UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ComFIRM — Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional

por

FÁBIO OLIVÉ LEITE

Dissertação submetida a avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof[a]. Dr[a]. Taisy Silva Weber
Orientador

Porto Alegre, dezembro de 2000.

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Leite, Fábio Olivé

ComFIRM — Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional / por Fábio Olivé Leite. — Porto Alegre: PPGC da UFRGS, 2000.

117 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2000. Orientador: Weber, Taisy Silva.

1. Injeção de falhas. 2. Validação experimental. 3. Comunicação confiável. 4. Tolerância a falhas. I. Weber, Taisy Silva. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof^a^. Wrana Maria Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Philippe Olivier Alexandre Navaux
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

*Dedico este trabalho à maravilhosa
criança que há quatro meses se
desenvolve no ventre de minha esposa.
Pedacinho de amor tornado carne;
ainda nem nasceu, mas quanta
alegria já trouxe!*

Agradecimentos

Muitas pessoas participaram na elaboração deste trabalho, e muitas outras o influenciaram de várias maneiras, portanto não poderia deixar de manifestar aqui minha profunda gratidão por todas elas. Agradeço em especial:

- Aos meus pais, pelas raízes e pelas asas. A eles devo a universalidade, como já me foi dito, de minha educação; raiz sólida com a qual me sinto pronto a enfrentar o mundo. A eles devo as asas da minha imaginação, com a qual ouso tentar voar cada vez mais alto.
- À minha esposa Renata, pelo amor, carinho e dedicação. Ponto de equilíbrio da minha vida, me mostra o lado humano quando insisto demais no técnico, me lembra do lado técnico quando fico "humano" demais. Fortaleza delicada, suavemente linda em sua detalhada forma, me faz transbordar de amor com apenas um olhar.
- À minha orientadora, Taisy, pelos ensinamentos, conselhos, avisos e puxões de orelha, sempre merecidos. Capturou e filtrou minhas explosões de idéias, indicou o caminho, e orquestrou este trabalho.
- Ao meu irmão Rafael, amigo eterno mesmo quando distante, pelo apoio, incentivo e pelo tempo agradável em que dividimos um apartamento durante a elaboração deste trabalho.
- Ao amigo Luis Claudio, irmão que a vida me trouxe, companheiro de todas as horas e de todas as empreitadas. Agora é a minha vez de cobrar a dissertação!
- Aos amigos da UCPel, pelos anos de convívio agradável, frutífero e sempre fraterno. Marilton, Luis Fernando, Ana Paula, Rafael Accorsi e ainda outros que a cabeça esquece mas o coração não. É ótimo tê-los como amigos!

Agradeço também aos professores Jorge Barbosa e Adenauer Yamin, pelos quais conheci a área de Software Básico, que tanto me fascina. Não posso deixar de agradecer também à Conectiva, empresa na qual trabalho, por propiciar um excelente ambiente de trabalho e incentivar a pesquisa e a participação em eventos acadêmicos.

Sumário

Lista de Figuras 7

Lista de Tabelas 9

Resumo 10

Abstract 11

1 Introdução 12
1.1 Motivação 12
1.2 Objetivos e resultados esperados 13
1.3 Trabalhos relacionados 14
1.4 Organização do texto 15

2 Validação através da Injeção de Falhas 17
2.1 A necessidade de validação 17
2.2 Introdução à Injeção de Falhas 17
2.2.1 Injeção de Falhas por Hardware 18
2.2.2 Injeção de Falhas por Simulação 18
2.2.3 Injeção de Falhas por Software 19
2.3 Injeção de Falhas de Comunicação 20
2.3.1 Seleção de Mensagens 20
2.3.2 Manipulação de Mensagens 22
2.3.3 A Ferramenta ORCHESTRA 23
2.4 Modelos de Falha 23
2.5 Especificação de experimentos de Injeção de Falhas 25
2.6 Conclusão 26

3 Implementação de Injetores de Falhas em Software . . 27
3.1 Injeção de Falhas no nível da aplicação 27
3.1.1 Injeção no código da aplicação 27
3.1.2 Injeção por processos concorrentes à aplicação 28
3.1.3 Injeção em código utilizado pela aplicação 29
3.1.4 Injeção no meta-nível 30
3.2 Injeção de Falhas no nível do Sistema Operacional 30
3.2.1 Influências sobre a implementação no Sistema Operacional 31
3.2.2 Inserção de injetores de falhas de comunicação 32
3.3 Conclusão 34

4 O Sistema Operacional Linux 35
4.1 Introdução 35
4.2 Arquitetura conceitual 36
4.2.1 Visão geral de um sistema Linux 36
4.2.2 Visão geral da estrutura do *kernel* 37
4.2.3 Arquitetura dos subsistemas 39
4.3 Arquitetura Concreta 42
4.3.1 O sistema de arquivos `proc` 42

4.3.2 Implementação do Subsistema de Rede 45
4.4 Programando em modo *kernel* 51
4.5 Mecanismos do *kernel* 52
4.6 Conclusão 54

5 A ferramenta ComFIRM 55
5.1 Considerações iniciais 55
5.2 Recursos da ferramenta ComFIRM 56
5.3 Instruções e regras 59
5.3.1 Codificação das Instruções 59
5.3.2 Formatação das regras 62
5.4 Utilização da ferramenta 63
5.5 A interface ComFIRM's Face 65
5.6 Conclusão 67

6 Testes e experimentos 69
6.1 Testes simples de funcionamento 69
6.1.1 Primeiro teste 69
6.1.2 Segundo teste 72
6.1.3 Terceiro teste 74
6.2 Experimentos 77
6.2.1 Primeiro experimento 77
6.2.2 Segundo experimento 79
6.3 Conclusão 82

7 Conclusões 83
7.1 Resultados 83
7.2 Experiência adquirida 84
7.3 Dificuldades encontradas 86
7.4 Trabalhos futuros 87

Anexo 1 Código da ferramenta ComFIRM 89
A.1 net/core/Makefile 89
A.2 include/linux/proc_fs.h 89
A.3 kernel/sched.c 89
A.4 net/core/dev.c 90
A.5 include/linux/comfirm.h 91
A.6 net/core/comfirm.c 92

Anexo 2 Código da interface ComFIRM's Face 108

Bibliografia 115

Lista de Figuras

FIGURA 2.1 – Seleção baseada em conteúdo 21
FIGURA 2.2 – Seleção baseada em fluxo 21
FIGURA 2.3 – Seleção baseada em elementos externos 21
FIGURA 2.4 – Tipos de ações sobre mensagens 22

FIGURA 3.1 – Diferentes implementações de uma pilha TCP/IP 32
FIGURA 3.2 – Inserção de uma ferramenta em um *microkernel* 33
FIGURA 3.3 – Inserção de uma ferramenta em implementações monolíticas . 33

FIGURA 4.1 – Decomposição de um Sistema Linux em camadas principais . . 37
FIGURA 4.2 – Decomposição do *kernel* de um sistema Linux 38
FIGURA 4.3 – Interação do VFS com o restante do sistema 40
FIGURA 4.4 – Relação do sistema de arquivos `proc` com o restante do *kernel* 40
FIGURA 4.5 – Disposição em camadas do subsistema de rede 41
FIGURA 4.6 – Organização em camadas dos protocolos de comunicação suportados 46
FIGURA 4.7 – Estrutura de um *socket buffer* 48

FIGURA 5.1 – Teste do registro da ferramenta ComFIRM 56
FIGURA 5.2 – Enviando comandos à ferramenta ComFIRM 57
FIGURA 5.3 – Configurando regras de recepção 64
FIGURA 5.4 – Interface gráfica experimental da ferramenta ComFIRM 66

FIGURA 6.1 – Configuração da ComFIRM's Face para o primeiro teste . . . 70
FIGURA 6.2 – Transmissão dos pacotes do primeiro teste 70
FIGURA 6.3 – Recepção dos pacotes do primeiro teste 71
FIGURA 6.4 – Registro da ferramenta durante o primeiro teste 71
FIGURA 6.5 – Recepção dos pacotes na variação do primeiro teste 72
FIGURA 6.6 – Configuração da ComFIRM's Face para o segundo teste 73
FIGURA 6.7 – Recepção dos pacotes na primeira rodada do segundo teste . . 73
FIGURA 6.8 – Recepção dos pacotes na segunda rodada do segundo teste . . 74
FIGURA 6.9 – Recepção dos pacotes na terceira rodada do segundo teste . . 74
FIGURA 6.10 – Programação e ativação da ferramenta ComFIRM para o terceiro teste 76
FIGURA 6.11 – Envio dos pacotes no terceiro teste 76
FIGURA 6.12 – Recepção dos pacotes no terceiro teste 76
FIGURA 6.13 – Registro da ferramenta no terceiro teste 77
FIGURA 6.14 – Configuração da ComFIRM's Face para o primeiro experimento 78
FIGURA 6.15 – Registro do Cluster Manager Heartbeat durante o primeiro experimento 79
FIGURA 6.16 – Registro da ferramenta ComFIRM durante o primeiro experimento 80
FIGURA 6.17 – Configuração da ComFIRM's Face para o segundo experimento 81
FIGURA 6.18 – Registro do Cluster Manager Heartbeat durante o segundo experimento 81

FIGURA 6.19 – Registro da ferramenta ComFIRM durante o segundo experimento 82

Lista de Tabelas

TABELA 5.1 – Instruções de seleção da ferramenta ComFIRM 57
TABELA 5.2 – Modificadores para instruções de seleção 57
TABELA 5.3 – Instruções de manipulação da ferramenta ComFIRM . . . 58
TABELA 5.4 – Modificadores para instruções de manipulação 58

Resumo

Este trabalho trata da técnica de validação experimental de protocolos de comunicação confiável, através da injeção de falhas de comunicação. São estudadas inicialmente as técnicas de injeção de falhas, por hardware, software e simulação, e então são aprofundados os conceitos de injeção de falhas de comunicação, modelos de falha e especificação de experimentos de injeção de falhas.

Em um segundo momento, são estudadas as formas de implementação de injetores de falhas em software, em suas duas formas mais comuns: no nível da aplicação e no nível do sistema operacional. São comentados os impactos da implementação de injetores no código da aplicação, por processos concorrentes à aplicação, em código utilizado pela aplicação e no meta-nível. Por fim, são estudados também que influências sofre a implementação de um injetor de falhas em um sistema operacional, e mais especificamente a de injetores de falhas de comunicação.

O objetivo específico deste trabalho é implementar um injetor de falhas de comunicação bastante abrangente e flexível, situado dentro do núcleo do Sistema Operacional Linux. Para viabilizar esta implementação foi estudada também a arquitetura do Sistema Operacional Linux, sua decomposição em subsistemas e a interação entre estes. Foram estudadas também as várias técnicas de programação e mecanismos que o Sistema Operacional Linux fornece aos seus subsistemas.

Estando completas a revisão bibliográfica a respeito de injeção de falhas e o estudo do código do Sistema Operacional Linux, são apresentadas a proposta e a implementação da ferramenta ComFIRM — Communication Fault Injection through Operating System Resource Modification, suas características e sua inserção dentro do núcleo do Sistema Operacional Linux.

Finalizando este trabalho, são apresentados uma pequena série de testes de funcionamento e experimentos realizados com a ferramenta ComFIRM, visando demonstrar a correção de seu funcionamento, o cumprimento de seus objetivos e também sua praticidade e flexibilidade de uso. São apresentadas as conclusões deste trabalho, propostas de melhorias à ferramenta apresentada, bem como possibilidades de trabalhos futuros.

Palavras-chave: Injeção de falhas, validação experimental, comunicação confiável, tolerância a falhas.

TITLE: "COMFIRM — COMMUNICATION FAULT INJECTION THROUGH OPERATING SYSTEM RESOURCE MODIFICATION"

Abstract

This work presents the experimental validation technique of reliable communication protocols, through the use of communication fault injection. Initially are studied the techniques of fault injection through hardware, software and simulation, and then the concepts of communication fault injection, fault models and fault injection experiment specification are discussed in more detail.

Further on, the present work studies the possibilities for the implementation of software fault injectors, in its two most common forms: in the application and operating system levels. Comments are made about the impacts of injectors implementation in application code, by processes that run concurrently with the application, in code used by the application, and also in the meta-level. To wrap the subject, are also studied what are the influences over the implementation of a fault injector in the operating system level, and more specificaly the implementation of communication fault injectors.

The specific goal of the present work is to implement a very comprehensive and flexible communication fault injector, placed inside the kernel of the Linux Operating System. In order to make this implementation viable, were also studied the Linux Operating System architecture, its decomposition in subsystems and the interaction among them. After that the various programming techniques and mechanisms that the Linux kernel offers to its subsystems were studied.

Having completed the bibliographic review on fault injection and the study of the Linux kernel code, this work then presents the proposal and implementation of the tool called ComFIRM — Communication Fault Injection through Operating System Resource Modification, its characteristics and the way it is inserted into the kernel of the Linux Operating System.

To end this work, are presented a small series of functional tests and experiments done with the ComFIRM tool, in order to demonstrate the correctness of its functioning, the accomplishment of its goals and the practicity and flexibility of its use. Also presented are the conclusions of this work, proposals of enhancements to the tool, as well as possibilities of future works.

Keywords: Fault Injection, Experimental Validation, Reliable Communication, Fault Tolerance.

1 Introdução

Os sistemas distribuídos estão cada vez mais inseridos na sociedade atual, onde a cada dia aumenta o número de recursos vitais controlados por redes de computadores. Um dos pontos fundamentais dos sistemas distribuídos é a comunicação entre nodos, sejam nodos físicos (máquinas) ou nodos lógicos (processos), portanto um dos principais campos de pesquisa da área de Tolerância a Falhas é a comunicação confiável.

Diversos protocolos já foram criados para atribuir algumas propriedades à comunicação, tais como: confiabilidade, atomicidade e ordenação. Estes protocolos são criados e comprovados segundo modelos de falhas por vezes bastante restritos, o que de certa forma limita sua aplicação. Ainda mais, é difícil de se comprovar na prática seu funcionamento, visto que a ocorrência de falhas reais não é determinística nem previsível.

Ao invés de se executar os protocolos e esperar pela ocorrência espontânea de falhas, uma técnica bastante popular hoje em dia para validar o funcionamento destes protocolos é a injeção de falhas. Ferramentas de injeção de falhas permitem que se aplique falhas específicas sobre um protocolo ou processo em funcionamento, facilitando a comprovação de que o protocolo ou o processo realmente tolera o modelo de falhas que se propôs a suportar [HSU 97].

