênfase em suas arquiteturas de segurança, no qual relatamos algumas fraquezas encontradas nos mecanismos de segurança utilizados pelos sistemas analisados.

- Adição de alguns mecanismos de segurança ao sistema de agentes móveis Aglets, dos quais ressaltamos a assinatura com chaves blindadas.

- Desenvolvimento de um conjunto de classes que permitem um maior controle e flexibilidade de conexões estabelecidas pelo protocolo SSL, na linguagem Java.

Como trabalhos futuros, vemos os seguintes tópicos:

- Estudar formas de se especificar políticas de uso para chaves blindadas, que possam ser verificadas de forma automática pelo notário. As políticas não podem ser muito restritivas, a ponto de descaracterizar o paradigma de agentes móveis, e nem muito flexíveis de modo a permitir o forjamento de assinaturas.

- Estudar o uso de funções cifradas para a proteção dos agentes móveis, procurando generalizar a aplicação deste mecanismo. Este tópico é extremamente difícil e pode não possuir uma solução geral mas parece ser, além do uso de *hardware* seguro, uma das poucas soluções efetivas.

- Estudar diretrizes de programação segura e aplicá-las, adaptando-as se necessário, ao desenvolvimento dos próprios sistemas de agentes móveis e a programas desenvolvidos para os mesmos.

Apêndice A

Fundamentos

A.1 Breve introdução à segurança da informação

Há algumas décadas, as informações eram normalmente armazenadas e transmitidas em papéis. Nos dias de hoje, o armazenamento é feito em mídias magnéticas e ópticas e a distribuição é realizada por sistemas de telecomunicação. Neste novo contexto eletrônico, tornou-se extremamente fácil realizar uma cópia idêntica de uma informação ou alterá-la de alguma maneira. Além disso, com redes interconectando computadores no mundo inteiro, a informação está cada vez mais acessível de qualquer ponto do globo. Junto com esse mundo digital, veio também a necessidade de novos mecanismos para a segurança da informação.

A segurança da informação possui diversos objetivos que variam de acordo com a aplicação e seus requisitos. A seguir, listamos alguns desses objetivos:

- Privacidade ou confidencialidade – permitir que somente as pessoas autorizadas tenham acesso a uma informação.
- Integridade de dados – garantir que um dado não sofra alterações não autorizadas de forma imperceptível.
- Autenticação de entidades ou identificação – comprovar a identidade de uma entidade.
- Autenticação de mensagens – comprovar a origem de um dado.
- Não-repúdio – impedir que uma entidade participante de uma transação ou comunicação negue os atos realizados.
- Anonimato – esconder a identidade de uma entidade.

- Controle de acesso – restringir o acesso aos recursos apenas às entidades autorizadas.
- Certificação – endosso de uma informação por uma entidade confiável.
- Autorização – permitir oficialmente a realização de alguma operação.

A.2 Criptografia

A **Criptografia** moderna é um conjunto de técnicas matemáticas e computacionais utilizadas para satisfazer alguns requisitos da segurança da informação como confidencialidade, integridade e autenticação de entidades. Enquanto isso, **criptoanálise** se refere ao estudo de técnicas matemáticas para se quebrar os diversos mecanismos criptográficos existentes. O estudo conjunto da criptografia e da criptoanálise é chamado de **criptologia**. A história da criptografia foi escrita pela "guerra" entre criptoanalistas e criptógrafos: quando estes criavam um novo mecanismo, aqueles se ocupavam de encontrar possíveis fraquezas que pudessem ser exploradas. Boas referências técnicas para a área são os livros [61, 95, 91, 94]. Para saber mais sobre a história da criptografia, recomendamos os livros [42, 92].

Nas seções seguintes, introduzimos brevemente as principais ferramentas providas pela criptografia.

A.2.1 Cifras

Uma cifra é um mecanismo criptográfico utilizado para prover confidencialidade de informação. É composta por uma transformação de **ciframento** e outra transformação de **deciframento**, como ilustrado na Figura A.1. A primeira recebe como entrada um **texto em claro** m e a **chave de ciframento** k_1 e origina um **texto cifrado** c, que provê a confidencialidade de m. Dessa forma, o texto cifrado pode ser enviado para o destinatário por canais potencialmente inseguros, sem a preocupação que um atacante intercepte o canal de comunicação e obtenha m. A segunda transformação é utilizada para recuperar a mensagem original por meio de uma **chave de deciframento** k_2. Os esquemas de ciframento podem ser considerados **simétricos** ou **assimétricos**, dependendo da relação entre as chaves k_1 e k_2.

Cifras simétricas

A cifra é dita simétrica ou de **chave secreta** quando é computacionalmente fácil determinar k_2 a partir de k_1 e vice-versa. Em muitos casos práticos, temos que $k_1 = k_2$. Em

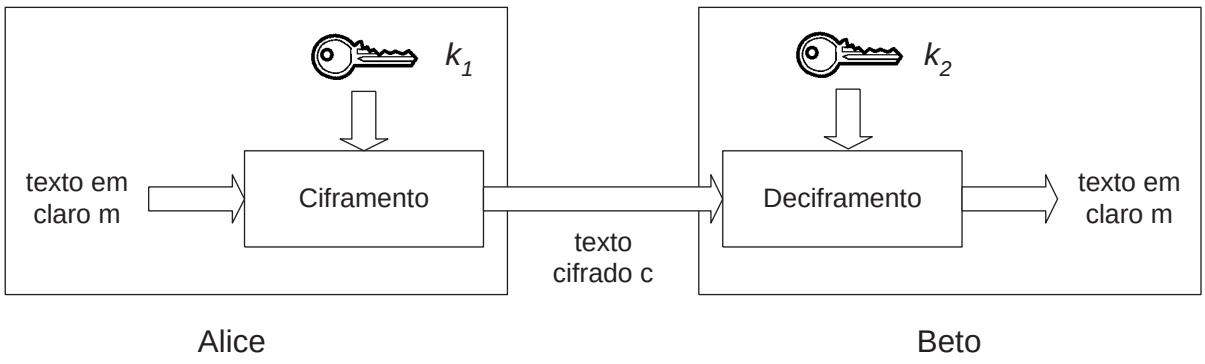

Figura A.1: Esquema de ciframento.

qualquer caso, as chaves só devem ser conhecidas pelas partes comunicantes. Os algoritmos desta classe são rápidos e utilizam chaves relativamente pequenas. Um problema destes esquemas é que o número de chaves em uma rede tende a ser grande. Isso ocorre porque quaisquer duas entidades que queiram se comunicar de forma confidencial devem compartilhar um par de chaves (k_1, k_2). Em uma rede com n entidades, se considerarmos que todo mundo deseja comunicar-se seguramente com o restante das pessoas, o número total de par de chaves seria $C_{n,2}$.

Um ponto a observar é que agora temos que nos preocupar em como concordar com o uso de um par de chaves específico sem que nenhuma terceira parte tome conhecimento delas. Essa troca de chaves pode ser feita pessoalmente ou por meio de protocolos criptográficos para estabelecimento de chaves, como o protocolo Diffie-Hellman.

As cifras simétricas podem ser classificadas em **cifras de blocos** e em **cifras de fluxo (*stream ciphers*)**. As primeiras são aquelas que dividem o texto em claro em blocos de tamanho t bits e processam um bloco por vez, com a mesma chave de ciframento. As cifras de fluxo, por sua vez, podem ser consideradas cifras de blocos simples com tamanho $t = 1$ e que podem variar a chave de ciframento utilizada para transformar cada bloco. A seqüência de chaves utilizadas é chamada de **fluxo de chaves (*keystream*)**

Como exemplos de cifras simétricas podemos citar DES, AES e RC4.

Cifras assimétricas

Uma cifra é chamada de assimétrica ou de **chave pública** quando é computacionalmente infactível derivar a chave de deciframento k_2 a partir da chave de ciframento k_1. Neste esquema de ciframento, k_1 recebe o nome de **chave pública** e k_2, de **chave privada**. Somente esta última deve ser mantida em segredo, enquanto que a primeira pode ser conhecida por qualquer entidade, como se pode inferir pelo seu nome.

Estas cifras são muito mais lentas que as cifras simétricas e necessitam de chaves

maiores para garantir um mesmo nível de segurança. Por outro lado, o número de chaves utilizadas é bem menor: cada entidade precisa possuir apenas um par (k_1, k_2) e assim, em uma rede com n entidades, haverá apenas n pares de chaves.

Para ilustrarmos um problema destes esquemas, vamos imaginar o seguinte cenário: Alice deseja enviar uma mensagem secreta para Beto. Ela então obtém a chave pública dele de uma fonte qualquer (uma página da Web, um anúncio de jornal, etc), utiliza-a para cifrar a mensagem e envia o resultado a Beto. Como somente Beto possui a chave de deciframento correspondente, somente ele será capaz de recuperar a mensagem original. Mas o que aconteceria se Eva, curiosa para saber das mensagens alheias, tivesse fornecido sua chave pública a Alice como sendo a de Beto? Como Eva possui a chave privada correspondente, ela é capaz de obter a mensagem original de Alice e, em seguida, cifrá-la com a chave pública de Beto, enviando-lhe o resultado. No fim das contas, Alice nunca saberá que a sua mensagem foi revelada a uma outra parte, que não Beto. O ataque foi possível porque não verificou-se a **autenticidade** da chave pública utilizada. Esse problema pode ser resolvido por meio de **certificados de chave pública**.