Além da aplicação de falhas, ferramentas de injeção de falhas também permitem a monitoração do protocolo em estudo, de forma a possibilitar o acompanhamento dos acontecimentos subseqüentes à aplicação das falhas. Desta forma se pode comprovar de forma prática que as implementações realmente apresentam o nível de confiabilidade pretendido, ou demonstrar problemas que necessitem correção, tanto na especificação quanto na implementação [DAW 96a].

O grupo de pesquisa em Tolerância a Falhas do Instituto de Informática da UFRGS vem já há algum tempo realizando trabalhos na área de Tolerância a Falhas em Sistemas Distribuídos, tendo já produzido alguns trabalhos na área de Injeção de Falhas. Como exemplos podem ser citados AFIDS [SOT 97], FIX [TEI 94] e ADC [BAR 96], bem como “Injeção de Falhas de Comunicação no Sistema Operacional Linux” [LEI 99], que pode ser considerado o prólogo a este trabalho.

1.1 Motivação

Embora abordagens como AFIDS [SOT 96] sejam interessantes por criarem um framework reutilizável, multiplataforma e facilmente extensível, a existência de ferramentas que funcionem intimamente ligadas ao núcleo do sistema operacional também possuem várias características interessantes. Talvez a principal delas seja a de não necessitar a recompilação ou mesmo o acesso ao código fonte do processo ou protocolo em estudo, visto que as falhas serão injetadas em um nível de abstração mais baixo. Algumas organizações de teste por vezes necessitam de metodologias que não instrumentem o código em teste [DAW 96b].

Isto é particularmente importante para testar sistemas já em uso, ou para os quais o código fonte não está disponível. Como o sistema operacional provê uma máquina virtual ao processo, o qual não conhece o “mundo exterior” senão pelas chamadas de sistema que o sistema operacional lhe proporciona, alterações no

funcionamento interno desta máquina virtual são completamente transparentes ao processo, e falhas injetadas em serviços prestados pelo *kernel* são, do ponto de vista do processo, indistinguíveis de falhas reais provenientes do universo físico.

Com esta possibilidade, fica fácil perceber que através da inserção de uma ferramenta de injeção de falhas de comunicação no núcleo de um sistema operacional se obtém um sistema capaz de controlar diretamente as trocas de mensagens entre aplicações distribuídas. Este sistema poderá então ser utilizado para monitorar as implementações de protocolos de comunicação confiável, inserindo falhas e verificando a correção de sua execução.

Tal ferramenta deve possibilitar a especificação de experimentos de forma flexível e sua modificação em tempo de execução, permitindo ao condutor dos testes ajustar as regras que descrevem o experimento à medida que casos mais específicos são atingidos. Através da monitoração de protocolos de comunicação confiável se pode compreender com mais clareza seu funcionamento, bem como perceber modificações que possam elevar a sua confiabilidade ou ampliar o modelo de falhas suportado.

Embora a técnica de injeção de falhas para validação de mecanismos de tolerância a falhas não seja nova, existem poucos trabalhos que visem *especificamente* a injeção de falhas de comunicação, e os vários trabalhos já realizados na área de injeção de falhas em geral normalmente deixam em segundo plano as falhas de comunicação. Mesmo a bibliografia básica da área de Tolerância a Falhas não traz muita luz à esta técnica [PRA 96].

1.2 Objetivos e resultados esperados

Este trabalho visa estudar modelos de falhas úteis à validação de protocolos de comunicação confiável, modos de implementar estes modelos, formas de especificação e condução de experimentos de injeção de falhas de comunicação e as abordagens de implementação de ferramentas de injeção de falhas de comunicação. Também serão estudadas as formas de inserção de ferramentas de injeção de falhas em sistemas operacionais.

Espera-se com isto desenvolver um trabalho que explore a técnica de validação de protocolos de comunicação confiável através da injeção de falhas de comunicação, gerando experiência e domínio na área e criando uma ferramenta que permita um maior entendimento dos mecanismos utilizados na comunicação tolerante a falhas em sistemas distribuídos.

Como resultado específico pretende-se projetar e implementar uma ferramenta de injeção de falhas de comunicação em nível de sistema operacional, que seja flexível quanto aos tipos de falhas suportados, que permita a especificação de experimentos através de regras simples e com possibilidade de alteração em tempo de execução. A ferramenta será integrada ao Sistema Operacional Linux e eventualmente pretende-se que esta seja portável ao Linux-RT.

As falhas a serem injetadas serão criadas através de primitivas simples percebidas na construção de modelos de falhas. Através da seleção de mensagens por meio de testes de seu conteúdo ou de recursos da própria ferramenta, poderá ser selecionado o protocolo objeto do experimento. As mensagens selecionadas sofrerão a injeção de falhas usando outras primitivas, desta vez de alteração da mensagem,

de alteração no processo de envio da mensagem (duplicação, descarte) ou mesmo de alteração de recursos da ferramenta (ação sobre contadores, disparo de temporizadores).

Com a ferramenta pronta serão gerados testes de comprovação de seu funcionamento, bem como testes que comprovem sua aplicabilidade na validação da implementação de protocolos de comunicação confiável.

1.3 Trabalhos relacionados

Muitos trabalhos *relacionados* à injeção de falhas de comunicação já foram apresentados à comunidade, com diferentes propósitos. Esta técnica de validação experimental é interessante por ser utilizada de forma quase idêntica, porém com objetivos levemente diferentes, por duas áreas de pesquisa: a Tolerância a Falhas e a área de Redes de Computadores.

Na Tolerância a Falhas se utilizam ferramentas de injeção de falhas de comunicação para validar protocolos de comunicação confiável, através de perdas, atrasos e duplicações de pacotes. Já na área de Redes de Computadores, normalmente são usadas ferramentas que modelam um canal de comunicação de forma a apresentar características especiais de latência, largura de banda e perda de mensagens, por exemplo, utilizando técnicas bastante parecidas.

Da área de Redes de Computadores podemos citar a RFC 2398 [PAR 98], por exemplo, que traz uma lista de ferramentas sugeridas a implementadores de pilhas TCP/IP. São vários tipos de ferramentas, que vão desde a geração e captura de pacotes, até a geração de tráfego intenso e modelagem de canais de comunicação, através de atrasos e perdas de mensagens. Algumas destas são citadas abaixo.

Dummynet: é uma ferramenta que simula a existência de filas de tamanho finito, limitações de largura de banda e atrasos de comunicação. Se insere entre os protocolos TCP e IP de sistemas baseados no BSD Unix, e permite reconfiguração de parâmetros dinamicamente;

NIST Net: esta ferramenta é um "simulador de redes"; funciona transformando um sistema Linux em um roteador "seletivamente ruim". Pacotes podem ser atrasados, perdidos, duplicados, ter sua largura de banda reduzida, etc;[1]

Packet Shell: baseado na linguagem TCL, esta ferramenta é na verdade uma biblioteca de funções, que permite a geração, modificação, envio e recebimento manual de pacotes de rede;

tcpanaly: esta é uma ferramenta que inspeciona uma implementação do protocolo TCP através da análise de capturas de pacotes;

Ttcp: este é um gerador de tráfego, que também funciona como absorvedor de tráfego, e que pode gerar padrões de dados, alinhamentos de páginas e etc.

O propósito destas ferramentas é claro, pois estão sempre ligadas à características de tráfego ou à análise de implementações de protocolos de comunicação

[1] O autor desta dissertação tentou entrar em contato com o autor da ferramenta NIST Net, para um possível intercâmbio de informações, porém infelizmente não obteve resposta.

em rede, como o TCP. Os objetivos de ferramentas de injeção de falhas de comunicação normalmente são mais específicos, onde não somente o tráfego será influenciado, mas pacotes cuidadosamente selecionados serão manipulados, de forma a levar implementações de protocolos de comunicação confiável a situações difíceis de serem alcançadas normalmente.

Ferramentas de injeção de falhas que atuem em memória, registradores e barramentos já foram bastante estudadas, e existem vários artigos sobre elas. Entretanto, ferramentas de injeção de falhas de comunicação não são tão comuns, sendo as mais conhecidas: PFI [DAW 95], CSFI [CAR 95a] e ORCHESTRA [DAW 96b]. Esta última é a que apresenta a maior quantidade de informação disponível, e é a principal fonte de inspiração para este trabalho.

1.4 Organização do texto

O texto desta dissertação está organizado de forma a apresentar o assunto estudado com profundidade crescente, porém deixando de lado os conceitos mais básicos da Tolerância a Falhas, visto que seu público alvo já os deve conhecer. Para tal, no capítulo 2, Validação através da Injeçao de Falhas, serão brevemente revisadas as técnicas de injeção de falhas (simulação, hardware e software) e de injeção de falhas de comunicação; serão expostos modelos de falhas comumente encontrados e as formas de se especificar experimentos de injeção de falhas.

Já o capítulo 3, Implementação de Injetores de Falhas em Software, desce um pouco mais o nível de abstração, tratando das formas de se implementar injetores de falhas em software e seus impactos no ambiente de execução em que estão inseridos. Aqui será também discutido como inserir injetores de falhas em diferentes arquiteturas de sistemas operacionais.

Como este trabalho objetiva a implementação de um injetor dentro do núcleo do Sistema Operacional Linux, faz-se necessário expor alguns de seus mecanismos de funcionamento, de forma a facilitar a compreensão do código gerado. O capítulo 4, O Sistema Operacional Linux, documenta todos os mecanismos do Linux que foram utilizados na implementação da ferramenta, como os arquivos virtuais, a temporização, o subsistema de rede (suas funções e estruturas) e os problemas encontrados quando se está programando um *kernel*.

Tendo já revisto o conhecimento teórico necessário à compreensão da ferramenta, pode-se apresentar sua proposta e implementação. O capítulo 5, A Ferramenta ComFIRM, é o capítulo onde serão examinados o código adicionado ao núcleo e as formas com que este código interage com o resto do sistema operacional. É apresentada também uma interface gráfica simples, cujo principal propósito é demonstrar que a ferramenta pode ser utilizada de forma amigável. Apenas as partes do código que são interessantes serão incluídas no texto do capítulo. Os códigos completos, tanto da ferramenta quanto da interface, estão disponíveis respectivamente nos apêndices A e B.

O capítulo 6, Testes e Experimentos, demonstra com testes práticos a funcionalidade da ferramenta. Inicialmente testes simples, que apenas demonstram a injeção de falhas de comunicação, e então testes com ferramentas que realmente dependem da comunicação confiável para conseguir seus objetivos.

Finalmente o capítulo 7, Conclusões, finaliza este trabalho resumindo o que

foi feito, o que se aprendeu, quais foram os problemas enfrentados, como foram resolvidos, quais são os trabalhos futuros e as outras possibilidades de se fazer injeção de falhas de comunicação sobre o Linux, que foram percebidos durante a execução deste trabalho.

2 Validação através da Injeção de Falhas

Este capítulo traz uma visão geral do que consiste a injeção de falhas, compilando assuntos de artigos recentes, e então aprofunda a área de injeção de falhas de comunicação. São revisadas as técnicas de injeção de falhas (por simulação, hardware e software) e injeção de falhas de comunicação, bem como são expostos modelos de falha comumente encontrados e formas de especificação de experimentos de injeção de falhas.

2.1 A necessidade de validação

Vivemos em uma sociedade cada vez mais dependente de recursos computacionais, principalmente de Sistemas Distribuídos, mesmo para alguns de seus serviços mais vitais. Nenhum destes sistemas está totalmente livre da ocorrência de falhas, portanto é necessário que contenham mecanismos de Tolerância a Falhas para garantir seu funcionamento correto, ou pelo menos a degradação não catastrófica de seus serviços.

Mesmo estes mecanismos de Tolerância a Falhas devem ser testados e depurados, pois de nada adianta um algoritmo que mascare ou recupere um determinado erro e crie outros. Algoritmos e protocolos podem ser testados e comprovados formalmente através de técnicas de matemática e lógica, provadores de teoremas e assim por diante. Assim se pode garantir a correção das especificações. Porém, um passo adicional ainda é necessário para estas especificações serem efetivamente utilizadas, a implementação.

A história nos mostra que mesmo bons programadores cometem erros, portanto nada garante que de uma *especificação* correta sempre se obtém uma *implementação* correta. Assim como se pode testar e comprovar formalmente uma especificação, se pode fazer o mesmo com uma implementação. Porém, com a atual complexidade dos sistemas microprocessados, ainda mais dos distribuídos, é realmente impossível se antecipar e verificar, em tempo razoável, todas as inúmeras possibilidades de comportamento desta implementação, quando da ocorrência de falhas.

Neste caso, a comprovação de funcionamento correto pode se dar através de uma técnica experimental, a Injeção de Falhas [ARL 93]. A Injeção de Falhas visa fornecer exatamente a entrada que os mecanismos de Tolerância a Falhas esperam, as falhas. Mais especificamente, ferramentas de Injeção de Falhas permitem que se crie um ambiente onde não só é certo que determinadas falhas ocorrerão, mas também que ocorrerão de forma controlada e que seu impacto no sistema alvo poderá ser medido e estudado posteriormente.

2.2 Introdução à Injeção de Falhas

A avaliação da dependabilidade envolve o estudo de defeitos e erros. A natureza destrutiva de um *crash* e a longa latência de erros tornam difícil identificar a causa de defeitos no ambiente operacional. É particularmente difícil recriar um cenário de defeito para um sistema amplo e complexo [HSU 97].

Para identificar e entender defeitos em potencial, se utiliza uma abordagem

baseada em experimentação para estudar a dependabilidade de um sistema. Uma abordagem deste tipo é aplicada não somente durante as fases de concepção e projeto, mas também ao longo de etapas de prototipação e operação.

Para se tomar uma abordagem experimental, deve-se primeiro entender a arquitetura, estrutura e o comportamento de um sistema. Especificamente, deve-se saber sua tolerância a falhas e defeitos, incluindo seus mecanismos internos de tecção e recuperação de erros, e precisa-se de instrumentos e ferramentas específicas para injetar falhas ou erros, e monitorar seus efeitos.

Várias técnicas já foram propostas para injeção de falhas. Geralmente, são baseadas em sistemas de hardware, software ou de simulação. Técnicas de hardware injetam falhas físicas no sistema alvo. Métodos baseados em simulação se utilizam de um modelo simulado do sistema em teste. A terceira opção permite emular falhas e erros de hardware através de software [CAR 98a]. As seções a seguir detalham um pouco mais cada uma destas opções.

2.2.1 Injeção de Falhas por Hardware

A injeção de falhas por hardware utiliza equipamentos extras para introduzir falhas no sistema físico em estudo. Dependendo das falhas e de suas localizações, os métodos de injeção de falhas por hardware se enquadram dentro de duas categorias [HSU 97]:

Injeção de falhas por hardware com contato: O injetor tem contato físico direto com o sistema alvo, produzindo alterações de corrente e voltagem externamente ao circuito alvo. Como exemplo cita-se métodos que utilizam pontas de prova para injeção em pinos e soquetes;

Injeção de falhas por hardware sem contato: O injetor não possui contato físico direto com o sistema alvo. Ao invés disso, uma fonte externa produz algum fenômeno físico, como radiação de íons pesados e interferência eletromagnética, causando correntes espúrias dentro do circuito em estudo.

Este tipo de Injeção de Falhas é o mais próximo que se pode chegar das falhas reais, visto que ocorrem no universo físico [PRA 96]. Ainda assim, existem três grandes desvantagens que tornam esta abordagem um tanto desinteressante. Em primeiro lugar, a injeção por hardware pode causar a queima de componentes e equipamentos, de forma que os custos podem ser altos e a colheita de resultados nem sempre é possível.

A segunda desvantagem é a dificuldade de controle sobre a injeção por hardware sem contato. Não se pode saber em que ponto dos circuitos os campos magnéticos estão realmente atuando, nem qual está sendo seu efeito. Finalmente, o desenvolvimento de injetores de falhas por hardware com contato é específico para o sistema em teste, e demanda muito tempo e conhecimento para ser corretamente implementado. Os injetores criados para uma plataforma dificilmente serão aproveitados para outro experimento, e novamente os custos podem ser altos.

2.2.2 Injeção de Falhas por Simulação

Nesta abordagem, falhas são injetadas em um modelo de simulação do sistema em estudo, o que permite total controle sobre a temporização, o tipo de falha e o

estado do componente afetado no modelo, com maior ou menor precisão, dependendo do nível de abstração do simulador.

Uma das vantagens desta técnica, que a torna agradável aos produtores de circuitos integrados é que ela pode ser utilizada logo no início da fase de projeto. Com um simulador é também possível injetar falhas bastante precisas e coletar informações detalhadas sobre seus efeitos. Como a injeção não se dá em tempo normal, e sim em incrementos de tempo simulados, pode-se levar o tempo que for necessário para analisar cada ponto de estudo, inclusive voltar no tempo simulado e tomar caminhos diferentes na simulação.

Embora pareça bastante atrativa, a injeção por simulação esbarra em dois grandes problemas. O teste é sobre um *modelo* do sistema, portanto todas as conclusões obtidas serão sobre este modelo, não sobre o sistema em si; quanto mais próximo o modelo estiver do sistema, mais próximas da realidade estarão as conclusões, mas mais tempo será gasto no desenvolvimento do modelo e na simulação. Em segundo lugar, conseguir modelos fiéis, completos, de sistemas microprocessados atuais é praticamente impossível, muitos segredos industriais estão envolvidos em qualquer processador atual.

2.2.3 Injeção de Falhas por Software

A Injeção de Falhas por software consiste da emulação de falhas reais (do universo físico) dentro do universo informacional. Esta abordagem poderia ser chamada Injeção de Erros, visto que é só o que pode fazer, porém não deve ser subestimada. Na Tolerância a Falhas implementada em software, que é muito comum hoje em dia, a Injeção de Falhas por software é exatamente o que se necessita, visto que na verdade o software só percebe as falhas na forma de erros.

A Injeção de Falhas por software consiste da criação de código que simule a ocorrência das falhas necessárias à validação do sistema. Este código normalmente deve ser inserido em algum local onde venha a ser ativado pelo sistema em teste, para que possa injetar as falhas. Sendo assim, pode-se pensar em alguns níveis onde o injetor pode estar. Por exemplo, o injetor pode estar dentro do sistema operacional, o que faz com que os processos em execução não percebam a presença do injetor. O injetor pode ainda estar no mesmo nível que as aplicações, executando como um processo ou uma thread. Finalmente, pode-se pensar em colocar o injetor no meta-nível, utilizando-se técnicas de Reflexão Computacional [ROS 98].

Estes três níveis criam uma boa gama de possibilidades para a implementação de um injetor, mas as vantagens são ainda maiores. O tempo de criação de um injetor por software é bem menor que o tempo gasto em simulação ou em criação de hardware específico. O código gerado pode ser generalizado de forma a permitir sua utilização em diferentes plataformas, diminuindo os custos totais de criação e adaptação de um injetor. Experimentos com um injetor por software podem ser mais facilmente conduzidos e adaptados que experimentos de injeção por hardware, e podem ter ativação mais precisa e mais flexível.

Tantas vantagens acabam por sobrepor as duas desvantagens desta abordagem: o fato da injeção ser de erros, não de falhas, e a alteração no ambiente de execução. A primeira desvantagem é inerente ao fato do injetor ser um programa, ser um código. Não há como, por exemplo, se testar um mecanismo de detecção e correção de erros de memória (ECC) apenas escrevendo nela. O mecanismo está em hardware, e é transparente ao software. Se for levado em consideração apenas a validação de

mecanismos de Tolerância a Falhas em software, esta abordagem ainda assim se aplica muito bem.

A segunda desvantagem (dependendo do ponto de vista, a *única* desvantagem) também é inerente ao fato do injetor ser um código a ser executado. Este código deve estar em algum lugar e eventualmente ser executado, de forma que o sistema se comportará de forma ligeiramente diferente de quando o injetor não estiver sendo utilizado. Algumas chamadas de sistema podem demorar mais para retornar, o sistema operacional pode demorar alguns milésimos ou milionésimos de segundo a mais para escalonar processos, menos memória livre estará disponível.

Este impacto no sistema em teste deve ser bem controlado, e na verdade nota-se que não necessariamente é nocivo. O próximo capítulo trata exatamente das abordagens de implementação de injetores de falhas em software e seus impactos.

2.3 Injeção de Falhas de Comunicação

Um sistema distribuído pode ser visto de duas formas: como um modelo físico, ou como um modelo lógico. Um modelo físico consiste de vários computadores em posições geográficas distintas, porém interligados por uma rede de comunicação. Um modelo lógico vê uma aplicação distribuída como um conjunto finito de processos e canais de comunicação entre os processos [JAL 94].

Em ambos modelos, os nodos (físicos ou lógicos) se comunicam através de mensagens. O serviço de comunicação é provido pelo sistema operacional através de primitivas *send* e *receive*. Para um processo, falhas de comunicação provenientes do universo físico e falhas emuladas pelo sistema operacional são indistinguíveis. A injeção de falhas de comunicação se dá exclusivamente através da manipulação de mensagens, a serem transmitidas ou recebidas, presentes em um dado nodo do sistema distribuído.

Para caracterizar a manipulação de mensagens, faz-se necessário distinguir dois momentos distintos: a seleção de uma mensagem a ser manipulada, e a manipulação propriamente dita [DAW 98]. As próximas seções discutem estes dois tópicos em profundidade.

2.3.1 Seleção de Mensagens

A seleção de uma mensagem para manipulação deve ser bastante criteriosa, envolvendo vários tipos de condições. Para que uma ferramenta seja flexível, deve-se poder selecionar precisamente onde ativá-la, ou seja, sobre que mensagem agir. Não basta apenas aplicar ações sobre as mensagens aleatoriamente, ou sobre todas ou utilizar alguma outra forma ainda menos específica.

Além da capacidade óbvia de diferenciar mensagens transmitidas de recebidas, identifica-se três tipos básicos de seleção: seleção baseada em conteúdo de uma mensagem, seleção baseada em fluxo de mensagens, e seleção baseada em elementos externos às mensagens.

Seleção baseada em conteúdo: Aqui estão agrupadas as condições de seleção que dependem do conteúdo de uma mensagem, ou seja, de algum padrão existente em sua seqüência de bytes. Aqui se encontram, por exemplo, seleção de mensagens de ACK de um protocolo específico, ou seleção de mensagens de

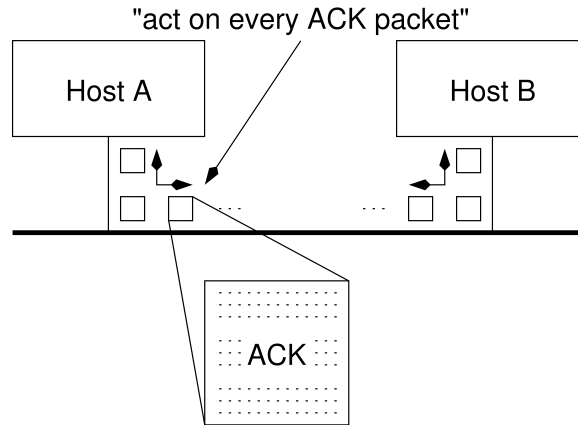

FIGURA 2.1 – Seleção baseada em conteúdo

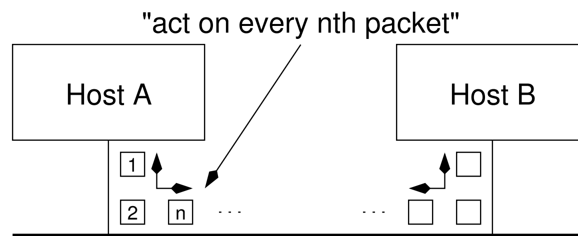

FIGURA 2.2 – Seleção baseada em fluxo

terminação de uma conexão (figura 2.1);

Seleção baseada em fluxo: Este grupo contém as seleções que dependem do fluxo de mensagens, e não de seu conteúdo individual. Por exemplo, pode-se desejar manipular a terceira mensagem de uma conversação entre dois processos, ou manipular 20% das mensagens, ou até mesmo manipular uma a cada cinco mensagens (figura 2.2);

Seleção baseada em elementos externos: Nesta categoria estão as condições de seleção que não dependem diretamente das mensagens, mas sim de temporizadores, variáveis controladas pelo usuário, medições de algum fenômeno físico, etc (figura 2.3).

A combinação de condições de seleção provenientes dos grupos acima permite uma extrema flexibilidade na especificação das mensagens alvo de uma ferramenta

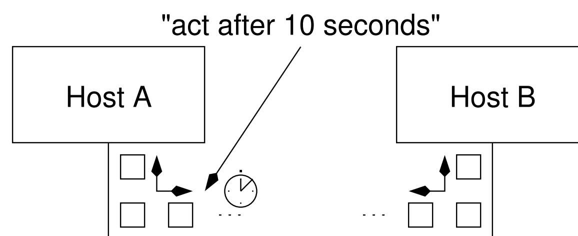

FIGURA 2.3 – Seleção baseada em elementos externos

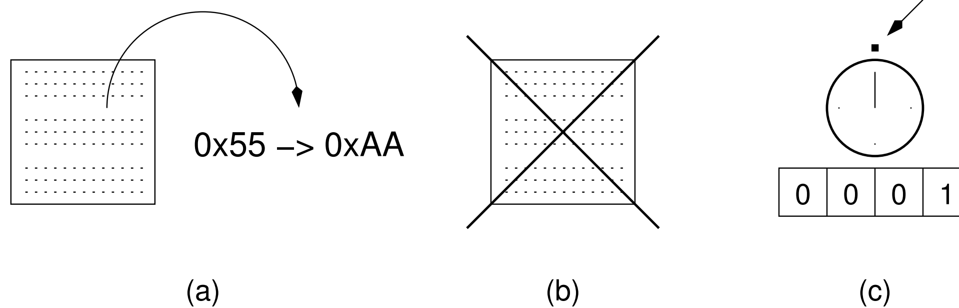

FIGURA 2.4 – Tipos de ações sobre mensagens

de injeção de falhas de comunicação. Observa-se que uma ferramenta deve permitir que se altere suas características de seleção de mensagens em tempo de execução, dando ao condutor dos testes a capacidade de adaptar seu experimento de acordo com os resultados preliminares obtidos.

2.3.2 Manipulação de Mensagens

Depois de estabelecidos os critérios de seleção de mensagens para manipulação, deve-se especificar o que será feito delas, ou seja, qual é a ação que será tomada sobre a mensagem selecionada. Da mesma forma que acontece com a seleção de mensagens, uma ferramenta flexível deve permitir uma ampla gama de ações sobre mensagens.

Novamente, pode-se caracterizar três classes de ação quando da seleção de uma mensagem: ação interna a uma mensagem, ação sobre a mensagem em si, ou ação não relacionada a nenhuma mensagem. Os parágrafos a seguir exemplificam estes casos.

Ação interna: Nesta categoria estão as ações que visem alterar algum campo da mensagem, como por exemplo alterar um sinalizador, simular um fechamento de conexão, alterar um endereço, etc. São todas ações que não implicam na impossibilidade de se transmitir[1] ou entregar[2] imediatamente a mensagem (figura 2.4a);

Ação sobre a mensagem: Aqui se encontram as ações mais comuns, como perda de mensagens (caso em que a mensagem escolhida é simplesmente descartada), atraso de mensagens (adição da mensagem a uma fila de entrega posterior), duplicação de mensagens, etc (figura 2.4b);

Ação sobre elementos externos: Aqui estão agrupadas as ações que, embora decorram do fato de alguma mensagem ter passado por critérios de seleção, não atuam diretamente sobre a mensagem que as causou. Exemplos são: atuar sobre contadores, temporizadores, alterar variáveis, gerar um aviso ou registro no sistema, etc (figura 2.4c).

Uma ferramenta que permita vários destes tipos de ação fornece uma grande capacidade de criação e flexibilidade na especificação de experimentos. Pode-se chegar à possibilidade mesmo de imitar o funcionamento de outras ferramentas menos poderosas.

[1]Transmitir para a rede uma mensagem originada pelo processo.
[2]Entregar ao processo uma mensagem proveniente da rede (*deliver*).

2.3.3 A Ferramenta ORCHESTRA

A ferramenta ORCHESTRA [DAW 98] implementa muito do que foi revisto, pois além de permitir a especificação do experimento através de *scripts*, proporciona os seguintes tipos de operações sobre mensagens:

Filtragem de mensagens: Para interceptar e examinar mensagens;

Manipulação de mensagens: Para perder, atrasar, reordenar, duplicar ou modificar uma mensagem;

Injeção de mensagens: Para testar uma máquina através da inserção de mensagens no sistema.

A ferramenta ORCHESTRA permite também que sejam definidos diferentes conjuntos de regras para transmissão e recepção de mensagens. As decisões podem ser baseadas em atributos das mensagens, no histórico das mensagens, ou em dados coletados pela camada de injeção de falhas, como contadores.

Nota-se nesta descrição da ORCHESTRA que existem as três possibilidades de condições para seleção de mensagens, tanto baseada nos conteúdos de uma única mensagem, quanto em características do fluxo de mensagens, como também em dados externos como temporizadores e contadores.