Exemplos desta classe de cifras são RSA e ElGamal.

A.2.2 Funções de espalhamento criptográficas

Também chamadas simplesmente de funções *hash*, segundo definição dada em [61], uma função de espalhamento criptográfica é uma função computacionalmente eficiente que mapeia cadeias binárias de tamanho arbitrário para cadeias binárias de tamanho fixo qualquer, chamadas de **valores *hash***. Estes valores se constituem numa espécie de impressão digital da cadeia mapeada.

Uma função de espalhamento criptográfica $h(x)$ deve ter algumas propriedades básicas relacionadas a seguir:

- resistência da pré-imagem – para essencialmente todos os valores *hash* y, é computacionalmente infactível encontrar qualquer pré-imagem x tal que $h(x) = y$ (desde que já não se conheça um tal x, obviamente).

- resistência da segunda pré-imagem – dado um x qualquer é computacionalmente infactível encontrar um $x' \neq x$ tal que $h(x) = h(x')$.

- resistência a colisões – é computacionalmente infactível encontrar quaisquer dois valores distintos, x e x', tal que $h(x) = h(x')$.

Uma classificação funcional das funções de espalhamento criptográficas, comumente usada, é a seguinte:

- **Código de detecção de modificações (MDC)** – a idéia é prover um representante de uma mensagem que permita verificar a integridade desta.

- **Código de autenticação de mensagens (MAC)** – é empregado para a verificação da autenticidade da origem de uma mensagem, bem como sua integridade. Além da mensagem, um MAC também requer uma chave secreta como entrada.

Alguns exemplos de funções de espalhamento criptográficas são MD4, MD5 e SHA-1.

A.2.3 Assinaturas digitais

A assinatura digital é uma primitiva criptográfica essencial para prover autenticação, autorização e não-repúdio. Por meio dela é possível a uma entidade associar sua identidade a alguma informação. Diferentemente de assinaturas manuais, que são constantes, a assinatura digital é uma função da mensagem sendo assinada e de algum segredo (uma chave) conhecido apenas pelo assinante. Um requisito importante para a efetividade deste mecanismo é que deve ser computacionalmente infactível construir uma mensagem para uma assinatura existente, bem como construir uma assinatura fraudulenta para uma mensagem qualquer.

Um **esquema de assinaturas digitais** consiste de um **algoritmo de geração de assinaturas** e de um **algoritmo de verificação de assinaturas**. O primeiro é um método que produz uma assinatura, dadas a mensagem e a chave privada do assinante. Por sua vez, o segundo é um método para verificar que a assinatura é autêntica e, portanto, foi criada pela entidade especificada.

Os esquemas de assinaturas digitais podem ser classificados em:

- **Esquemas de assinaturas digitais com apêndice** – requerem a mensagem original para o processo de verificação.

- **Esquemas de assinaturas digitais com recuperação de mensagens** – o processo de verificação não requer a mensagem original, que é recuperada a partir da própria assinatura.

Exemplos: RSA, DSA e ElGamal.

A.2.4 Certificados de chave pública

Um certificado de chave pública contém um conjunto de informações sobre uma determinada entidade, além da chave pública da mesma. Uma entidade confiável, normalmente chamada de **autoridade certificadora**, assina digitalmente o certificado e, dessa forma,

atesta que a chave pública e os dados contidos no certificado pertencem à mesma entidade. Se voltarmos ao cenário descrito na Seção A.2.1, Eva não mais consegue fazer Alice acreditar que a sua chave pública é a de Beto, pois seu certificado não contém as informações dele, necessárias à validação da chave pública.

A.2.5 Alguns protocolos criptográficos

Nesta seção veremos um protocolo para estabelecimento de chaves (Diffie-Hellman) e outro para autenticação de entidades (desafio-resposta).

Protocolo Diffie-Hellman

O protocolo Diffie-Hellman permite que duas entidades A e B estabeleçam uma chave simétrica K utilizando para tanto um canal aberto. A segurança do esquema se baseia no problema do logaritmo discreto e no problema de Diffie-Hellman [61]. O protocolo é apresentado a seguir:

1. Configuração inicial do sistema, realizada uma única vez: um primo p e um gerador α de $\mathbb{Z}_p^*$ ($2 \leq \alpha \leq p-2$) são selecionados e publicados.
2. Mensagens do protocolo:
   (a) $A \longrightarrow B : \alpha^x \bmod p$
   (b) $B \longrightarrow A : \alpha^y \bmod p$
3. Ações do protocolo:
   (a) A escolhe aleatoriamente um valor $x, 1 \leq x \leq p-2$ e envia a mensagem (2a).
   (b) B escolhe aleatoriamente um valor $y, 1 \leq y \leq p-2$ e envia a mensagem (2b).
   (c) B recebe α^x e calcula $K = (\alpha^x)^y \bmod p$.
   (d) A recebe α^y e calcula $K = (\alpha^y)^x \bmod p$.

Protocolos de desafio-resposta

A classe de protocolos de desafio-resposta é utilizada para a autenticação de entidades. Nestes protocolos, a entidade que deseja provar sua identidade deve mostrar conhecimento de um segredo que somente ela sabe sem, no entanto, revelá-lo durante a execução do protocolo. A entidade argüidora então, lhe fornece um *nonce* que deve ser usado juntamente com o segredo para o cálculo da resposta. Estes protocolos podem ser implementados com base em diversas primitivas criptográficas [61]. Vejamos um exemplo que utiliza deciframento assimétrico e **testemunha**:

1. Mensagens do protocolo:
   (a) $B \longrightarrow A : H(r), B, E_{K_A}(r \mid\mid B)$
   (b) $A \longrightarrow B : r$
2. Ações do protocolo:
   (a) B escolhe um valor aleatório r e calcula a testemunha $x = H(r)$ que demonstra conhecimento de r sem revelá-lo. Em seguida, B calcula $c = E_{K_A}(r \mid\mid B)$ e envia a mensagem (1a).
   (b) A decifra c, obtendo r' e B'. Se $B' \neq B$ ou se $H(r') \neq H(r)$, A termina o protocolo; senão, A envia a mensagem (1b). B então verifica que o valor r recebido é o esperado e com isso, obtém-se a autenticação unilateral da entidade A, pois somente ela conhece a chave necessária para decifrar c e obter r.

Apêndice B

Outros Tópicos sobre Segurança em Java

B.1 O método `doPrivileged()`

O algoritmo básico de inspeção de pilha apresentado na Seção 4.8 impede que métodos de domínios menos privilegiados obtenham permissões adicionais chamando métodos de domínios mais privilegiados. Algumas vezes, porém, esse algoritmo pode ser muito restritivo ao obrigar que todos os métodos na pilha de execução possuam as permissões necessárias. Como exemplo disso, Venners [108] cita a situação em que um *applet* solicita que a API Java escreva em uma janela um texto em fonte Helvética. É provável que o *applet* não possua as permissões necessárias para ler do disco o arquivo descrevendo a fonte Helvética o que impediria, pelo algoritmo de inspeção de pilha, que a API Java realizasse a renderização do texto.

O método `doPrivileged()` visa contornar esse problema ao indicar para o algoritmo de inspeção de pilha um término antecipado do processo sem a verificação de toda a pilha de execução. Dessa forma, códigos com mais privilégios podem realizar suas tarefas mesmo sendo chamados por códigos sem as permissões necessárias. Ao todo, há quatro versões do método `doPrivileged()` fornecidas pela classe `AccessController` [108, 73].

Para mostrarmos o funcionamento deste método, vamos inicialmente dar um exemplo de inspeção de pilha que resulta em uma permissão não concedida. Vamos supor a existência de um arquivo `/home/nelson/qualquer.txt` e de dois domínios de proteção que chamaremos de `AMIGO` e `ESTRANHO`. Somente o primeiro domínio, `AMIGO`, possui permissão para leitura do arquivo mencionado. Além disso, chamaremos de `SISTEMA` o domínio que é associado às classes da API Java e portanto, que possui todas as permissões possíveis.

Consideremos as classes `MenosPrivilegiada` e `LeitorPrivilegiado`, listadas a se-

guir, que estão associadas aos domínios de proteção `ESTRANHO` e `AMIGO`, respectivamente. Para que a inspeção de pilha funcione como desejado, é necessário iniciar a aplicação `MenosPrivilegiada` com o gerenciador de segurança concreto instalado. Isso é feito utilizando-se a opção `-D` no seguinte comando:

```
java -Djava.security.manager -cp .:Classes.jar MenosPrivilegiada
```

O *classpath* acima especificado pela opção `-cp` reflete a maneira como organizamos nossas classes e deve ser alterado de acordo com cada ambiente. Em nosso caso, armazenamos a classe `LeitorPrivilegiado` em `Classes.jar` residente no mesmo diretório que `MenosPrivilegiada.class`.

Classe B.1: MenosPrivilegiada.java - chama método de domínio mais privilegiado

```
public class MenosPrivilegiada {
  public static void main(String[] args) {
    LeitorPrivilegiado l = new LeitorPrivilegiado();

    l.leArquivo();
  }
}
```

Classe B.2: LeitorPrivilegiado.java - pode ler arquivo `/home/nelson/qualquer.txt`

```
import java.io.FileReader;

public class LeitorPrivilegiado {
  public void leArquivo() {
    try {
      FileReader fr = new FileReader("/home/nelson/qualquer.txt");
      // aqui deve vir o código para leitura do arquivo
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Quando a aplicação é iniciada, o método `main()` de `MenosPrivilegiada` é executado, criando um objeto `LeitorPrivilegiado` e invocando o método `leArquivo()` deste objeto. Este método cria um objeto `FileReader`, cujo construtor cria um objeto `FileInputStream`. O construtor deste último verifica que há um gerenciador de segurança instalado e então chama o método `checkRead()` deste último. O método `checkRead()`, por sua vez, chama o método `checkPermission()` do gerenciador, que invoca `checkPermission()` do controlador de acesso. Embora conceitualmente a inspeção de pilha seja realizada pelo controlador de acesso, podemos ver pela pilha de execução ilustrada na Tabela B.1 que esta tarefa é de fato implementada pelo método `checkPermission()` da classe `java.security.AccessControlContext`.

A Tabela B.1 mostra a pilha de execução da aplicação `MenosPrivilegiada` que será verificada pelo algoritmo de inspeção de pilha. De todos os *frames* da pilha, somente o de número 1 não possui os privilégios para leitura do arquivo `qualquer.txt`. O algoritmo de inspeção de pilha começa verificando se o método no topo da pilha (*frame* 8) possui os privilégios para leitura do arquivo e assim prossegue até chegar ao método `main()` no *frame* 1. Como o domínio de proteção `ESTRANHO` não possui as devidas permissões, o algoritmo de inspeção de pilha gera uma exceção do tipo `AccessControlException`.

Método	Domínio de Proteção	No.
`MenosPrivilegiada.main()`	`ESTRANHO`	1
`LeitorPrivilegiado.leArquivo()`	`AMIGO`	2
`java.io.FileReader.<init>()`	`SISTEMA`	3
`java.io.FileInputStream.<init>()`	`SISTEMA`	4
`java.lang.SecurityManager.checkRead()`	`SISTEMA`	5
`java.lang.SecurityManager.checkPermission()`	`SISTEMA`	6
`java.security.AccessController.checkPermission()`	`SISTEMA`	7
`java.security.AccessControlContext.checkPermission()`	`SISTEMA`	8

Tabela B.1: Pilha de execução da aplicação `MenosPrivilegiada`.

Vejamos agora o funcionamento do método `doPrivileged()`. Vamos considerar a mesma classe MenosPrivilegiada que a listada na Classe B.1, mas uma nova versão da classe `LeitorPrivilegiado` listada na Classe B.3. As classes foram organizadas da mesma forma que no exemplo anterior com relação ao *classpath*. Assim, o mesmo comando é usado para iniciarmos a aplicação.

Pouca coisa muda durante a execução da aplicação com a nova versão da classe `LeitorPrivilegiado`. Agora, o método `leArquivo()`, ao ser invocado, chama o método `doPrivileged()` passando uma instância de classe anônima interna que implementa `PrivilegedAction`. O método `doPrivileged()` invoca o método `run()` do objeto pas-

sado como parâmetro que então cria o objeto `FileReader`. Deste ponto em diante, a execução segue o mesmo fluxo que no exemplo anterior.

Classe B.3: LeitorPrivilegiado.java - com `doPrivileged()`

```
import java.io.FileReader;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class LeitorPrivilegiado {
  public void leArquivo() {
    AccessController.doPrivileged(
      new PrivilegedAction() {
        public Object run() {
          try {
            FileReader fr = new FileReader("/home/nelson/qualquer.txt");
          } catch (Exception e) {
            e.printStackTrace();
          }
          return null;
        }
      }
    );
  }
}
```

A Tabela B.2 mostra a pilha de execução do novo exemplo. Novamente, somente o *frame* 1 não possui privilégio de leitura do arquivo `qualquer.txt`. Em relação à Tabela B.1, temos dois novos *frames*, 3 e 4, que representam as chamadas a `doPrivileged()` e a `run()`, respectivamente. O algoritmo de inspeção de pilha inicia a verificação de privilégios a partir do topo da pilha (*frame* 10), como de costume, e prossegue até o *frame* 3, quando encontra o método `doPrivileged()`. Conforme já mencionado, este método sinaliza um término antecipado do algoritmo de inspeção que, então, somente verifica se o próximo *frame* (no caso, número 2) possui as permissões necessárias. Como o domínio de proteção `AMIGO` associado ao *frame* 2 possui os devidos privilégios, o algoritmo de inspeção pára a verificação neste *frame* e o arquivo `qualquer.txt` é aberto para leitura.

Método	Domínio de Proteção	No.
`MenosPrivilegiada.main()`	`ESTRANHO`	1
`LeitorPrivilegiado.leArquivo()`	`AMIGO`	2
`java.security.AccessController.doPrivileged()`	`SISTEMA`	3
`LeitorPrivilegiado$1.run()`	`AMIGO`	4
`java.io.FileReader.<init>()`	`SISTEMA`	5
`java.io.FileInputStream.<init>()`	`SISTEMA`	6
`java.lang.SecurityManager.checkRead()`	`SISTEMA`	7
`java.lang.SecurityManager.checkPermission()`	`SISTEMA`	8
`java.security.AccessController.checkPermission()`	`SISTEMA`	9
`java.security.AccessControlContext.checkPermission()`	`SISTEMA`	10

Tabela B.2: Pilha de execução da aplicação `MenosPrivilegiada` com `doPrivileged()`.

B.2 JCA e JCE

Java™ Cryptography Architecture (JCA) [64] é um *framework* para utilização e desenvolvimento de primitivas criptográficas na plataforma Java. JCA define, entre outras coisas, APIs para assinaturas digitais e para *message digests*. Enquanto isso, APIs para ciframento, estabelecimento de chaves e MACs são definidas no *framework* Java™ Cryptography Extension (JCE) [65] que, outrora um pacote opcional, agora foi incorporado na versão 1.4 da plataforma.

Tanto JCA quanto JCE foram projetados com os seguintes princípios em mente:

- independência e interoperabilidade de implementação; e
- independência e extensibilidade de algoritmos.

A independência de algoritmos é obtida por meio de classes de *engine* que definem as funcionalidades das primitivas criptográficas de forma abstrata. Exemplos dessas classes são `java.security.MessageDigest` e `javax.crypto.Cipher`. Assim, sabemos que ao chamarmos o método `digest()` de um objeto `MessageDigest`, independentemente de estarmos usando SHA-1 ou MD5, obteremos o *hash* da mensagem de entrada. Com essa estrutura adotada, a adição de um novo algoritmo qualquer é uma tarefa fácil.

A independência de implementação é obtida por meio de provedores de serviços criptográficos (*Cryptographic Security Providers*) também chamados apenas de provedores. Um provedor é composto por pacotes que implementam um subconjunto dos algoritmos e aspectos criptográficos definidos abstratamente pelas classes de *engine*. A interoperabilidade garante que podemos usar qualquer provedor para utilizar um dado algoritmo

(desde que o provedor implemente o algoritmo). Dessa forma, por exemplo, uma assinatura RSA gerada utilizando-se um provedor `A` deve ser verificável utilizando-se um provedor `B` qualquer para o mesmo algoritmo.

B.3 JAAS

O Java™ Authentication and Authorization Service (JAAS) [63, 73, 49] foi introduzido na versão 1.3 do Java 2 SDK como um pacote opcional e depois integrado na versão 1.4. O JAAS é um *framework* para autenticação e autorização baseadas em usuários do sistema. Nas versões anteriores, o controle de acesso era efetuado apenas com base na origem das classes e nas entidades que as assinavam, como visto no Capítulo 4.

O JAAS permite que sejam utilizados quaisquer serviços de autenticação conforme desejado. O administrador do sistema determina quais módulos de autenticação serão usados para uma dada aplicação através de diretivas contidas em um arquivo de configuração. A arquitetura do JAAS também permite que os aplicativos funcionem de forma independente dos serviços de autenticação. Assim, é possível utilizar um outro serviço de autenticação sem a necessidade de alterar os programas já existentes.

Um *subject* em JAAS é um serviço ou pessoa que solicita o uso de um recurso do sistema e que precisa ser autenticado. Uma vez realizada a autenticação, um objeto `javax.security.auth.Subject` é criado contendo diversas identidades chamadas de *principals*. Como exemplo de *principals* temos o nome de um usuário e seu número de documento de identidade.

O arquivo de política de Java foi estendido para possibilitar a concessão de permissões a *principals*, além de *code sources*. A cláusula seguinte é um exemplo da nova sintaxe:

```
grant principal javax.security.auth.x500.X500Principal "cn=Nelson" {
   permission  java.io.FilePermission "/home/nelson/tese.ps", "read";
};
```

A cláusula acima permite que código Java executado por "Nelson" leia o arquivo `/home/nelson/tese.ps`. Para isso, o *subject* associado ao contexto de controle de acesso atual precisa conter um objeto `X500Principal` cujo método `getName()` retorne `cn=Nelson`.

Para executar ações como um usuário em particular, a classe `Subject` fornece os métodos estáticos `doAs()` e `doAsPrivileged()`. Esses métodos associam o objeto `Subject` especificado ao `AccessControlContext` corrente e executam a ação passada como parâmetro. Quando uma operação potencialmente perigosa é realizada, o algoritmo de inspeção de pilha funciona da maneira usual; a diferença é que os *frames* da pilha de execução acima do *frame* de `doAs()` estarão associados a domínios de proteção contendo também permissões concedidas aos *principals* relacionados ao *subject* especificado.

B.4 *Keystore* e *truststore*

O sistema de gerenciamento de chaves criptográficas em Java é baseado na noção de um ***keystore***, que é um banco de dados contendo chaves privadas, chaves simétricas e certificados de chave pública. *Keystores* podem ser criados e manipulados por meio de métodos da API Java ou utilizando-se a ferramenta `keytool` fornecida com a distribuição padrão da plataforma Java.

Pode haver dois tipos de itens em um *keystore*: o primeiro, chamado de **item de chave**, consiste de uma chave simétrica ou de uma chave privada acompanhada da cadeia de certificados para a chave pública correspondente. Estas chaves são protegidas contra acesso indevido por meio de autenticação por senhas. A outra entrada possível chama-se **item de certificado confiável** e sua função é armazenar um certificado de chave pública. Esta entrada é assim chamada porque o proprietário do *keystore* confia na autenticidade da chave pública contida no certificado. A autenticidade é atestada por meio da assinatura digital de alguma outra entidade, sendo possível a existência de **certificados auto-assinados**, isto é, aqueles que são assinados com a própria chave privada que faz par com a chave pública sendo certificada.

Um ***truststore*** nada mais é que um *keystore*, mas que é utilizado para um propósito específico como veremos a seguir. No protocolo SSL, durante o *handshake*, é necessário apresentar um certificado digital e utilizar a chave privada correspondente para o estabelecimento da chave secreta a ser usada no ciframento do canal. A assinatura do certificado (ou da cadeia de certificados) deve ser verificada e isso é feito com a chave pública da autoridade certificadora emissora. Os certificados auto-assinados das autoridades conhecidas pelo sistema são obtidas a partir do *truststore*.

Apêndice C

Alguns Aglets Maliciosos

C.1 Ataque de negação de serviço

Classe C.1: DoSAglets.java - ataque de negação de serviço em Aglets

```
package exemplos;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.*;

public class DoSAglets extends Aglet {
  private class DoSThread extends Thread {
    Thread t;
    Byte[] ba = new Byte[50000];

    public void run() {
      t = new DoSThread();
      t.start();

      while (true)
        Math.acos(Math.atan(Math.asin(Math.random())));
    }
  }

  public void onCreation(Object init) {
```

```
    addMobilityListener(
      new MobilityAdapter() {
        public void onArrival(MobilityEvent e) {
          setText("Cheguei para te atacar!");
          Thread t = new DoSThread();
          t.start();
        }
      }
    );
    try {
      dispatch(new URL("atp://mozart.ic.unicamp.br:4634"));
    } catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
  }
}
```

C.2 Aglet imortal

Classe C.2: ImortalAglet.java - Aglet que "ressuscita" ao ser destruído

```
package exemplos;

import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class ImortalAglet extends Aglet {
  public void onDisposing() {
    try {
      System.out.println("Recriando ImortalAglet!");
      getAgletContext().createAglet(getCodeBase(),
        "exemplos.ImortalAglet", null);
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
```