As possibilidades de manipulação igualmente atingem os três grupos de possibilidades: pode-se agir internamente em uma mensagem, agir sobre a mensagem como um todo, ou agir sobre elementos externos, pertencentes à ferramenta de injeção. Por estas razões a ferramenta ORCHESTRA é tida como uma das motivações e guia das ações deste trabalho.

2.4 Modelos de Falha

Quando se fala em injeção de falhas, logo se pensa que, para que o experimento seja válido, é necessário se criar um modelo de falhas específico para o protocolo ou processo em teste. Modelos de falhas bem especificados permitem a repetição de um experimento tantas vezes quantas sejam necessárias para atingir um objetivo.

A possibilidade de se especificar detalhadamente quando inserir falhas e quais falhas devem ser inseridas é crucial para que se possa conduzir protocolos e aplicações a estados "difíceis de alcançar" [DAW 96a]. Como muitos protocolos não são exercitados em toda a sua potencialidade no uso comum, um protocolo de comunicação confiável pode parecer satisfatório, porém apresentar um caso específico, difícil de alcançar em uso normal, que corrompa suas estruturas de funcionamento.

Os modelos usuais de falhas em sistemas distribuídos, como o apresentado por Christian [CHR 86], consideram, quase que exclusivamente, falhas de comunicação. Assim um nodo está em *crash* quando pára de enviar mensagens. É omisso quando não envia ou não recebe algumas mensagens. Está com falhas de temporização quando atrasa ou adianta mensagens. Apresenta comportamento bizantino quando, além das falhas já mencionadas, pode alterar o conteúdo das mensagens ou enviar mensagens contraditórias para os demais nodos do sistema. Assim, um injetor de falhas que atua sobre o sistema de comunicação tem a potencialidade de validar uma vasta gama de protocolos de tolerância a falhas nesses sistemas.

Além dos já citados, modelos de falha comuns na literatura de sistemas distribuídos incluem [HAD 93]: falhas de *crash* de processos, falhas de *crash* de rede, falhas de omissão de transmissão ou recepção, falhas de temporização ou desempenho, falhas arbitrárias (bizantinas) e falhas de computação incorreta. A maior parte destas falhas atinge mensagens em si, e não seu conteúdo, porém existe uma vasta possibilidade de injeção de falhas bastante específicas quando se altera o conteúdo de uma mensagem. Por exemplo: falhas de endereçamento, falhas de tipo de protocolo, falhas em flags específicas e falhas de acordo com o significado do protocolo.

Uma experimentação feita com seis implementações comerciais diferentes do protocolo TCP [DAW 96a], utilizou vários modelos diferentes de falhas para testar cada aspecto das implementações. Para testar os intervalos de retransmissão, a máquina com a ferramenta de injeção de falhas foi configurada para deixar passar os primeiros trinta pacotes recebidos da rede, e então descartar todos os pacotes subseqüentes. Cada pacote perdido era registrado pela ferramenta junto com sua marca de tempo.

Como a ferramenta perdia os pacotes além do trigésimo, a máquina onde a ferramenta estava sendo executada não percebia estes pacotes, e então não enviava reconhecimentos. As máquinas que tinham conexão com esta então começavam a retransmitir os pacotes enviados, considerando-os perdidos pela rede. Isto serve para demonstrar também que uma máquina com uma ferramenta de injeção de falhas pode ser utilizada na experimentação e avaliação de outras máquinas, com as quais ela irá interagir.

Uma ferramenta de injeção de falhas, além de apresentar modelos de falha prontos, deve também possibilitar a criação de modelos através da combinação de primitivas de seleção e manipulação de mensagens. Experimentos específicos requerem modelos de falhas específicos, portanto a flexibilidade de especificação da ferramenta determina o que pode ser experimentado.

Ferramentas provenientes da área de Redes de Computadores normalmente são do tipo *traffic shaper*, ou seja, fazem com que todas as mensagens de uma máquina, ou uma controladora de rede de uma máquina, ou mesmo de uma conexão específica sofram ação de alguma política de fila, que lhes atribui latência ou largura de banda específica. Em experimentos de Tolerância a Falhas, é importante se especificar experimentos com maior precisão.

Por exemplo, pode-se querer testar exatamente o que ocorre quando um protocolo de *two-phase commit* sofre perdas de mensagens na segunda fase de operação. É importante deixar o protocolo executar livremente até esta fase, e então injetar as falhas. Mas como perceber isto? Na verdade, a criação de um modelo de falhas, para um experimento de injeção de falhas de comunicação, demanda um profundo conhecimento do protocolo em teste, e também das possibilidades da ferramenta de injeção de falhas utilizada.

É impossível projetar uma ferramenta de injeção de falhas de comunicação que conheça o significado de cada mensagem de cada protocolo já inventado ou que venha a ser projetado, mas uma ferramenta flexível permite que uma pessoa com conhecimento do protocolo em teste possa selecionar mensagens mesmo baseado em bits específicos, como por exemplo um sinalizador de que o protocolo de *two-phase commit* está em sua segunda fase. A idéia é que a ferramenta não necessite saber do significado de tudo em uma mensagem, porém auxilie quem o sabe permitindo o teste de tudo.

2.5 Especificação de experimentos de Injeção de Falhas

A forma de se especificar experimentos de Injeção de Falhas está intimamente ligada à forma com que se dá a injeção em si. Quando se utiliza Injeção de Falhas por Simulação, como existe controle completo, pode-se simplesmente interromper a execução do simulador e alterar arbitrariamente algum valor, de forma a injetar manualmente uma falha no sistema. Mas isto ainda não é o mais interessante, principalmente quando se pensa que o simulador é um programa, e programas podem ser ampliados para fazer todo o trabalho pesado.

Um simulador pode incluir mecanismos de *triggers*, que sejam disparados quando determinadas variáveis da simulação atingem um valor específico. Assim se automatiza a injeção, e se obtém o determinismo, essencial para a reprodução dos experimentos. Estes *triggers* podem ser armazenados como parte do modelo de simulação, ou mesmo como entidades separadas, de forma que múltiplos experimentos possam ser facilmente executados sobre um mesmo modelo de sistema. Pequenos *scripts* podem ser criados, coordenando as ações do simulador.

Na Injeção de Falhas por Hardware, as técnicas sem contato inerentemente não possuem nenhum determinismo, portanto na verdade acabam por não permitir uma forma prática de se reconstruir um experimento (não se consegue nem controlar exatamente o que se experimenta). Já quando há contato, pode-se então utilizar os próprios pontos de coleta de informação (níveis lógicos) como disparadores da injeção. Por exemplo, pode-se injetar falhas de barramento apenas quando um determinado endereço é acessado. O injetor pode ser feito, então, de forma a armazenar seqüências de injeções disparadas por eventos percebidos no sistema em teste.

Com certeza os injetores de falhas em software permitem a maior flexibilidade em termos de especificação e armazenamento de experimentos. Qualquer que seja o injetor, ele pode ser parametrizado, e ter estes parâmetros alterados. Adicionando-se uma forma padronizada de armazenamento destes valores tem-se um injetor flexível, que permite que experimentos sejam criados, aplicados, arquivados e alterados. Mais interessante ainda é a possibilidade de se embutir no injetor de falhas uma linguagem de alto nível, que permita criar com uma sintaxe clara experimentos baseados em rotinas de injeção acessíveis através desta linguagem.

Praticamente todos os injetores de falhas em software utilizam uma arquitetura modular, onde através de arquivos de entrada (especificando experimentos) e de saída (com o registro das operações do injetor) se pode representar muito claramente a atividade do injetor. A ferramenta ORCHESTRA [DAW 96b] possui exatamente esta abordagem, permitindo a execução de experimentos através de *scripts* escritos em uma extensão à linguagem TCL, que podem ser alterados sem que se precise reiniciar o computador ou o ambiente de injeção de falhas.

Mesmo que não se use uma linguagem de alto nível tão flexível quando TCL ou Perl, definindo-se uma linguagem com primitivas de injeção de falhas independentes das possibilidades desta ou daquela arquitetura, permite-se uma descrição de alto nível do experimento sendo conduzido, que pode então ser repetido em ferramentas que implementem a semântica da linguagem em outro sistema computacional. O injetor se torna mais flexível e portanto mais facilmente aplicável.

Claro que tamanha flexibilidade nem sempre é possível, dependendo do ambiente onde a ferramenta de injeção se insere. O próximo capítulo discute a influência dos locais onde um injetor de falhas em software pode ser implementado.

2.6 Conclusão

Este capítulo demonstrou a importância da injeção de falhas na validação experimental da implementação de aplicações e protocolos tolerantes a falhas, bem como introduziu aspectos interessantes e elucidativos a respeito da injeção de falhas de comunicação.

Embora já existam muitas ferramentas de injeção de falhas, poucas são as que permitem injetar falhas de comunicação, e ainda mais raras são as que permitem uma boa flexibilidade no tipo de falha de comunicação injetado. Existe pouca bibliografia na área de injeção de falhas de comunicação, portanto com este capítulo espera-se ter contribuído idéias e associações novas, bem como concentrado idéias já presentes em outros trabalhos.

3 Implementação de Injetores de Falhas em Software

Este capítulo trata das formas de se implementar injetores de falhas em software e seus impactos no ambiente de execução em que estão inseridos, visto que não podem estar presentes sem causar alguma alteração da carga ou das características temporais do sistema. Será também discutido como inserir injetores de falhas em diferentes arquiteturas de sistemas operacionais.

Como uma classificação genérica, pode-se ver um injetor de falhas em software como podendo estar, dependendo de seu objetivo, em dois níveis: no nível da aplicação e no nível do sistema operacional. Se o alvo da injeção de falhas é uma aplicação, o injetor de falhas pode ser inserido em vários locais: na própria aplicação (necessita a disponibilidade de código fonte ou objeto), entre a aplicação e o sistema operacional (em bibliotecas ou *wrappers*), ou no próprio sistema operacional (permite a execução da aplicação de forma inalterada). Se o alvo for algo no próprio sistema operacional, o injetor deve ser embutido neste próprio, visto que é bastante difícil se colocar uma camada entre o sistema operacional e o hardware [PRA 96].

3.1 Injeção de Falhas no nível da aplicação

A inserção do injetor de falhas no nível da aplicação visa a introdução de falhas na própria aplicação sob teste, ou em código que execute em mesmo nível. Neste caso, a aplicação implementa um protocolo para alcançar tolerância a falhas em sistemas distribuídos ou é um programa mais complexo que possui ou usa procedimentos para alcançar tolerância a falhas. A abordagem apresenta vantagens principalmente quando o protocolo e o injetor são desenvolvidos em conjunto. Injetores que implementam esta abordagem podem estar no código da aplicação, em código concorrente com a aplicação, em código usado pela aplicação ou no meta-nível.

3.1.1 Injeção no código da aplicação

Considerando o protocolo a ser validado e o injetor de falhas um único processo [BAR 99], existe a necessidade de contar com o código fonte do protocolo disponível para alteração. Isto porque se o injetor de falhas irá fazer parte do protocolo durante a execução do teste, são necessárias alterações no código fonte do protocolo para que o mesmo possa atuar em conjunto com o injetor.

O protocolo sob teste deve incluir, em seu código, chamadas às rotinas do injetor de falhas. A cada chamada ao injetor, o protocolo sob teste desvia sua operação normal, executa as ações do injetor e então retorna a sua execução. O posicionamento correto das chamadas no código fonte do protocolo exige uma análise cuidadosa do código e uma boa documentação do protocolo sob teste. Para injeção de falhas de comunicação, por exemplo, todas as chamadas a rotinas de comunicação, como send, receive, broadcast, multicast, deliver e outras funcionalmente semelhantes podem ser substituídas por rotinas do injetor. Alternativamente, as próprias rotinas de comunicação do protocolo podem ser alteradas.

Esta abordagem de implementação restringe a utilização do injetor para aque-

le protocolo específico sob teste, que sofreu alteração no seu código. A portabilidade para outros protocolos fica comprometida. Cada novo protocolo a ser validado deve ter seu código minuciosamente analisado para determinação dos melhores pontos para posicionamento das chamadas às rotinas de injeção de falhas. Outra desvantagem corresponde a integridade do protocolo.

Deve-se garantir que o procedimento de validação não irá comprometer a execução do protocolo em si. Como o injetor roda no mesmo espaço de endereçamento do protocolo e com os mesmos privilégios, ele tem acesso a todos os recursos do protocolo e pode, inadvertidamente, alterar o comportamento do protocolo, mascarando as medidas de confiabilidade que podem ser obtidas pelo teste.

Essa abordagem entretanto pode ser bastante útil quando o injetor é desenvolvido simultaneamente com a implementação do protocolo. Nesse caso o desenvolvedor tem perfeito domínio sobre o código e pode tirar proveito da ausência de proteção do espaço de endereçamento do protocolo em relação ao injetor.

Ainda no nível de aplicação, existe a possibilidade de se inserir em um programa um código adicional que crie uma *thread* extra que executará o injetor. Este é o ponto máximo de intrusão que se pode chegar, visto que altera o código fonte, altera o ambiente de execução, compartilha espaços de memória e rouba ciclos de CPU do sistema em teste. Problemas inesperados com o injetor podem atrapalhar a aplicação, e pode ser necessário mudar a característica de implementação dos algoritmos do sistema em teste, para acomodar a inspeção e possível alteração pelo injetor.

Para tirar o máximo proveito da implementação dessa abordagem, entretanto, o sistema operacional deve poder suportar a execução de *threads*. Assim, quando a *thread* do protocolo abandona a CPU por alguma razão, a *thread* do injetor pode executar e liberar a CPU antes do protocolo voltar a execução. O impacto sobre a temporização do sistema nesse caso é mínima, considerando que o injetor executa apenas umas poucas e rápidas tarefas cada vez que assume a CPU. Se *threads* são suportadas apenas pela linguagem de programação, entretanto, quando o protocolo abandona a CPU leva junto o injetor, e a vantagem da redução do tempo de execução se anula.

3.1.2 Injeção por processos concorrentes à aplicação

Esta abordagem consiste da criação de um processo injetor, que vai de alguma forma alterar o ambiente de execução do processo simulando a ocorrência de falhas. Uma possibilidade é a injeção de falhas via chamadas de sistema de depuração [GON 00][KAN 95]. Esta técnica é especialmente interessante por não necessitar da disponibilidade de código fonte do sistema em teste. Esta característica é vital para organizações independentes de teste e validação, que recebem sistemas prontos, fechados, cujo código fonte normalmente é coberto por patentes e registros comerciais.