```
}
```

Apêndice D

Acrônimos

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CORBA	Common Object Request Broker Architecture
DARPA	Defense Advanced Research Projects Agency
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DN	Distinguished Name
DNS	Domain Name System
DSA	Digital Signature Algorithm
HTML	HyperText Markup Language
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
JAAS	Java™ Authentication and Authorization Service
JAR	Java ARchive
JCA	Java™ Cryptography Architecture
JCE	Java™ Cryptography Extension
JDBC	Java Database Connectivity
JDK	Java Development Kit
MAC	Message Authentication Code
MAF	Mobile Agent Facility
MASIF	Mobile Agent System Interoperability Facility
MDC	[1]Manipulation Detection Code [2]Modification Detection Code

MIC	Message Integrity Code
NIST	National Institute of Standards and Technology
OMG	Object Management Group
ORB	Object Request Broker
OSI	[1]Open Source Initiative
	[2]Open System Interconnection
PRAC	Partial Result Authentication Code
PDA	Personal Digital Assistant
PGP	Pretty Good Privacy
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SQL	Structured Query Language
SSL	Secure Sockets Layer
TTP	Trusted Third Party
URL	Uniform Resource Locator
URN	Uniform Resource Name

Apêndice E

Glossário

Abstract Syntax Notation One (ASN.1) – é uma linguagem formal para descrever, de forma abstrata, mensagens a serem trocadas entre sistemas de computação distribuídos.

Advanced Encryption Standard (AES) – também conhecido como Rijndael, é uma cifra simétrica de blocos que pode utilizar chaves de 128, 192 e 256 bits e que opera sobre blocos de 128 bits.

American National Standards Institute (ANSI) – é uma organização privada e sem fins lucrativos que administra e coordena o sistema de padronização norte-americano.

applet – é um programa Java executado por um navegador Web e que é embutido em páginas HTML. Adiciona conteúdo dinâmico a páginas Web possibilitando exibir animações, tocar músicas e prover interatividade.

Application Programming Interface (API) – é a definição das interfaces para um dado conjunto de rotinas. São especificados o nome da rotina, parâmetros, retorno e o contrato, i.e., a tarefa a ser realizada.

área de métodos – é uma área compartilhada por todos os *threads* da máquina virtual Java que armazena estruturas de cada classe instanciada.

bytecode – é a linguagem de máquina da máquina virtual Java.

carga de classe – refere-se ao processo de encontrar a forma binária de uma classe ou interface com um dado nome e, a partir dessa forma binária, construir um objeto `Class` representando a classe ou interface. Essa forma binária é chamada de formato de arquivo `class`.

carregador de classes definidor – é o carregador que gera um objeto `Class` para uma classe a ser carregada. Isto é feito efetuando-se uma chamada ao método `ClassLoader.defineClass()`.

carregador de classes iniciador – é o carregador que inicia a carga de uma classe. No modelo de delegação, nem sempre o iniciador é o responsável pela definição da classe, o que pode ser realizada por um carregador de nível mais alto na hierarquia de delegação.

checkpoint – é o processo de se gravar o estado do sistema em mídia não-volátil. Com isso, no caso de uma falha, pode-se retornar o sistema ao último estado preservado por este mecanismo e continuar a execução a partir daquele ponto.

classe abstrata – é uma classe que implementa apenas parte de seus métodos. Serve de base para outras classes que a estendem e fornecem as implementações faltantes. Com isso, é possível prover apenas as operações comuns a um conjunto de classes e deixar a implementação de comportamento especializado para as subclasses.

classe aninhada – é uma classe cuja declaração ocorre dentro do corpo de outra classe ou interface.

classe anônima interna – é uma classe que estende uma classe ou implementa uma interface e que não recebe um nome. Tais classes são definidas ao mesmo tempo em que são instanciadas com `new`. A Classe B.3, ilustrada na página 120, utiliza uma classe anônima interna do tipo `PrivilegedAction`.

classe concreta – é uma classe que implementa todos os métodos definidos em sua interface.

classe interna – é uma classe aninhada não estática.

classe local interna – pode ser definida em blocos de códigos, como o corpo de um método ou construtor. Estas classes são locais aos blocos onde são definidas e não membras da classe que contém esses blocos. Elas somente podem ser acessadas dentro do bloco do qual fazem parte, da mesma forma que uma variável local.

Common Object Request Broker Architecture (CORBA) – infra-estrutura para objetos distribuídos padronizada pela OMG (Object Management Group). CORBA automatiza muitas tarefas de programação como registro e localização de objetos, entre outras.

Data Encryption Standard (DES) – também conhecido como Data Encryption Algorithm pela ANSI, é uma cifra simétrica de blocos que foi amplamente utilizada nas duas últimas décadas. Recentemente foi substituída como cifra padrão pelo AES. DES utiliza chaves de 56 bits e opera em blocos de 64 bits de texto em claro, produzindo 64 bits de texto cifrado.

deadlock – é uma situação na qual tem-se um conjunto de processos que não podem prosseguir porque cada um deles espera por um evento (como a liberação de um recurso, por exemplo) que depende de um processo do próprio conjunto. Como todos os processos estão em estado de espera, nenhum dos eventos necessários ocorrerá.

Distinguished Encoding Rules (DER) – é um conjunto de regras de codificação ASN.1 que permite transformar dados especificados na linguagem ASN.1 em um formato padrão que pode ser decodificado usando-se o mesmo conjunto de regras. As regras DER garantem que uma dada mensagem seja codificada de uma maneira única e assim, são utilizadas em aplicações que empreguem mecanismos de segurança que fazem uso de primitivas criptográficas.

Domain Name System (DNS) – é um modelo para gerenciamento de nomes de domínios da Internet de forma hierárquica e distribuída. O DNS é responsável por mapear nomes de máquinas para endereços IP e vice-versa.

`final` – é um modificador da linguagem Java que pode ser usado em declarações de classes, métodos e atributos. Este modificador indica que não pode haver extensão, redefinição e modificação, respectivamente.

formato de arquivo `class` – é uma representação binária independente de plataforma de uma classe ou interface.

gateway – dispositivo que interliga duas redes e pode operar em níveis a partir da camada 3 do modelo RM-OSI. Quando ele se encarrega do roteamento de pacotes é chamado de roteador e opera na camada de rede. Quando a sua função é traduzir mensagens de uma rede para outra, que utilizam protocolos diferentes para uma determinada camada, recebe o nome de tradutor de protocolo.

HyperText Markup Language (HTML) – é a linguagem utilizada para a criação de documentos na World Wide Web. A estrutura do documento é definida por meio da utilização de *tags* e atributos.

International Data Encryption Algorithm (IDEA) – é uma cifra simétrica de blocos que opera em blocos de 64 bits de texto em claro, produzindo 64 bits de texto cifrado. O tamanho da chave utilizada é de 128 bits.

International Organization for Standardization (ISO) – é uma organização, composta por institutos de padronização de 140 países e com escritório central em Geneva, Suíça, que coordena a definição de padrões internacionais e publica as versões finais dos mesmos.

Internet Engineering Task Force (IETF) – é uma comunidade internacional aberta composta por profissionais da área de redes de computadores que estão preocupados com a evolução da arquitetura da Internet e seu funcionamento. O trabalho da IETF é feito em diversas áreas como roteamento, transporte e segurança.

Java ARchive (JAR) – é um formato de arquivo independente de plataforma que agrega diversos arquivos. Este formato de arquivo também suporta compressão e permite que assinaturas digitais sobre itens individuais do conteúdo do arquivo sejam incluídas.

Java Database Connectivity (JDBC) – é uma API que permite acessar praticamente qualquer dado tabular a partir da linguagem de programação Java. Provê conectividade a uma grande quantidade de sistemas gerenciadores de bancos de dados, além de acesso a planilhas e arquivos.

ligação de classe – é o processo de combinar a forma binária de uma classe ao estado de execução da máquina virtual Java de modo que a classe possa ser executada. A ligação sempre é precedida pela carga da classe.

linguagem de descrição de página – é uma linguagem para descrever o *layout* e conteúdo de uma página impressa. Como exemplos, podemos citar as linguagens PostScript e PCL.

livelock – situação em que um processo não consegue terminar uma tarefa porque o servidor lhe fornece serviço indefinidamente.

MD5 – é um algoritmo de *hash* criptográfico cuja saída consiste de um valor de 128 bits. A entrada é processada, dividindo-a em blocos de 512 bits.

Message Authentication Code (MAC) – é um mecanismo utilizado para prover autenticidade da origem e integridade de uma mensagem. MACs recebem como entrada, além da mensagem, uma chave simétrica.

Message Integrity Code (MIC) – ver Modification Detection Code.

Mobile Agent Facility (MAF) – é uma especificação da OMG que atualiza a especificação MASIF. Ela trata da interoperabilidade entre sistemas de agentes móveis.

Mobile Agent System Interoperability Facility (MASIF) – é uma especificação da OMG, atualmente conhecida como MAF, que trata da interoperabilidade entre sistemas de agentes móveis.

Modification Detection Code (MDC) – também chamado de **código de detecção de manipulação** e de **Message Integrity Code (MIC)**, consiste de uma cadeia de bits representativa de uma mensagem que é utilizada para verificação de integridade.

Nome Completamente Qualificado (NCQ) – todo pacote, classe não aninhada, interface não aninhada e tipos primitivos possuem um nome completamente qualificado [27] definido da seguinte maneira:

- O NCQ de um tipo primitivo é a palavra-chave para o tipo primitivo. Ex.: `boolean`.
- O NCQ de um pacote com nome que não é subpacote de outro pacote é seu nome simples. Ex.: pacote `java`.
- O NCQ de um um subpacote é o NCQ do pacote que o contém seguido de "." e do nome simples do subpacote. Ex.: `java.lang`.
- O NCQ de uma classe ou interface não aninhada declarada em um pacote sem nome é o nome simples da classe ou interface. Ex.: `ClasseExemplo`.
- O NCQ de uma classe ou interface não aninhada declarada em um pacote com nome consiste do NCQ do pacote seguido de "." e o nome simples da classe ou interface. Ex.: `java.lang.Object`.

nonce – um valor utilizado não mais que uma vez para o mesmo propósito. Tipicamente é usado para prevenir ataques por repetição [61].

Object Management Group (OMG) – é um consórcio sem fins lucrativos que produz e mantém especificações de aplicações interoperáveis para a indústria de computadores. Como exemplos de especificações da OMG temos CORBA e MAF.

Object Request Broker (ORB) – é uma tecnologia de *middleware* que gerencia a comunicação e a troca de dados entre objetos. É responsabilidade do ORB prover transparência de localização, isto é, dar a impressão de que um objeto é local ao cliente, quando na verdade pode se encontrar em um servidor remoto. Um ORB também pode prover transparência de linguagem de programação, permitindo que um objeto ignore a linguagem utilizada pelo objeto com o qual interage. Os detalhes de implementação do ORB normalmente não são importantes para os desenvolvedores de sistemas distribuídos, que se preocupam apenas com a interface dos objetos.

opcode – parte de uma instrução em linguagem de máquina que define a operação a ser realizada.

Open Source Initiative (OSI) – é uma organização sem fins lucrativos dedicada ao gerenciamento e incentivo do código aberto.

Open System Interconnection (OSI) – é um padrão ISO para comunicação global que define um *framework* de rede para implementação de protocolos em sete camadas.

Personal Digital Assistant (PDA) – um dispositivo de mão que combina computação, telefonia/FAX, Internet e rede. Normalmente utiliza um mecanismo baseado em caneta óptica como entrada e, portanto, possui tecnologia de reconhecimento de escrita. Também é chamado de *palmtop*, computador de mão e computador de bolso.

Pretty Good Privacy (PGP) – é um programa para segurança de correio eletrônico desenvolvido por Philip Zimmermann. O programa utiliza IDEA para ciframento, RSA para assinaturas digitais e MD5 para gerar *hashes* criptográficos.

Printer Control Language (PCL) – é uma linguagem de descrição de páginas da Hewlett-Packard utilizada para controlar impressoras a laser.

RC4 – é uma cifra simétrica de fluxo desenvolvida em 1987 por Ronald Rivest para a RSA Data Security, Inc. O algoritmo utiliza chaves que variam de 1 a 256 bytes.

Remote Method Invocation (RMI) – possibilita efetuar uma chamada a um método de um objeto localizado remotamente e receber o resultado de volta, como se o objeto estivesse na máquina local.

Remote Procedure Call (RPC) – é um modelo para chamada de procedimentos remotos. Neste modelo um processo cliente envia os parâmetros para o procedimento a ser executado pelo processo servidor e tem sua execução bloqueada até que o resultado seja fornecido.

robustez de um sistema – a capacidade de um sistema de continuar funcionando apesar da existência de falhas em seus subsistemas ou componentes.

RSA – criado por **R**ivest, **S**hamir e **A**dleman, é o criptossistema de chaves públicas mais utilizado no mundo. Pode ser usado para ciframento e assinaturas digitais e sua segurança se baseia na dificuldade de se fatorar inteiros grandes. O criptossistema é padronizado pelo documento [86].

runtime package – um conjunto de classes e *interfaces* que pertencem a um mesmo pacote e que foram definidas pelo mesmo carregador de classes.

Secure Sockets Layer (SSL) – é um protocolo utilizado sobre *sockets* para fornecer comunicação segura e confiável entre duas partes. SSL permite autenticar as entidades comunicantes e provê integridade e confidencialidade do canal de comunicação. Originalmente projetado pela Netscape para uso em seus servidores seguros e navegadores, atualmente é mantido e desenvolvido pela IETF, que fez pequenas modificações no protocolo e rebatizou-o de Transport Layer Security (TLS).

serialização de objetos em Java – é um mecanismo que permite transformar qualquer objeto que implementa a interface `Serializable` em uma seqüência de bytes independente de plataforma.

sockets – é uma interface de programação desenvolvida para a linguagem C e utilizada para comunicação. Ela utiliza chamadas de sistema do Unix e adiciona algumas outras para suportar TCP/IP.

Structured Query Language (SQL) – é uma linguagem de pesquisa a bancos de dados relacionais.

thread – também chamado de *lightweight process*, é uma subtarefa de um processo que possui *program counter*, registradores e pilha próprios. Compartilha com os demais *threads* que compõem o processo a seção de dados, código e recursos do sistema operacional.

Transport Layer Security (TLS) – protocolo correspondente à versão 3.1 do protocolo SSL.

Uniform Resource Locator (URL) – é um endereço global para documentos e outros recursos localizados na Web. A primeira parte do endereço especifica o protocolo a ser usado (HTTP, FTP, etc.), a segunda, o endereço IP ou o nome de domínio do servidor e a terceira, informações de caminho do recurso.

Uniform Resource Name (URN) – serve como um identificador de recursos independente de localização.

Apêndice F

Notação

A Tabela F.1 ilustra a notação que utilizamos ao longo desta dissertação.

Notação	Significado
K	uma chave simétrica
K_A	chave pública da entidade A
K_A^{-1}	chave privada da entidade A
$E_K(x)$	ciframento simétrico de x utilizando a chave K
$D_K(x)$	deciframento simétrico de x utilizando a chave K
$E_{K_A}(x)$	ciframento assimétrico de x utilizando a chave K_A
$D_{K_A^{-1}}(x)$	deciframento assimétrico de x utilizando a chave K_A^{-1}
$MAC(x)$	código de autenticação de mensagens
$MAC_K(x)$	MAC utilizando chave K
$H(x)$	função de *hash* criptográfica
$Sig_A(x)$	assinatura digital da entidade A sobre x
$Ver_A(x)$	função de verificação de assinatura digital da entidade A
$A \rightarrow B : M$	A envia a mensagem M para B
N	um *nonce*
$A \mid\mid B$	concatenação de A com B

Tabela F.1: Notação utilizada.

Referências Bibliográficas

[1] James P. Anderson. Computer security technology planning study. Relatório Técnico ESD-TR-73-51, Air Force Electronic Systems Division, outubro de 1972.

[2] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm e M. Straβer. Mole 3.0: A Middleware for Java-Based Mobile Software Agents. Em Nigel Davies, Kerry Raymond e Jochen Seitz, editores, *Middleware '98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Proceedings*, pp. 355–372, The Lake District, Inglaterra, setembro de 1998. Springer.

[3] J. Baumann, F. Hohl, K. Rothermel e M. Straβer. Mole - Concepts of a Mobile Agent System. Relatório Técnico 1997/15, University of Stuttgart, Stuttgart, Alemanha, agosto de 1997.

[4] Paolo Bellavista, Antonio Corradi e Cesare Stefanelli. A Secure and Open Mobile Agent Programming Environment. Em *Proceedings of the 4^{th} International Symposium on Autonomous Decentralized Systems (ISADS '99)*, pp. 238–245, Tóquio, Japão, março de 1999. IEEE Computer Society Press.

[5] Andrzej Bieszczad, Bernard Pagurek e Tony White. Mobile Agents for Network Management. *IEEE Communications Surveys*, 1(1), Quarto trimestre de 1998.

[6] Antonio Carzaniga, Gian Pietro Picco e Giovanni Vigna. Designing Distributed Applications with Mobile Code Paradigms. Em *Proceedings of the 19^{th} International Conference on Software Engineering*, pp. 22–32, Boston, Massachusetts, E.U.A., maio de 1997. ACM Press.

[7] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris e Gene Tsudik. Itinerant Agents for Mobile Computing. IBM Research Report RC 20010, IBM Research Division T. J. Watson Research Center, março de 1995.

[8] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris e Gene Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5):34–49, outubro de 1995.

[9] David Chess, Colin Harrison e Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? IBM Research Report, IBM Research Division – T. J. Watson Research Center, dezembro de 1994.

[10] Christopher Colby, Peter Lee e George C. Necula. A Proof-Carrying Code Architecture for Java. Em E. Allen Emerson e A. Prasad Sistla, editores, *Computer Aided Verification, 12th International Conference, CAV 2000, Proceedings*, volume 1855 de *Lecture Notes in Computer Science*, pp. 557–560, Chicago, IL, E.U.A., julho de 2000. Springer.

[11] Antonio Corradi, Marco Cremonini, Rebecca Montanari e Cesare Stefanelli. Mobile Agents and Security: Protocols for Integrity. Em Lea Kutvonen, Hartmut König e Martti Tienari, editores, *Proceedings of the 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS '99)*, pp. 177–190, Helsinki, Finlândia, junho–julho de 1999. Kluwer Academic Publishers.

[12] Antonio Corradi, Rebecca Montanari e Cesare Stefanelli. Mobile Agents Protection in the Internet Environment. Em *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC '99)*, Phoenix, Arizona, E.U.A., outubro de 1999. IEEE Computer Society Press.

[13] Severino Collier Coutinho. *Números Inteiros e Criptografia RSA*. Série de Computação e Matemática. IMPA/SBM, 1997.

[14] Mary Dageforde. The Java™ Tutorial – Trail: Security in Java 2 SDK 1.2. Disponível em `http://java.sun.com/docs/books/tutorial/security1.2/index.html`.

[15] Nicodemos Damianou, Naranker Dulay, Emil Lupu e Morris Sloman. Ponder: A Language for Specifying Security and Management Policies for Distributed Systems – The Language Specification – Version 2.3. Imperial College Research Report DoC 2000/1, Department of Computing – Imperial College of Science, Technology and Medicine, London, Reino Unido, outubro de 2000.

[16] Oren Etzioni e Daniel S. Weld. Intelligent Agents on the Internet: Fact, Fiction, and Forecast. *IEEE Expert: Intelligent Systems and Their Application*, 10(4):44–49, agosto de 1995.

[17] William M. Farmer, Joshua D. Guttman e Vipin Swarup. Security for Mobile Agents: Authentication and State Appraisal. Em Elisa Bertino, Helmut Kurth, Giancarlo Martella e Emilio Montolivo, editores, *Computer Security - ESORICS 96,*

4th European Symposium on Research in Computer Security, Proceedings, volume 1146 de *Lecture Notes in Computer Science*, pp. 118–130, Roma, Itália, setembro de 1996. Springer.

[18] William M. Farmer, Joshua D. Guttman e Vipin Swarup. Security for Mobile Agents: Issues and Requirements. Em *Proceedings of the 19th National Information Systems Security Conference*, pp. 591–597, Baltimore, Maryland, E.U.A., outubro de 1996.