Neste caso, o processo correspondente ao protocolo sob teste e o processo injetor executam concorrentemente. Somente haverá interação entre os dois processos na inserção de uma falha e na posterior coleta de resultados da inserção. Entretanto, tais procedimentos não implicam em alterações de código dos processos. Para interagir com o protocolo sem alterar o código do mesmo, o processo injetor de falhas deve receber privilégios especiais para alterar determinadas áreas de dados do processo protocolo (por exemplo para suprimir mensagens, alterar a ordem ou a

própria mensagem).

Para tanto o injetor deve saber determinar quais as áreas alocadas ao protocolo (a cada momento) que correspondem à manipulação de mensagens. Determinar as áreas e obter privilégios de acesso a outros processos não é uma tarefa trivial para processos no nível da aplicação e está intimamente relacionada ao recursos que um determinado sistema operacional pode oferecer aos processos. Se entretanto, for prevista no desenvolvimento do protocolo a validação por um injetor, as áreas de manipulação de mensagens podem ser declaradas como de acesso comum a mais de um processo, facilitando a obtenção dos privilégios necessários por parte do injetor.

Outra vantagem em relação à abordagem anterior se refere à portabilidade. Sendo o protocolo da aplicação um processo em separado do injetor de falhas é possível, com pouca ou nenhuma alteração, utilizar o mesmo injetor de falhas para protocolos de comunicação com características semelhantes.

A desvantagem com relação à integridade da aplicação, apresentada pela abordagem anterior, é consideravelmente menor nesta abordagem, mas não totalmente ausente. Isto se deve ao fato de o injetor de falhas não interferir diretamente no código da aplicação, ou seja, a execução dos módulos injetor de falhas e protocolo da aplicação em dois processos distintos evita a interferência de um no código do outro. A única interação que ocorre nesta abordagem se refere à introdução das falhas, por parte do injetor, na área de manipulação de mensagens durante a execução.

Não existe alteração manual do código fonte e nem interferência dinâmica na área de código durante a execução, isso se o esquema de proteção do sistema operacional for adequado. Entretanto, nessa abordagem o custo de chaveamento de contexto para a execução alternada dos dois processos é considerável. Se o objetivo é minimizar a carga do injetor no sistema e evitar comprometer o seu comportamento temporal, a aplicação dessa solução deve considerar cuidadosamente os tempos envolvidos no chaveamento de contexto.

3.1.3 Injeção em código utilizado pela aplicação

Uma abordagem de implementação interessante é a utilização de bibliotecas dinâmicas modificadas, cujo uso pode ser ativado ou desativado facilmente. Nos sistemas padrão Unix, por exemplo, a variável `LD_LIBRARY_PATH` pode conter um diretório onde se encontram bibliotecas a serem carregadas antes de se verificar os diretórios padronizados do sistema operacional. Isto permite que se execute um mesmo programa, sem alteração, porém que sofrerá a ação de uma biblioteca dinâmica com funções modificadas, que podem muito bem injetar qualquer tipo de erro no processo ao qual estão ligadas.

Esta técnica se parece muito com as técnicas que envolvem alteração no código da aplicação, porém não necessita de código fonte. É aplicável somente a programas que não tenham sido ligados estaticamente, para que possam usar a biblioteca dinâmica alterada. Novamente um problema no injetor pode atrapalhar a aplicação. Na verdade esta abordagem mistura algumas das vantagens e desvantagens tanto da inserção no código da aplicação quanto da inserção em processos concorrentes à aplicação.

3.1.4 Injeção no meta-nível

Pode-se utilizar técnicas de programação reflexiva para a implementação do injetor de falhas [ROS 98]. Reflexão é uma maneira fácil de separar funcionalidades de implementação do sistema alvo. A reflexão introduz um novo modelo de arquitetura, no qual existem dois níveis: o meta-nível e o nível funcional. O nível funcional pode ser usado para implementar os objetos do sistema alvo enquanto o meta-nível permite que programadores observem e manipulem estruturas de dados e/ou ações realizadas no nível funcional. Desta forma, chamadas a métodos podem ser interceptadas, e então parâmetros de entrada ou saída podem ser monitorados e modificados. Um perfil da execução no nível funcional pode ser obtido, e falhas podem ser injetadas com uma ativação bastante flexível.

As vantagens desta abordagem são a transparência, uma vez que para o procedimento de injeção não há interrupção explícita do código da aplicação ou execução desta em modo de depuração, como ocorre usualmente em mecanismos de injeção de falhas por software. Outra vantagem é o reuso, já que os objetos responsáveis tanto pela injeção de falhas como pelo monitoramento podem ser incorporados a outras aplicações. Além disso, injeção de falhas no meta-nível não necessita da execução do programa em modo privilegiado, tampouco é preciso hardware dedicado. Entretanto, nesta abordagem existe interação com a execução do programa e há exigência de disponibilidade do código da aplicação.

Injetores de falhas no meta-nível se prestam bem a sistemas orientados a objetos e podem ser usados para validar protocolos de comunicação em sistemas distribuídos. Os meta-objetos do meta-nível podem interceptar a comunicação entre objetos distribuídos e dessa forma selecionar e manipular as mensagens desejadas durante a injeção de falhas.

3.2 Injeção de Falhas no nível do Sistema Operacional

A melhor forma de implementação de um injetor de falhas em software parece ser dentro do sistema operacional. Isto porque um processo só conhece o mundo externo através das chamadas de sistema que o sistema operacional lhe proporciona. Se um processo implementa um protocolo de comunicação confiável, ele pede através de chamadas de sistema que o sistema operacional entregue estas mensagens. Para este processo, uma falha real na rede e uma falha injetada pelo sistema operacional são indistinguíveis. Assim também acontece para falhas em memória e em registradores, por exemplo.

Esta abordagem possui a vantagem de permitir que o protocolo sob teste execute sem qualquer interferência do injetor de falhas. Esta característica aumenta o reuso do injetor de falhas. A portabilidade da ferramenta está vinculada à portabilidade do sistema operacional usado como plataforma. Entretanto, por encontrar-se no nível do sistema operacional e utilizar-se dele para atuar, o injetor de falhas pode interferir em sua integridade. Além disso, há necessidade de alterações do próprio sistema operacional, que deve suportar a presença de intrusos.

Aqui é importante salientar que a forma com que a falha é injetada pode acabar por dar um significado diferente à falha. Suponhamos uma mensagem gerada por um processo e que deve ser descartada pelo sistema operacional, para simular uma falha de perda de pacote na rede. Caso a falha real acontecesse, a mensagem passaria

por todos os protocolos de rede utilizados, atualizando números de série e *flags*, por exemplo, e então seria enviada pela placa de rede, que colocaria em suas estatísticas o número de mensagens enviadas, de bytes, de pacotes com erro, colisões, etc. A mensagem seria perdida na rede, porém já teria influenciado todo o código por onde passou.

Sendo assim, se fosse escolhido descartar a mensagem no ponto mais próximo do processo, digamos na chamada de sistema `socket_call`, esta não influenciaria todo o código do protocolo TCP, por exemplo, portanto os números de série estariam errados para a próxima mensagem. Do ponto de vista do protocolo, não houve mensagem. Isto não serve como uma emulação de falha de rede, pois a próxima mensagem será entregue normalmente e tudo prosseguirá como se aquela mensagem descartada nunca tivesse sido gerada. Isto é uma falha da aplicação, não da rede.

3.2.1 Influências sobre a implementação no Sistema Operacional

Muitas são as formas de influência que um sistema operacional exerce sobre uma ferramenta de injeção de falhas. Principalmente, são afetadas a localização, as possibilidades de disparo e ação do injetor, e também a possibilidade de reconfiguração da ferramenta ou do experimento em tempo de execução.

Com um sistema operacional do qual não está disponível o código fonte, como é o caso dos sistemas comerciais, é normalmente impossível, ou pelo menos bastante difícil, a inclusão da ferramenta dentro do sistema. Deve-se neste caso recorrer aos mecanismos citados anteriormente que colocam o injetor no mesmo nível que a aplicação.

A implementação de um injetor de falhas em um sistema operacional está obviamente ligada à arquitetura deste sistema operacional. Um sistema do tipo *microkernel*, por exemplo, mantém, através da abstração criada pela troca de mensagens, uma fronteira muito bem definida entre cada subsistema. Isto permite que se troque apenas para um processo em teste os servidores de determinados dispositivos, para que estes lhe pareçam falhos.

O Sistema Operacional GNU/Hurd, por exemplo, permite que certos processos tenham suas chamadas de sistema substituídas apenas pela troca de servidores em tempo de execução [STA 00]. Uma ferramenta de injeção de falhas de comunicação pode simplesmente substituir o processo responsável pela comunicação em rede e, como a própria ferramenta é um processo, pode até mesmo rodar *scripts* interpretados para decidir o que fazer a cada mensagem.

Já um sistema operacional do tipo monolítico não apresenta grande flexibilidade quanto à reconfiguração, sendo freqüentemente necessário reinicializar o sistema para alterar parâmetros de operação. Ainda assim, uma ferramenta bastante parametrizada pode permitir algum nível de flexibilidade mesmo quando não se pode adicionar ou retirar código de um sistema em funcionamento.

O Sistema Operacional Linux é do tipo monolítico, porém com um adicional que o torna bastante flexível: a possibilidade de inserção e retirada de módulos de código executável do *kernel* em funcionamento. Nas versões atuais do Linux, pode-se configurar como módulos do *kernel* praticamente qualquer pedaço de código não essencial para o *boot* do sistema. Depois de inicializado, o *kernel* procura no sistema de arquivos e insere no código em funcionamento *drivers*, protocolos e sistemas de arquivo secundários, à medida que se fazem necessários.

É questionável a possibilidade de se inserir um interpretador de alguma lin-

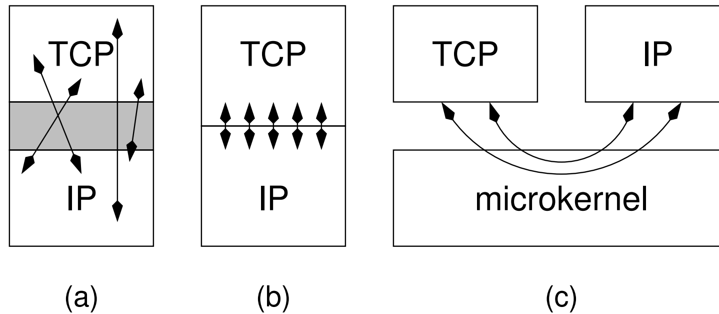

FIGURA 3.1 – Diferentes implementações de uma pilha TCP/IP

guagem no núcleo de um sistema operacional, já que o desempenho do núcleo afeta o sistema como um todo. Porém, existe a possibilidade de se instrumentar o código de forma que ele percorra uma lista de regras definidas pelo usuário para cada mensagem manipulada. Com uma linguagem otimizada, especialmente codificada para este tipo de aplicação, a exemplo de informações de roteamento ou as regras de um *firewall*, pode-se chegar a uma ferramenta quase tão flexível quanto uma baseada em linguagens interpretadas de alto nível.

Outro ponto de grande diferença entre modelos de sistemas operacionais diz respeito à interdependência de código. Sistemas monolíticos têm a tendência a ter um código mais otimizado, visto que todas as funções e variáveis globais estão no mesmo espaço de endereçamento. Os limites entre as partes funcionais nem sempre são bem estabelecidos, bem como a interface entre elas. Nas figuras 3.1a e 3.1b são retratadas duas implementações hipotéticas de uma pilha TCP/IP em sistemas monolíticos.

A figura 3.1a ilustra uma implementação bastante confusa e com uma área de indefinição quanto a que parte o código pertence. Já na figura 3.1b tem-se uma implementação monolítica, porém com interface bem definida entre os dois protocolos. Por fim, a figura 3.1c mostra uma possível implementação baseada em *microkernel*, possuindo a interface mais clara e compreensível de todas.

3.2.2 Inserção de injetores de falhas de comunicação

Muitos protocolos de comunicação em sistemas distribuídos são organizados em pilhas. Em uma pilha de protocolos, cada camada de protocolo depende da camada subseqüente (dita mais abaixo) para a utilização de alguns serviços. A abordagem da ferramenta PFI [DAW 96a] (precursora da ORCHESTRA) para testar estes sistemas coloca uma camada de injeção de falhas entre duas camadas na pilha de protocolos. Na maioria dos casos a camada de injeção de falhas, chamada camada PFI, é inserida diretamente abaixo da camada alvo, que é a camada a ser testada.

Embora seja possível colocar a camada PFI em camadas mais baixas da pilha, o teste é normalmente mais fácil se a camada de injeção de falhas está imediatamente abaixo da camada alvo, porque todos os pacotes (mensagens) que a camada PFI vê são pacotes do protocolo alvo.

Uma vez que a camada PFI tenha sido inserida na pilha de protocolos abaixo da camada alvo, cada mensagem enviada ou recebida pela camada alvo passa pela camada PFI. Esta camada pode então manipular estas mensagens de forma a injetar falhas e modificar o estado do sistema [DAW 96a].

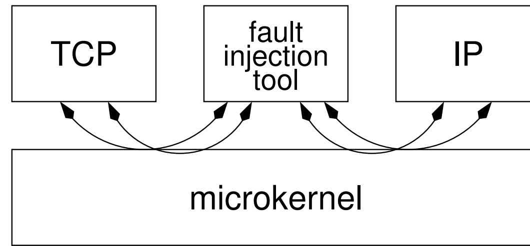

FIGURA 3.2 – Inserção de uma ferramenta em um *microkernel*

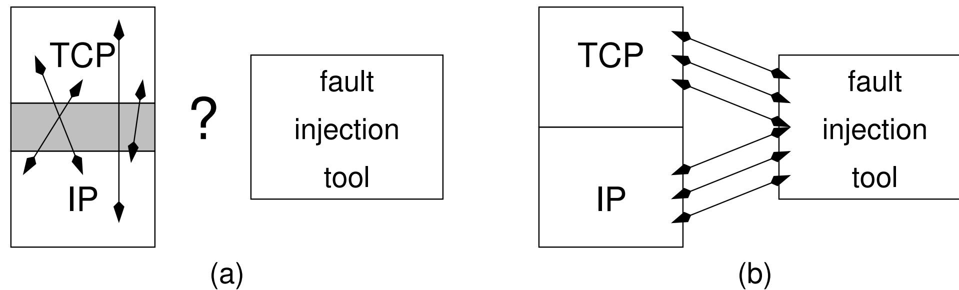

FIGURA 3.3 – Inserção de uma ferramenta em implementações monolíticas

Assim como o sistema operacional alvo de uma ferramenta de injeção de falhas de comunicação influencia suas possibilidades de configuração e flexibilidade, ele também influencia na maneira como a ferramenta pode se inserir entre as pilhas de protocolos. Um sistema operacional do tipo *microkernel* possui a maior flexibilidade neste caso, visto que as camadas de protocolos podem ser implementadas em processos separados, permitindo a interceptação de mensagens e reorganização da pilha de protocolos sem a alteração do núcleo do sistema operacional (figura 3.2).