[19] Lucas C. Ferreira e Ricardo Dahab. Blinded-key signatures: securing private keys embedded in mobile agents. Em *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pp. 82–86, Madri, Espanha, março de 2002. SIGAPP, ACM Press.

[20] Lucas de Carvalho Ferreira e Ricardo Dahab. Blinded-Key Signatures: securing private keys embedded in mobile agents. Relatório Técnico IC-01-015, Instituto de Computação – Universidade Estadual de Campinas, Campinas, SP, Brasil, novembro de 2001.

[21] Stan Franklin e Art Graesser. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. Em Jörg P. Müller, Michael Wooldridge e Nicholas R. Jennings, editores, *Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Proceedings*, volume 1193, pp. 21–35, Budapest, Hungria, agosto de 1996. Springer.

[22] Stefan Fünfrocken. Protecting Mobile Web-Commerce Agents with Smartcards. Em *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents, ASA/MA '99, Proceedings*, pp. 90–102, Palm Springs, Califórnia, E.U.A., outubro de 1999. IEEE Computer Society Press.

[23] Li Gong. Java™ Platform Security Architecture. Disponível em `http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-spec.doc.html`.

[24] Li Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, novembro–dezembro de 1998.

[25] Li Gong, Marianne Mueller, Hemma Prafullchandra e Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2. Em *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pp. 103–112, Monterey, Califórnia, E.U.A., dezembro de 1997.

[26] Li Gong e Roland Schemers. Implementing Protection Domains in the Java™ Development Kit 1.2. Em *Proceedings of the Internet Society's Network and Distributed System Security Symposium (NDSS '98)*, pp. 125–134, San Diego, Califórnia, E.U.A., março de 1998. Internet Society.

[27] James Gosling, Bill Joy, Guy Steele e Gilad Bracha. *The Java™ Language Specification*. The Java™ Series. Addison-Wesley, 2ª edição, junho de 2000.

[28] Robert Gray, David Kotz, George Cybenko e Daniela Rus. Mobile agents: Motivations and state-of-the-art systems. Relatório Técnico TR2000-365, Thayer School of Engineering – Department of Computer Science – Dartmouth College, abril de 2000.

[29] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. Em *Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop*, pp. 9–23, Monterey, Califórnia, E.U.A., julho de 1996.

[30] Robert S. Gray. *Agent Tcl: A Flexible and Secure Mobile-Agent System*. Tese de Doutorado, Dartmouth College, Hanover, New Hampshire, E.U.A., junho de 1997. Disponível como Dartmouth Computer Science Technical Report TR98-327.

[31] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson e Daniela Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software–Practice and Experience*, 32(6):543–573, maio de 2002.

[32] Robert S. Gray, David Kotz, George Cybenko e Daniela Rus. D'Agents: Security in a Multiple-Language, Mobile-Agent System. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 154–187. Springer, 1998.

[33] Fritz Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 92–113. Springer, 1998.

[34] IKV++ GmbH, Berlim, Alemanha. *Grasshopper Basics and Concepts – Release 2.2*, março de 2001. Disponível em `http://grasshopper.ikv.de`.

[35] IKV++ GmbH, Berlim, Alemanha. *Grasshopper Programmer's Guide – Release 2.2*, março de 2001. Disponível em `http://grasshopper.ikv.de`.

[36] Wayne Jansen e Tom Karygiannis. Mobile Agent Security. Special Publication 800-19, Computer Security Division – National Institute of Standards and Technology, Gaithersburg, E.U.A., agosto de 1999.

[37] Wayne A. Jansen. Countermeasures for Mobile Agent Security. *Special Issue on Advanced Security Techniques for Network Protection, Elsevier Science BV*, novembro de 2000.

[38] Dag Johansen. Mobile Agent Applicability. Em Kurt Rothermel e Fritz Hohl, editores, *Mobile Agents, Second International Workshop, MA '98, Proceedings*, volume 1477 de *Lecture Notes in Computer Science*, pp. 80–98, Stuttgart, Alemanha, setembro de 1998. Springer.

[39] Dag Johansen, Fred B. Schneider e Robbert van Renesse. What TACOMA Taught Us. Em Dejan Milojicic, Frederick Douglis e Richard Wheeler, editores, *Mobility, Mobile Agents and Process Migration – An edited Collection*. Addison Wesley Publishing Company, 1998.

[40] Dag Johansen, Robbert van Renesse e Fred B. Schneider. An Introduction to the TACOMA Distributed System – Version 1.0. Relatório Técnico 95-23, University of Troms∅, junho de 1995.

[41] Dag Johansen, Robbert van Renesse e Fred B. Schneider. Operating System Support for Mobile Agents. Em *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pp. 42–45, Orcas Island, Washington, E.U.A., maio de 1995. IEEE Computer Society Press.

[42] David Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner, 1996.

[43] Günter Karjoth, N. Asokan e C. Gülcü. Protecting the Computation Results of Free-Roaming Agents. Em Kurt Rothermel e Fritz Hohl, editores, *Mobile Agents, Second International Workshop, MA '98, Proceedings*, volume 1477 de *Lecture Notes in Computer Science*, pp. 195–207, Stuttgart, Alemanha, setembro de 1998. Springer.

[44] Günter Karjoth, Danny B. Lange e Mitsuru Oshima. A Security Model for Aglets. *IEEE Internet Computing*, 1(4):68–77, julho–agosto de 1997.

[45] Neeran Karnik. *Security in Mobile Agent Systems*. Tese de Doutorado, Department of Computer Science and Engineering – University of Minnesota, Minneapolis, E.U.A., 1998. Disponível como Relatório Técnico 98-032.

[46] Neeran M. Karnik e Anand R. Tripathi. A Security Architecture for Mobile Agents in Ajanta. Em Mohamed G. Gouda, editor, *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp. 402–409, Taipei, Taiwan, abril de 2000. IEEE Computer Society, IEEE Computer Society Press.

[47] Neeran M. Karnik e Anand R. Tripathi. Security in the Ajanta Mobile Agent System. *Software—Practice and Experience*, 31(4):301–329, abril de 2001. John Wiley & Son, Ltd.

[48] Mitsubishi Electric ITA Horizon Systems Laboratory. Technology at a Glance – Concordia - Java Mobile Agent Technology. Disponível em `http://www.concordiaagents.com/Concordia-at-a-glance.pdf`.

[49] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin e Roland Schemers:. User Authentication and Authorization In the Java™ Platform. Em *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*, pp. 285–290, Phoenix, Arizona, E.U.A., dezembro de 1999. IEEE Computer Society Press.

[50] Danny B. Lange. Mobile Objects and Mobile Agents: The Future of Distributed Computing? Em Eric Jul, editor, *ECCOP'98 - Object-Oriented Programming, 12th European Conference, Proceedings*, volume 1445 de *Lecture Notes in Computer Science*, pp. 1–12, Bruxelas, Bélgica, julho de 1998. Springer.

[51] Danny B. Lange e Yariv Aridor. Agent Trasnfer Protocol – ATP/0.1. Draft 4, IBM Tokyo Research Laboratory, março de 1997. Disponível em `http://www.trl.ibm.com/aglets/atp/atp.htm`.

[52] Danny B. Lange e Mitsuru Oshima. Mobile Agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998. Baltzer Science Publishers.

[53] Danny B. Lange e Mitsuru Oshima. *Programming and Deploying Java™ Mobile Agents with Aglets™*. Addison Wesley, novembro de 1998.

[54] Danny B. Lange e Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM (CACM)*, 42(3):88–89, março de 1999.

[55] Sheng Liang e Gilad Bracha. Dynamic Class Loading in the Java™ Virtual Machine. Em *Proceedings of the 1998 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33 de *ACM SIGPLAN Notices*, pp. 36–44, Vancouver, British Columbia, Canada, outubro de 1998. ACM SIGPLAN, ACM Press, Addison Wesley.

[56] Tim Lindholm e Frank Yellim. *The Java™ Virtual Machine Specification*. The Java™ Series. Addison-Wesley, 2ª edição, 1999. Também disponível em `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[57] Antonio Liotta. *Towards Flexible and Scalable Distributed Monitoring with Mobile Agents*. Tese de Doutorado, University College London, julho de 2001.

[58] Antonio Liotta, George Pavlou e Graham Knight. Exploiting Agent Mobility for Large-Scale Network Monitoring. *IEEE Network*, 16(3):7–15, maio/junho de 2002.

[59] Gary McGraw e Ed Felten. *Securing JAVA – Getting Down to Business with Mobile Code*. John Wiley & Sons, 2ª edição, janeiro de 1999.

[60] Catherine Meadows. Detecting Attacks on Mobile Agents. Em *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, Califórnia, E.U.A., março de 1997.

[61] Alfred J. Menezes, Paul C. van Oorschot e Scott A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press Series on Discrete Mathematics and its Applications. CRC Press, agosto de 2001.

[62] Sun Microsystems, Inc. Java™ Security. Disponível em `http://java.sun.com/security/index.html`.

[63] Sun Microsystems, Inc. Java™ Authentication and Authorization Service (JAAS) Reference Guide for the Java™ 2 SDK, Standard Edition, v 1.4. Disponível em `http://java.sun.com/j2se/1.4/docs/guide/security/jaas/JAASRefGuide.html`, agosto de 2001.

[64] Sun Microsystems, Inc. Java™ Cryptography Architecture API Specification & Reference. Disponível em `http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html`, fevereiro de 2002.

[65] Sun Microsystems, Inc. Java™ Cryptography Extension (JCE) Reference Guide. Disponível em `http://java.sun.com/j2se/1.4/docs/guide/security/jce/JCERefGuide.html`, janeiro de 2002.

[66] Sun Microsystems, Inc. Java™ Security – Chronology of security-related bugs and issues, 3/19/02. Disponível em `http://java.sun.com/sfaq/chronology.html`, março de 2002.

[67] Dejan Milojicic. Mobile agent applications. *IEEE Concurrency*, 7(3):80–90, 1999.

[68] Mitsubishi Electric ITA Horizon Systems Laboratory. *Mobile Agent Computing*, janeiro de 1998. Disponível em `http://www.concordiaagents.com/MobileAgentsWhitePaper.pdf`.

[69] R. Moats. RFC 2141: URN Syntax. Disponível em `ftp://ftp.isi.edu/in-notes/rfc2141.txt`, maio de 1997.

[70] George C. Necula. Proof-Carrying Code. Em *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium*, pp. 106–119, Paris, França, janeiro de 1997. ACM SIGPLAN e SIGACT, ACM Press.

[71] George C. Necula e Peter Lee. Proof-Carrying Code. Relatório Técnico CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, novembro de 1996.

[72] George C. Necula e Peter Lee. Safe, Untrusted Agents Using Proof-Carrying Code. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 61–91. Springer, 1998.

[73] Scott Oaks. *Java Security*. O'Reilly & Associates, 2ª edição, maio de 2001.

[74] Object Management Group, Inc. (OMG). *Mobile Agent Facility Specification*, janeiro de 2000. Disponível em `http://www.omg.org/cgi-bin/doc?formal/00-01-02.pdf`.

[75] Joann J. Ordille. When agents roam, who can you trust? Em *1st Conference on Emerging Technologies and Applications in Communications (etaCOM '96)*, Portland, Oregon, E.U.A., maio de 1996.

[76] Mitsuru Oshima, Guenter Karjoth e Kouichi Ono. Aglets Specification 1.1 Draft. Draft 0.65, IBM Tokyo Research Laboratory, setembro de 1998. Disponível em `http://www.trl.ibm.com/aglets/spec11.html`.

[77] John K. Ousterhout, Jacob Y. Levy e Brent B. Welch. The Safe-Tcl Security Model. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 217–234. Springer, 1998.

[78] Holger Peine. Ara - Agents for Remote Action. Em William R. Cockayne e Michael Zyda, editores, *Mobile Agents: Explanations and Examples*, pp. 96–164. Manning Publications Co., 1997.

[79] Holger Peine. Security Concepts and Implementation in the Ara Mobile Agent System. Em *Proceedings of the 7th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '98)*, pp. 236–242, Palo Alto, Califórnia, E.U.A., junho de 1998. IEEE Computer Society and CERC, IEEE Computer Society Press.

[80] Holger Peine e Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. Em Kurt Rothermel e Radu Popescu-Zeletin, editores, *Mobile Agents, First International Workshop, MA '97, Proceedings*, volume 1219 de *Lecture Notes in Computer Science*, pp. 50–61, Berlin, Alemanha, abril de 1997. Springer.

[81] Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. Tese de Doutorado, Politecnico di Torino – Dipartimento di Automatica e Informatica, 1998.

[82] James Riordan e Bruce Schneier. Environmental Key Generation Towards Clueless Agents. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 15–24. Springer, 1998.

[83] Volker Roth. Secure Recording of Itineraries through Co-operating Agents. Em Serge Demeyer e Jan Bosch, editores, *Object-Oriented Technology, ECOOP'98 Workshop Reader, ECOOP'98 Workshops, Demos, and Posters, Proceedings*, volume 1543 de *Lecture Notes in Computer Science*, pp. 297–298, Bruxelas, Bélgica, julho de 1998. Springer.

[84] Volker Roth. Mutual Protection of Co-operating Agents. Em Jan Vitek e Christian D. Jensen, editores, *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 de *Lecture Notes in Computer Science*, pp. 275–285. Springer, 1999.

[85] Volker Roth. On the Robustness of Some Cryptographic Protocols for Mobile Agent Protection. Em Gian Pietro Picco, editor, *Mobile Agents, 5^{th} International Conference, MA 2001, Proceedings*, volume 2240 de *Lecture Notes in Computer Science*, pp. 1–14, Atlanta, GA, E.U.A., dezembro de 2001. Springer.

[86] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, junho de 2002.

[87] Tomas Sander e Christian F. Tschudin. Towards Mobile Cryptography. Relatório Técnico TR-97-049, International Computer Science Institute, novembro de 1997.

[88] Tomas Sander e Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 44–60. Springer, 1998.

[89] Vijay Saraswat. Java is not type-safe. Disponível em `http://www.research.att.com/~vj/bug.html`, agosto de 1997.

[90] Fred B. Schneider. Towards Fault-tolerant and Secure Agentry. Em M. Mavronicolas e P. Tsigas, editores, *11th International Workshop, WDAG '97, Proceedings*, volume 1320 de *Lecture Notes in Computer Science*, pp. 1–14, Saarbrücken, Alemanha, setembro de 1997. Springer.

[91] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, 2ª edição, outubro de 1995.

[92] Simon Singh. *The Code Book – The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography*. Doubleday, 1999.

[93] Karen Sollins e Larry Masinter. RFC 1737: Functional Requirements for Uniform Resource Names. Disponível em `ftp://ftp.isi.edu/in-notes/rfc1737.txt`, dezembro de 1994.

[94] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2ª edição, junho de 1998.

[95] Douglas R. Stinson. *Cryptography: Theory and Practice*. The CRC Press Series on Discrete Mathematics and its Applications. CRC Press, março de 1995.

[96] Markus Straßer, Joachim Baumann e Fritz Hohl. Mole - A Java Based Mobile Agent System. Em M. Muehlhaeuser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP '96)*, pp. 327–334. dpunkt.verlag, Linz, Áustria, julho de 1996.

[97] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Allan R. Ditzel, Gregory A. Hill, Brian R. Pouliot e David S. Smith. NOMADS: Toward an Environment for Strong and Safe Agent Mobility. Em Carles Sierra, Maria Gini e Jeffrey S. Rosenschein, editores, *Proceedings of the Fourth International Conference on Autonomous Agents 2000*, pp. 163–164, Barcelona, Catalonia, Espanha, junho de 2000. ACM Press.

[98] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill e Renia Jeffers. Strong Mobility and Fine-Grained Resource Control in NOMADS. Em David Kotz e Friedemann Mattern, editores, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Proceedings*, volume 1882 de *Lecture Notes in Computer Science*, pp. 2–15, Zurich, Suíça, setembro de 2000. ETH Zurich, Springer.

[99] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers e Timothy S. Mitrovich. An Overview of the NOMADS Mobile Agent System. Em *6th ECOOP WORKSHOP ON MOBILE OBJECT SYSTEMS: Operating System Support, Security and Programming Languages*, Sophia Antipolis, França, junho de 2000. Disponível em `http://cuiwww.unige.ch/~ecoopws/ws00/papers/nomads.pdf`.

[100] Joseph Tardo e Luis Valente. Mobile Agent Security and Telescript. Em *Proceedings of the 41st IEEE International Computer Conference (COMPCON Spring '96)*, pp. 58–63, Santa Clara, Califórnia, E.U.A., fevereiro de 1996. IEEE Computer Society Press.

[101] Anand Tripathi, Tanvir Ahmed, Sumedh Pathak, Abhijit Pathak, Megan Carney, Murlidhar Koka e Paul Dokas. Active Monitoring of Network Systems using Mobile Agents. Em Benny Bing e Pascal Lorenz, editores, *NETWORKS The Proceedings of the Joint International Conference on Wireless LANs and Home Networks (ICWLHN 2002) and Networking (ICN 2002)*, Atlanta, Geórgia, E.U.A., agosto de 2002. IEEE Communications Society, World Scientific.

[102] Anand R. Tripathi e Neeran M. Karnik. Delegation of Priviliges to Mobile Agents in Ajanta. Em Peter Graham e Muthucumaru Maheswaran, editores, *Proceedings of the First International Conference on Internet Computing, IC '2000*, pp. 379–386, Las Vegas, Nevada, E.U.A., junho de 2000. CSREA Press.

[103] Anand R. Tripathi, Neeran M. Karnik, Manish K. Vora, Tanvir Ahmed e Ram D. Singh. Mobile Agent Programming in Ajanta. Em Mohamed G. Gouda, editor, *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pp. 190–197, Austin, Texas, E.U.A., maio de 1999. IEEE Computer Society Press.

[104] Christian F. Tschudin. Mobile Agent Security. Em Matthias Klusch, editor, *Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet*, capítulo 18, pp. 431–445. Springer, 1999.

[105] Nelson Uto e Ricardo Dahab. *Survey* sobre Segurança de Sistemas de Agentes Móveis. Relatório Técnico IC-01-008, Instituto de Computação – Universidade Estadual de Campinas, Campinas, SP, Brasil, julho de 2001.

[106] Nelson Uto e Ricardo Dahab. Segurança de Sistemas de Agentes Móveis. Em Clovis Torres Fernandes, editor, *Anais do 3º Simpósio Segurança em Informática (SSI '2001)*, pp. 211–220, São José dos Campos, SP, Brasil, outubro de 2001.

[107] Bill Venners. Solve real problems with aglets, a type of mobile agent. Disponível em `http://www.javaworld.com/javaworld/jw-05-1997/jw-05-hood\_p.html`, maio de 1997.

[108] Bill Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill Professional Publishing, 2ª edição, dezembro de 1999.

[109] Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications*. Tese de Doutorado, Politecnico di Milano, 1997.

[110] Giovanni Vigna. Protecting Mobile Agents through Tracing. Em *3rd ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems (MOS '97)*, Jyväskylä, Finlândia, junho de 1997.

[111] Giovanni Vigna. Cryptographic Traces for Mobile Agents. Em Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pp. 137–153. Springer, 1998.

[112] Robert Wahbe, Steven Lucco, Thomas E. Anderson e Susan L. Graham. Efficient Software-Based Fault Isolation. Em *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, volume 27 de *Operating Systems Review*, pp. 203–216, Asheville, Carolina do Norte, E.U.A., dezembro de 1993. ACM Press.

[113] Tom Walsh, Noemi Paciorek e David Wong. Security and Reliability in Concordia™. Em Hesham El-Rewini, editor, *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS '98)*, volume 7: Software Technology Track, pp. 44–53, Kohala Coast, E.U.A., janeiro de 1998. IEEE Computer Society Press.

[114] Jim White. Mobile Agents White Paper. Relatório técnico, General Magic Inc., 1996.

[115] Uwe G. Wilhelm, Sebastian Staamann e Levente Buttyán. On the Problem of Trust in Mobile Agent Systems. Em *Proceedings of the Internet Society's Network and Distributed System Security Symposium (NDSS '98)*, San Diego, Califórnia, E.U.A., março de 1998. Internet Society.

[116] Uwe Georg Wilhelm. *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-resistant Hardware*. Tese de Doutorado, École Polytechnique Fédérale de Lausanne, 1999.

[117] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young e Bill Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. Em Kurt Rothermel e Radu Popescu-Zeletin, editores, *Mobile Agents, First International Workshop, MA '97, Proceedings*, volume 1219 de *Lecture Notes in Computer Science*, pp. 86–97, Berlin, Alemanha, abril de 1997. Springer.

[118] Bennet S. Yee. A Sanctuary for Mobile Agents. Relatório Técnico CS97-537, University of California at San Diego, La Jolla, CA, abril de 1997.

[119] Bennet S. Yee. A Sanctuary for Mobile Agents. Em Jan Vitek e Christian D. Jensen, editores, *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 de *Lecture Notes in Computer Science*, pp. 261–273. Springer, 1999.

[120] Adam Young e Moti Yung. Sliding Encryption: A Cryptographic Tool for Mobile Agents. Em Eli Biham, editor, *Fast Software Encryption, 4^{th} International Workshop, FSE '97, Proceedings*, volume 1267 de *Lecture Notes in Computer Science*, Haifa, Israel, janeiro de 1997. Springer.