Em sistemas de implementação monolítica, freqüentemente as pilhas de protocolos suportados possuem uma grande parte dos dados compartilhados entre as camadas, o que aumenta o desempenho. Porém isto dificulta a definição das fronteiras reais entre as camadas, e mesmo coloca em dúvida a validade da estratégia da implementação em camadas. Torna-se difícil, portanto, garantir a possibilidade de inserção da ferramenta de injeção de falhas entre quaisquer camadas (figura 3.3a). Ainda assim, uma implementação monolítica bem estruturada permite a inserção da ferramenta com algumas alterações (figura 3.3b).

Uma fronteira que sempre existe, contudo, é a fronteira entre as diversas pilhas de protocolos existentes em um sistema operacional e os dispositivos físicos, as interfaces de rede que irão efetivamente transmitir as mensagens. Neste ponto tem-se a possibilidade de filtrar e analisar todos os pacotes de rede enviados e recebidos pelo sistema, o que pode ser visto tanto como uma vantagem quanto como uma desvantagem.

É uma vantagem por permitir que se analise diferentes protocolos ao mesmo tempo, principalmente quando existem protocolos de auxílio de configuração e diagnóstico, como é o caso do ICMP, no conjunto de protocolos TCP/IP. Ao se injetar falhas em uma conexão TCP, UDP ou IGMP, pode ser interessante analisar também as respostas recebidas de outras máquinas ou geradas pela máquina em teste através do protocolo ICMP. Outros casos podem ser imaginados, como a

análise de informações de protocolos de roteamento quando da simulação de falhas de *link*.

Também pode ser visto como uma desvantagem por aumentar a quantidade de processamento necessário a cada mensagem enviada ou recebida, o que pode ser significativo quando a injeção de falhas é bastante específica. Pode não ser aceitável percorrer regras de alteração que visem um determinado protocolo quando este é responsável por apenas uma pequena fração do tráfego observado na rede em questão.

3.3 Conclusão

Este capítulo mostrou aspectos interessantes da implementação de injetores de falhas em software. A Injeção de Falhas por software se mostra uma alternativa bastante prática, pois usa o sistema real nos testes e tem um custo de implementação, adaptação e experimentação bastante baixo. Um injetor em software é uma ferramenta flexível, cuja ativação pode ser bem controlada. Quando feito no nível do sistema operacional, um injetor de falhas se torna um sistema de testes poderoso, pois pode-se chegar muito próximo das falhas reais, embora mantendo uma certa facilidade de configuração e condução de experimentos.

O que se pode notar como regra geral é que deve-se estar o mais próximo possível do hardware, porém não tão perto que o trabalho de programar e inserir o injetor tenha que ser refeito para cada gerenciador de dispositivo presente no sistema operacional. Um sistema operacional bem projetado permite isto, como é o caso do Linux, assunto do próximo capítulo.

4 O Sistema Operacional Linux

Este capítulo dá uma rápida apresentação do Sistema Operacional Linux, escolhido como sistema hospedeiro da ferramenta ComFIRM, e então aprofunda os aspectos mais importantes para este trabalho. Serão expostos não somente os pontos modificados pela ferramenta mas também os diversos mecanismos internos do *kernel* que foram utilizados durante a sua implementação, bem como aspectos estruturais importantes para a compreensão de seu funcionamento.

Para este trabalho foi escolhido trabalhar com o *kernel* versão 2.2.9, por ser esta a versão estável mais recente quando do início da atividade de implementação. Os arquivos referenciados neste capítulo se referem sempre ao diretório raiz dos fontes do *kernel* do Linux, salvo quando expressos em forma absoluta. Devido à natureza pouco intrusiva das alterações feitas e seu encapsulamento em arquivos criados especificamente para isso, a ferramenta poderá ser aplicada facilmente em versões mais atuais do Sistema Operacional Linux.

Pessoas com experiência em programação de funcionalidades para o núcleo do Linux podem pular este capítulo. Os mecanismos utilizados para a implementação da ferramenta são: manipulação de *socket buffers* (`sk_buffs`), implementação de arquivos virtuais no sistema de arquivos `proc`, a utilização de *wait queues* e temporizadores e a alteração de um *bottom-half handler*.

4.1 Introdução

Até há poucos anos, o Linux era considerado um sistema para amadores. Ele começou como um projeto de um estudante finlandês, Linus Torvalds, para entender como o hardware de gerenciamento de memória do processador Intel 386 realmente funcionava. Mais especificamente, Linus queria identificar quais passos detalhados eram necessários para construir um sistema de memória virtual prático com aquele sistema.

Como ele compartilhou seu código pela Internet, havia a possibilidade de outras pessoas fazerem suas contribuições para o sistema. Este processo tão simples de desenvolvimento tem estado em operação já há dez anos, e tem se mostrado um dos processos de desenvolvimento de software mais eficazes e eficientes que se tem notícia. Para se melhor perceber este processo, abaixo estão listadas algumas datas importantes do início do desenvolvimento do Linux [BEN 96]:

Janeiro, 1991. Linus Torvalds compra um 386;

Abril, 1991. Linus começa a se interessar por explorar o que o 386 consegue fazer com troca de contexto;

Julho, 1991. Linus pergunta sobre a especificação POSIX no newsgroup comp.os.-minix;

Outubro, 1991. Linux versão 0.02 anunciado em comp.os.minix;

Dezembro, 1991. Versão 0.11, provendo paginação sob demanda e suportando seu próprio desenvolvimento (podia executar o gcc);

Janeiro, 1992. Versão 0.12, já com memória virtual;

Agosto, 1992. Versão 0.97pl2, já com TCP/IP e Ethernet;

Primavera, 1993. Primeira distribuição comercial (Yggdrasil);

14 de março de 1994. Linux 1.0.

Embora o Linux tenha claramente começado como um projeto de estudantes, ele é hoje considerado uma base apropriada para muitos produtos comerciais, e desde o início da década de 1990 muitas empresas surgiram e fizeram fortunas baseadas exclusivamente no Linux. O Linux hoje em dia é o sistema operacional escolhido por milhões de usuários ao redor do mundo.

Em sua versão de desenvolvimento mais atual, o Linux suporta o maior número de protocolos de comunicação em um sistema operacional, o maior número de sistemas de arquivos, suporta multitarefa, multiprocessamento, multithreading, multiusuários, funciona em múltiplas plataformas, pode trabalhar com Terabytes de espaço de armazenamento e é visto como um dos sistemas operacionais mais robustos entre os disponíveis.

Com a grande aceitação que o Linux vem recebendo ultimamente, tanto no universo acadêmico quanto no comercial, existem inúmeras fontes de informação a seu respeito na Internet, em variados níveis de profundidade. Assim sendo, este capítulo tratará apenas daquilo que realmente importa a este trabalho, sem repetir informações introdutórias em excesso. Uma boa fonte de informação sobre o Linux, para aqueles que julgarem necessário, é o site `http://www.linuxdoc.org/`.

4.2 Arquitetura conceitual

Compreender a organização de um sistema complexo como o *kernel* do Linux é uma tarefa trabalhosa, porém que pode ser simplificada quando se estuda antes a estrutura de subsistemas que ele apresenta, e como estes subsistemas interagem. Sendo assim, esta seção apresenta a arquitetura conceitual do Linux, cujo propósito é exatamente fornecer um modelo mental de sua estrutura em geral [BOW 98a].

4.2.1 Visão geral de um sistema Linux

O *kernel* do Linux é inútil isoladamente; ele participa como parte de um sistema maior que, completo, é útil. Assim sendo, faz sentido discutir o *kernel* no contexto do sistema inteiro. A figura 4.1 mostra a decomposição do sistema Linux como um todo. Este sistema é composto de quatro principais camadas:

Aplicações: o conjunto de aplicações em uso em um dado sistema Linux;

Serviços do SO: são os serviços que tipicamente são considerados parte do sistema operacional, como sistema de janelas, interpretador de comandos, etc;

Kernel do Linux: este é a área de atenção desta seção; o *kernel* abstrai e media o acesso ao hardware, inclusive o processador;

Hardware: esta camada é composta por todos os possíveis dispositivos físicos em uma instalação de Linux.

FIGURA 4.1 – Decomposição de um Sistema Linux em camadas principais

O *kernel* do Linux apresenta uma interface virtual ao hardware para processos de usuários. Processos são escritos sem precisar nenhum conhecimento de que hardware físico está instalado no computador — o *kernel* do Linux abstrai todos os tipos de hardware em uma única interface virtual. Em adição a isso, o Linux suporta multitarefa de uma maneira completamente transparente aos processos: cada processo pode agir como se fosse o único no sistema, com uso exclusivo da memória e dos dispositivos de hardware. O *kernel* na verdade executa vários processos simultaneamente, e é responsável por mediar o acesso ao hardware de forma a dar a cada processo uma porção justa deste.

4.2.2 Visão geral da estrutura do *kernel*

O *kernel* do Linux é composto de cinco principais subsistemas:

Escalonador: é responsável por controlar o acesso dos processos ao processador. O escalonador aplica uma política que assegura que todos os processos terão acesso justo ao processador, enquanto garante também que ações necessárias de hardware são feitas a tempo pelo *kernel*;

Gerenciador de Memória: permite que múltiplos processos seguramente compartilhem a memória física do computador. Ainda mais, ele permite que dispositivos de armazenamento sejam usados como memória secundária para estender a capacidade total de memória do sistema;

Virtual Filesystem Switch (VFS): abstrai os detalhes de acesso aos dispositivos de hardware, fornecendo uma interface de arquivos comum para todos eles. Também abstrai os vários tipos de sistemas de arquivos utilizados, locais ou remotos;

Subsistema de Rede: provê acesso a uma variedade de protocolos de comunicação sobre um grande número de dispositivos de rede;

Comunicação Inter-Processo: suporta vários mecanismos de interação e comunicação entre processos em um sistema Linux.

A figura 4.2 mostra uma decomposição de alto nível do *kernel* do Linux, onde linhas são traçadas de subsistemas dependentes a subsistemas dos quais dependem. Esta figura enfatiza que o subsistema mais central é o escalonador de processos:

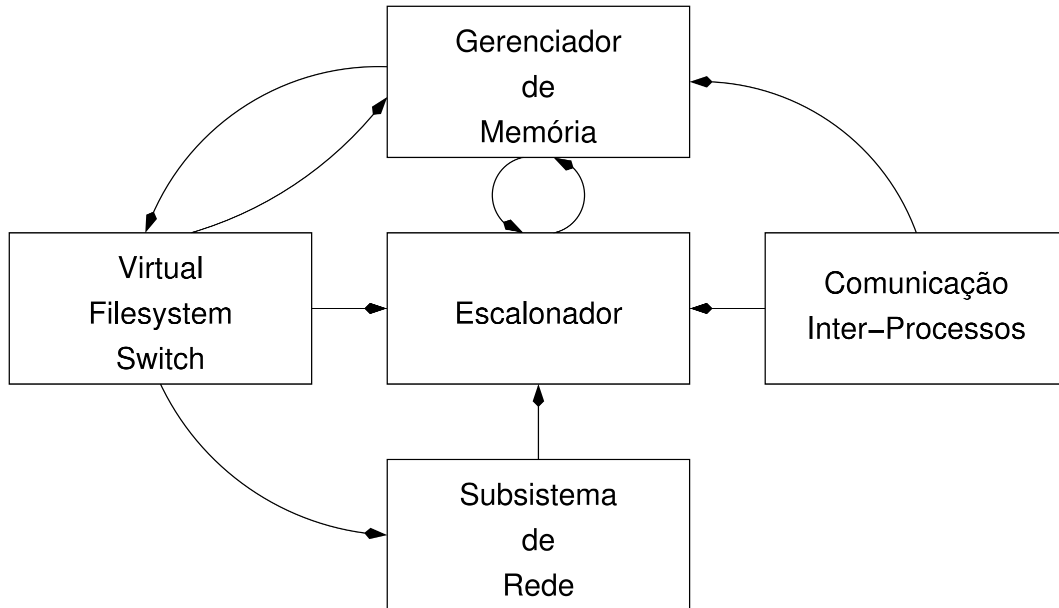

FIGURA 4.2 – Decomposição do *kernel* de um sistema Linux

todos os outros subsistemas dependem do escalonador, pois todos precisam utilizá-lo para suspender e reativar processos. Normalmente, um subsistema vai suspender um processo que está esperando que uma operação de hardware termine, e reativar o processo quando esta estiver pronta.

Por exemplo, quanto um processo tenta enviar uma mensagem pela rede, o subsistema de rede vai suspender o processo até que o hardware termine de enviar a mensagem com sucesso. Depois que a mensagem for enviada, ou o hardware sinalizar um erro, o subsistema de rede reativa o processo e retorna um código de erro indicando o sucesso ou falha da operação. Os demais subsistemas dependem do escalonador por motivos semelhantes.

As outras dependências são menos óbvias, mas igualmente importantes:

- O escalonador de processos utiliza o gerenciador de memória para restaurar o estado de um determinado processo quando este é reativado;
- O subsistema de comunicação inter-processo depende do gerenciador de memória para suportar o mecanismo de comunicação por memória compartilhada;
- O Virtual Filesystem Switch utiliza o subsistema de rede para suportar sistemas de arquivos de rede, como NFS, e o gerenciador de memória para prover dispositivos do tipo ramdisk;
- O gerenciador de memória utiliza o Virtual Filesystem Switch para suportar memória virtual (swap). Esta é a única razão pela qual depende do escalonador de processos: quando um processo acessa um endereço que não está presente na memória física, o gerenciador de memória faz uma requisição ao VFS para buscar a página de memória referenciada de algum dispositivo de armazenamento, e precisa suspender o processo.

Em adição às dependências explicitamente demonstradas, todos os subsistemas dependem de alguns recursos que não pertencem a nenhum subsistema. São funções

de apoio que todos usam, como procedimentos para imprimir avisos ou mensagens de erro e rotinas de depuração. Como são funções bastante genéricas, não serão citadas nesta revisão.

O escalonador mantém uma estrutura de dados para cada processo em execução. Estas estruturas estão organizadas como uma lista, chamada `task list`, e o escalonador sempre mantém um ponteiro chamado `current` que indica qual processo está ativo no momento.

O gerenciador de memória armazena um mapeamento entre endereços virtuais e físicos para cada cada processo em execução, e também informações sobre como buscar e substituir determinadas páginas. Estas informações são encapsuladas em estruturas chamadas `mm_struct`, que são ligadas à estrutura `task_struct` de cada processo.

O VFS utiliza *i-nodes* para representar os arquivos em um sistema de arquivos. A estrutura de dados `inode` armazena o mapeamento entre números de blocos do arquivo e blocos físicos do disco. Estas estruturas podem ser compartilhadas por dois processos, caso os dois mantenham o mesmo arquivo aberto. Este compartilhamento se dá de maneira que as duas estruturas `task_struct` (dos dois processos) apontam para a mesma estrutura `inode`.

Todas estas estruturas estão ligadas às estruturas `task_struct` que compõem a `task list`. Além das já citadas, existem também para cada processo estruturas para suas conexões de rede e mecanismos de comunicação inter-processo em uso.

4.2.3 Arquitetura dos subsistemas

Nesta seção serão abordados somente os subsistemas importantes à compreensão da implementação da ferramenta ComFIRM, para tornar o texto mais objetivo. Serão abordados primeiramente o Virtual Filesystem Switch, do qual nos interessa o sistema de arquivos `proc`, e então o subsistema de rede, onde será visto sua organização em camadas e seus mecanismos de funcionamento.

Virtual Filesystem Switch. Este subsistema é projetado para apresentar uma visão consistente dos dados gravados em dispositivos de armazenamento. Quase todos os dispositivos de hardware em um computador são representados utilizando uma interface genérica de gerenciador de dispositivo. O VFS vai ainda mais longe permitindo que se monte qualquer conjunto de sistemas de arquivos lógicos em qualquer dispositivo físico. Sistemas de arquivos lógicos promovem compatibilidade entre padrões de diferentes Sistemas Operacionais, e permitem que desenvolvedores criem sistemas de arquivos com políticas diferentes.

O VFS abstrai os detalhes tanto de diferentes sistemas de arquivos lógicos como de diferentes dispositivos físicos, e permite que processos de usuários acessem arquivos utilizando uma interface comum, sem necessariamente saber em qual dispositivo físico ou lógico o arquivo reside. Outra característica muito importante do VFS é suportar sistemas de arquivos virtuais, que não existem como estruturas em um dispositivo de armazenamento, mas sim como estruturas em memória, que representam algum estado do sistema que seja interessante para processos de usuário. Dois exemplos destes sistemas são o sistema de arquivos `proc` e o `devfs`. A figura 4.3 mostra a interação do VFS com o restante do sistema.

O `proc` é um sistema de arquivos que apresenta inúmeras informações sobre o estado do sistema, obtidas de estruturas de dados do *kernel*, que são temporariamente transformadas em arquivos virtuais quando é feito um acesso a algum arquivo

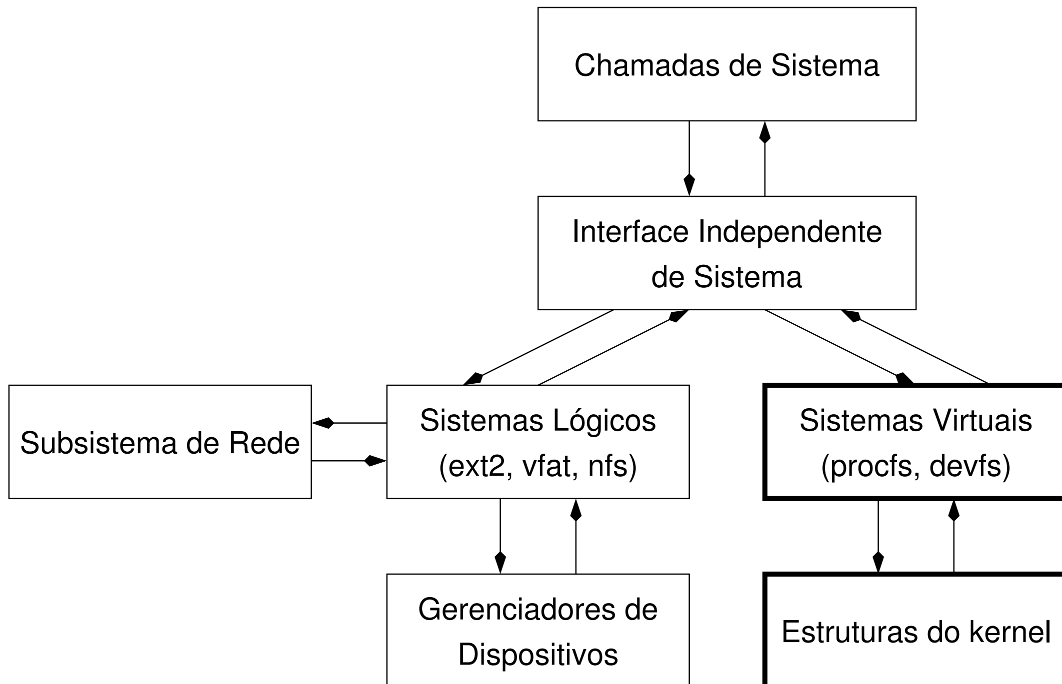

FIGURA 4.3 – Interação do VFS com o restante do sistema

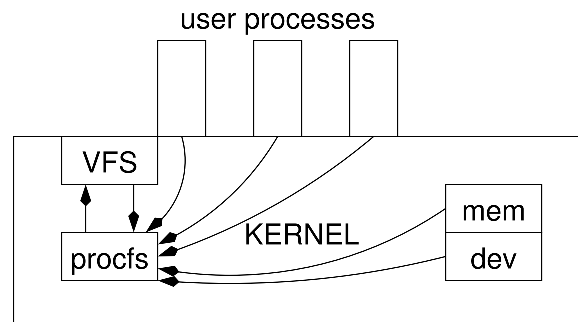

FIGURA 4.4 – Relação do sistema de arquivos `proc` com o restante do *kernel*

deste sistema de arquivos. A figura 4.4 representa a obtenção destes dados, oriundo dos mais diversos pontos do sistema. Estes dados não são apenas coletados e apresentados, pois muitos destes arquivos suportam inclusive operações de escrita, afetando portanto alguma característica do *kernel*, como o número máximo de arquivos abertos por processo, ou o algoritmo escalonador sendo utilizado.

O sistema de arquivos experimental `devfs` visa terminar com a confusão na gerência de arquivos especiais que representam dispositivos em um sistema Unix. Qualquer sistema Linux ou Unix apresenta em seu diretório `/dev` centenas de arquivos especiais, cada um apontando para um dispositivo que pode ser suportado pelo *kernel* e pode estar presente no sistema em uso. O problema é que normalmente apenas uma pequena fração destes arquivos é realmente usada, e nada impede que um gerenciador de dispositivo seja carregado mas não exista nenhum arquivo especial apontando para ele.

Estes problemas são resolvidos pelo `devfs` na medida que todos os gerenciadores de dispositivos, quando inicializados, registram no `devfs` arquivos especiais

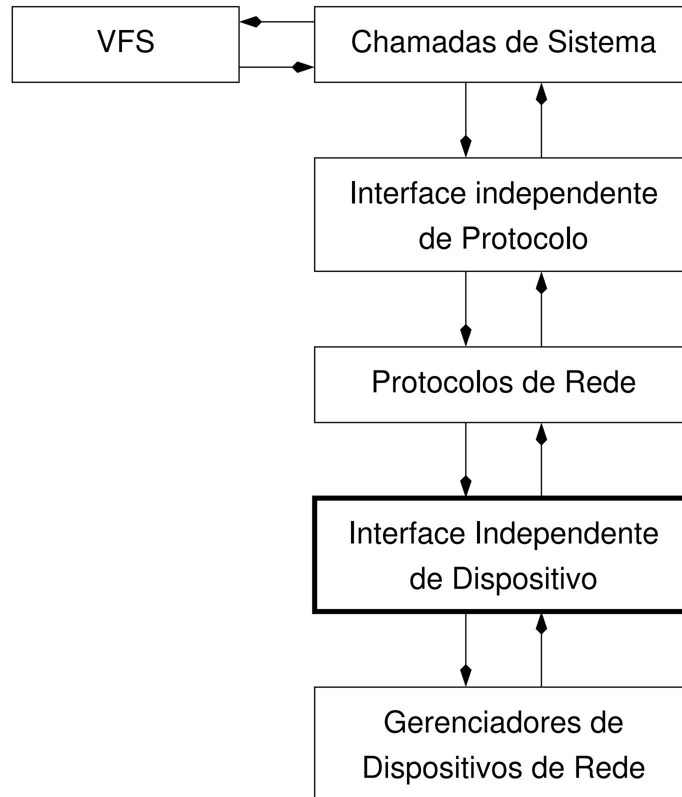

FIGURA 4.5 – Disposição em camadas do subsistema de rede

apontando para exatamente aquilo que foi detectado. Se o diretório `/dev` for deixado vazio e o `devfs` for montado sobre ele, arquivos especiais serão criados e excluídos à medida que os gerenciadores forem carregados e descarregados. Com isso, os arquivos presentes no `/dev` sempre apresentarão uma visão coerente daquilo que realmente existe no sistema, e não todas as suas inúmeras e talvez até incorretas possibilidades.

Subsistema de Rede. Este subsistema permite que sistemas Linux se comuniquem com outros sistemas através de uma rede. Existem inúmeros dispositivos de hardware suportados, e vários protocolos de comunicação que podem ser utilizados. O subsistema de rede abstrai ambos detalhes de implementação (dispositivos e protocolos), de forma que processos de usuários e outros subsistemas do *kernel* possam acessar a rede sem necessariamente conhecer que dispositivo específico ou protocolo está sendo usado. A figura 4.5 mostra a organização em camadas do subsistema de rede.

Em sua organização, são particularmente interessantes sua conexão com o Virtual Filesystem Switch e a camada de abstração de dispositivo. A ligação com o VFS se dá na forma de chamadas de sistema `read` e `write` virtualizadas, que tanto podem ler e escrever em um arquivo quanto em uma conexão de rede. Neste caso, a chamada é passada transparentemente para o subsistema de rede. Ainda mais, pelo projeto orientado a objetos do Linux, no momento da criação de uma conexão (um *socket*), o subsistema de redes automaticamente passa, para o *inode* que descreve esta conexão, uma estrutura com funções de arquivo (como `read`, `write` e `close`) ligadas ao subsistema de rede; o VFS chamará estas funções sem nem precisar saber que se trata de uma conexão, ao invés de um arquivo.

A camada de abstração de dispositivo existe para que os protocolos de comu-

nicação não precisem saber dos detalhes de operação de cada dispositivo. Isto é conseguido através de duas rotinas que concentram toda a atividade de transmissão e recepção de pacotes de rede. Estas rotinas gerenciam as filas de todos os dispositivos, e ao receber um pacote verificam automaticamente por qual dispositivo devem ser enviados ou a qual protocolo devem ser entregues. As rotinas `dev_queue_xmit` e `netif_rx` são portanto um ponto bastante interessante para a inserção de código que vise monitorar ou influenciar a capacidade de comunicação de um sistema Linux.

4.3 Arquitetura Concreta

Esta seção traz mais detalhes de implementação dos subsistemas do *kernel* do Linux mais interessantes a este trabalho. Novamente, são vistos o Virtual Filesystem Switch (em especial o sistema de arquivos `proc`) e o subsistema de rede. Já tendo sido estudada a arquitetura conceitual do Linux, esta seção traz agora sua arquitetura concreta [BOW 98b], extraída diretamente de seu código.

4.3.1 O sistema de arquivos `proc`

O sistema de arquivos `proc` surgiu da necessidade de se reportar dados do *kernel* para programas de usuário. Antes de sua existência, os programas tinham que examinar a memória vista pelo *kernel*, que era mapeada em um dispositivo `/dev/kmem`, e utilizar a tabela de símbolos do *kernel* para decodificar seu conteúdo. Isto se tornou obsoleto por subrotinas de acesso que simulam um sistema de arquivos para qualquer processo de usuário.

Embora os dados venham de estruturas do *kernel* ao invés de blocos de armazenamento em disco, o processo vê os dados como texto estruturado em um arquivo (figura 4.4). Por exemplo, o comando `cat /proc/version` revela o número de versão e a hora em que foi compilado o *kernel* em execução. Este sistema de arquivos passa por constantes atualizações. Algumas das mais atuais tornaram muito mais fácil adicionar novos conteúdos a este sistema de arquivos. Uma mudança visível é que agora o usuário `root` pode escrever em alguns destes arquivos para causar mudanças internas ao *kernel* [BEN 96].

Atualmente, muitas partes do *kernel* são completamente configuráveis através de hierarquias de arquivos e diretórios virtuais presentes no sistema de arquivos `proc`. Arquivos e diretórios podem ser criados dinamicamente, portanto mesmo módulos de código inseridos em tempo de execução podem utilizar este recurso. Por exemplo, o suporte a roteamento IP pode ser habilitado ou desabilitado através da escrita de `1` ou `0` no arquivo `/proc/sys/net/ipv4/ip_forward`. Informações tão diversas quanto o número de interrupções já acionadas por cada periférico sendo utilizado e as regiões de entrada e saída alocadas a cada um podem ser encontradas neste sistema de arquivos.

Conteúdo do sistema de arquivos. Embora o conteúdo do sistema de arquivos `proc` seja dinâmico, alguns arquivos e diretórios estão sempre presentes, por serem muito importantes para o funcionamento de várias ferramentas de administração. Os arquivos fixos do sistema de arquivos `proc` são os seguintes [CAR 98b]:

cmdline: argumentos passados ao *kernel* quando da inicialização do sistema;

cpuinfo: descrição das características dos processadores detectados;

devices: lista de controladores de dispositivos incluídos no sistema;

dma: lista dos canais de DMA usados por controladores de dispositivos;

filesystems: lista dos sistemas de arquivos suportados pelo *kernel*;

interrupts: lista das interrupções de hardware usadas pelos controladores de dispositivos;

ioports: lista de portas de entrada e saída usadas pelos controladores de dispositivos;

kcore: memória utilizada pelo *kernel* (pode ser diferente da memória total, se forem usados dispositivos que utilizam a memória do sistema, como algumas controladoras de vídeo AGP);

kmsg: últimas mensagens geradas pelo *kernel*;

ksyms: lista de símbolos do *kernel* usados pelos módulos;

loadavg: o número médio de processos em execução no último minuto;

locks: lista de *locks* em arquivos;

meminfo: estado da memória do sistema (alocações, *swap*, etc);

modules: lista de módulos inseridos no *kernel*;

mounts: lista de sistemas de arquivos montados;

pci: lista de periféricos conectados aos barramentos PCI;

rtc: informações sobre o relógio de tempo real;

stat: várias estatísticas sobre operações executadas pelo *kernel*, como consumo de tempo de processador, número de operações de disco, número de interrupções processadas, etc;

smp: informações sobre operações de multiprocessamento;

uptime: tempo decorrido desde a carga do sistema;

version: versão do *kernel* em execução.

Em adição a estes arquivos, estão presentes neste sistema de arquivos os seguintes diretórios:

net: árvore contendo arquivos de informações sobre protocolos de comunicação;

scsi: arquivos contendo informações de controle sobre dispositivos SCSI;

sys: árvore de arquivos contendo informações ligadas à variáveis do *kernel*, gerenciadas também pela primitiva *sysctl*;

um diretório por processo: o nome de cada diretório é o número de identificação do processo (PID) que ele representa. Cada diretório contém os seguintes arquivos:

cmdline: lista de argumentos do processo;

cwd: *link* para o diretório atual do processo;

environ: lista de variáveis de ambiente;

exe: *link* para o arquivo binário executado pelo processo;

fd: diretório contendo *links* para os arquivos abertos pelo processo;

maps: lista de zonas de memória alocadas ao processo;

mem: conteúdo do espaço de endereçamento do processo;

root: *link* para o diretório raiz do processo;

stat, statm, status: estado do processo.

self: *link* para o diretório que corresponde ao processo que fez a chamada.

Implementação do sistema de arquivos `proc`. Os itens no diretório `/proc` (arquivos ou diretórios) são gerenciados dinamicamente: uma lista de descritores é mantida em memória pelo *kernel*, e seu conteúdo é varrido quando da realização de alguma operação neste diretório.

A estrutura `proc_dir_entry`, definida no arquivo `<linux/proc_fs.h>`, representa o tipo descritor. Ela contém os seguintes campos:

low_ino: número do *inode* atribuído ao ítem;

namelen: tamanho do nome do ítem;

name: nome do ítem;

mode: permissões de acesso;

nlink: número de referências a este ítem;

uid: identificação do usuário dono do ítem;

gid: identificação do grupo dono do ítem;

size: tamanho em bytes;

ops: operações ligadas a este ítem;

get_info: ponteiro para a função de leitura;

fill_inode: ponteiro para a função que inicializa a estrutura;

next, parent, subdir: ponteiros utilizados na estruturação da árvore de diretórios;

data: dados privados deste ítem.

Os números dos *inodes* são alocados estaticamente aos itens no `/proc` (os itens não são criados ao acaso como em um sistema de arquivos comum). O arquivo de cabeçalho citado acima define vários tipos para este propósito:

root_directory_inos: números dos *inodes* atribuídos aos itens do diretório raiz do sistema de arquivos, na faixa de 1 a 127;

net_directory_inos: números dos *inodes* dos itens do diretório `/proc/net`, na faixa de 128 a 255;

scsi_directory_inos: números para o diretório `/proc/scsi`, na faixa de 256 a 511.

Ainda mais, as constantes `PROC_DYNAMIC_FIRST` e `PROC_NDYNAMIC` definem números de *inodes* que devem ser alocados dinamicamente.

Números de *inodes* alocados aos diretórios correspondentes a processos são calculados tendo como base o número da identificação do processo. Este número é deslocado em 16 bits à esquerda para gerar o número base, e o tipo `pid_directory_inos` define um número a ser somado à base para obter o número final do *inode*. Por exemplo, o número do *inode* do arquivo `root` contido no diretório atribuído ao processo de identificação p será $p * 65536 + 6$, já que a constante `PROC_PID_ROOT` tem o valor 6.

O arquivo `fs/proc/root.c` contém as funções que permitem o gerenciamento dos conteúdos deste sistema de arquivos. A variável global `proc_root` contém o diretório raiz do sistema de arquivos. A função `proc_register` cria um novo ítem em um diretório, e a função `proc_unregister` remove um ítem retirando seu descritor da lista. A função `proc_register_dynamic` atribui um número de *inode* dinâmico a um descritor e o insere na lista.

A inicialização da lista de arquivos e diretórios localizados no `/proc` é executada pela função `proc_root_init`. Esta função chama a função `proc_register` para todas as entradas fixas contidas no diretório raiz deste sistema de arquivos. Cada descritor contém ponteiros para suas funções de leitura e escrita, portanto cada arquivo pode retornar as mais diferentes informações sobre o *kernel* em execução.

4.3.2 Implementação do Subsistema de Rede

Assim como os próprios protocolos de comunicação em rede, o Linux implementa o suporte a protocolos como uma série de camadas de software conectadas (figura 4.6). A interface BSD *sockets* é suportada através de um software de gerenciamento genérico, que apresenta as chamadas de BSD *sockets* para as aplicações. Por baixo desta camada estão os gerenciadores de protocolos como TCP, UDP e IPX. Este último produz pacotes de rede completos, portanto está ligado diretamente à última camada, que é a dos dispositivos de conexão à rede. Os protocolos TCP e UDP passam ainda pela camada IP, que então os entrega aos dispositivos de rede.

A interface BSD *sockets* não apenas suporta várias formas de comunicação em rede, mas também é um mecanismo de comunicação interprocesso. Um *socket* descreve uma ponta de um canal de comunicação; dois processos em comunicação terão um *socket* cada um, que descreverá a conexão para o processo. O Linux suporta várias classes de *sockets*, e estas são conhecidas como *address families*. Isto porque cada classe tem seu próprio método de gerenciar suas comunicações. O Linux suporta as seguintes famílias de protocolos ou domínios [RUS 97]:

UNIX *Sockets* locais para comunicação interprocessos;

INET A família que suporta os protocolos TCP/IP;

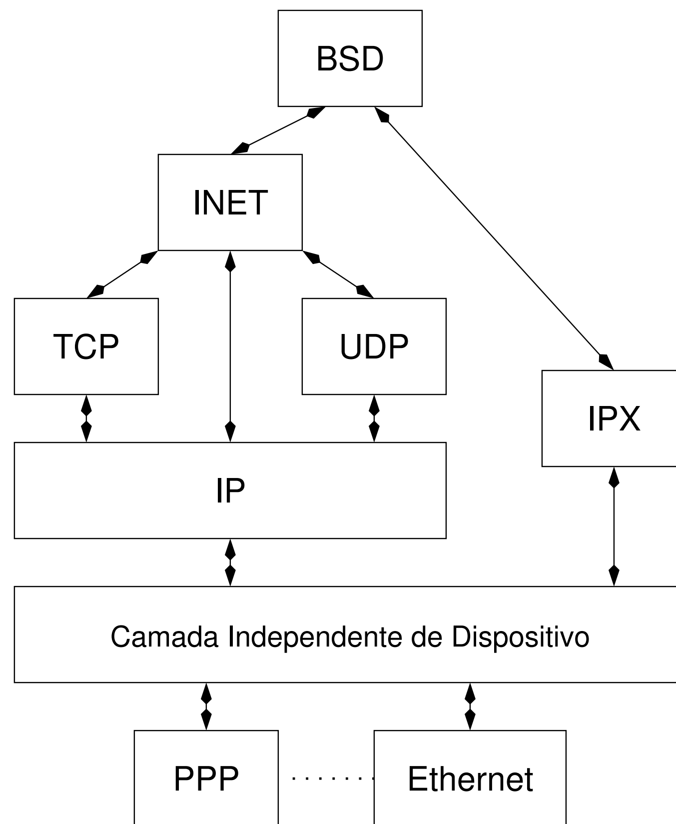

FIGURA 4.6 – Organização em camadas dos protocolos de comunicação suportados

AX25 Uma família de protocolos para radioamadores;

IPX O protocolo Novell IPX;

APPLETALK Appletalk DDP;

X25 Família X25.

Dentro destas famílias, existem vários tipos de *sockets*, que representam o tipo de serviço suportado pela conexão. Nem todas as famílias suportam todos os tipos de serviço. Os tipos suportados pela camada BSD *sockets* do Linux são os seguintes:

Stream Estes *sockets* suportam comunicação bidirecional seqüencial, com a garantia que os dados não serão perdidos, corrompidos ou duplicados em trânsito. *Sockets stream* são suportados pelo protocolos TCP na família INET;

Datagram Estes *sockets* também proporcionam comunicação bidirecional mas, diferentemente dos anteriores, não garantem a entrega das mensagens. Mesmo que entregues, não há garantia de ordem, duplicação ou corrupção. Na família INET é suportado pelo UDP;

Raw São *sockets* que permitem acesso direto às camadas inferiores possibilitando, por exemplo, que se receba pacotes IP diretamente;

Reliably Delivered Messages São como Datagram, porém com a garantia de entrega. Não são correntemente implementados em nenhuma família.

Sequenced Packets São como Stream, porém com o tamanho dos pacotes de dados fixo;

Packet Não é um tipo padrão do BSD *sockets*, mas sim uma extensão específica do Linux para permitir acesso direto ao nível de hardware.

O exato significado das operações em um *socket* BSD depende da família de protocolos da qual faz parte. Configurar uma conexão TCP/IP é bastante diferente de configurar uma conexão AX.25 via rádio. O Linux abstrai as diferenças entre as famílias de protocolos através da interface BSD *sockets*, que por sua vez é suportada por todas as famílias para implementar suas funcionalidades.

A estrutura sk_buff. Um dos problemas de se ter muitos protocolos de rede organizados em camadas, cada um usando serviços de outros, é que cada protocolo precisa adicionar cabeçalhos e rodapés[1] aos dados à medida que são transmitidos e removê-los durante o processamento de dados recebidos. Isto torna difícil passar *buffers* de dados entre os vários protocolos porque cada protocolo precisa encontrar onde seus cabeçalhos e rodapés estão. Uma solução é copiar os *buffers* em cada camada, porém isto é ineficiente. Ao invés disso, o Linux usa *socket buffers* ou `sk_buffs` para passar dados entre as camadas de protocolos e os *drivers* de dispositivos de rede [RUS 97].

`sk_buffs` contém campos de tamanho e ponteiros que permitem a cada protocolo manipular os dados através de funções ou métodos padronizados. Cada `sk_buff` contém um bloco de dados associado, e um conjunto de quatro ponteiros, que são usados para manipular estes dados (figura 4.7). São os seguintes:

[1] De *headers* e *tails*.

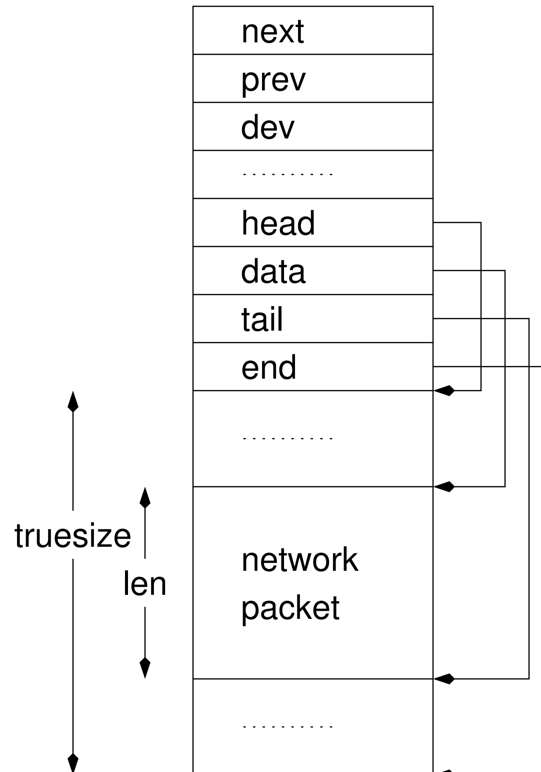

FIGURA 4.7 – Estrutura de um *socket buffer*

head: Aponta para o início da área de dados na memória. É fixado quando da alocação do `sk_buff`;

data: Aponta para o atual início da área de dados. Este ponteiro varia dependendo do protocolo que atualmente contém o `sk_buff`;

tail: Aponta para o final atual da área de dados. Assim como o `data`, varia de acordo com o protocolo que contém o `sk_buff`;

end: Aponta para o final da área de dados na memória. É também fixado quando o `sk_buff` é alocado.

Existem dois campos de tamanho, `len` e `truesize`, que descrevem o tamanho do pacote no protocolo atual e o tamanho da área de memória total alocada. O código de manuseio de `sk_buffs` proporciona mecanismos padronizados para a adição e remoção de cabeçalhos e rodapés de protocolos à área de dados. As funções detalhadas a seguir manipulam com segurança os campos `data`, `tail` e `len`[2].

skb_push: Move o ponteiro `data` em direção ao início da área de dados alocada e incrementa o campo `len`. É utilizada na adição de cabeçalhos e dados ao início da mensagem a ser transmitida;

skb_pull: Realiza o contrário da função anterior, decrementando o campo `len` e movendo o ponteiro `data` em direção ao final da área de dados. Tem a função de retirar cabeçalhos das mensagens;

[2]As funções estão definidas em `include/linux/skbuff.h`.

skb_put: Move o ponteiro `tail` em direção ao final da área de dados, incrementando o campo `len`. Permite a adição de dados ao final da mensagem;

skb_trim: Realiza o contrário da função anterior, retirando dados do final da mensagem sendo processada;

skb_reserve: Move os dois ponteiros em direção ao final da área de dados, reservando espaço no início do `sk_buff`. Deve ser chamada antes de se começar a preencher a mensagem, já que não copia os dados ao mover os ponteiros.

A estrutura `sk_buff` contém também ponteiros que são usados quando mensagens são armazenadas em listas circulares duplamente encadeadas durante seu processamento. Existem rotinas genéricas para adicionar `sk_buffs` ao início ou final destas listas e para removê-las. Listas são descritas através da estrutura `sk_buff_head`. As funções mais importantes na manipulação destas filas são as seguintes[3]:

skb_queue_head: Insere um `sk_buff` no início de uma lista;

skb_queue_tail: Insere um *buffer* no final de uma lista;

skb_dequeue: Remove um `sk_buff` de uma lista. Esta rotina é atômica, portanto pode ser chamada quando outros processos estão adicionando pacotes à lista;

skb_insert: Insere um pacote antes de outro em uma lista;

skb_append: Coloca um pacote depois de outro em uma lista;

skb_unlink: Remove um `sk_buff` de sua lista. Funciona de maneira diferente de `skb_dequeue` na medida que recebe apenas o *socket buffer* a ser retirado. `skb_dequeue`, por sua vez, recebe a lista e retorna seu primeiro pacote;

kfree_skb: Libera a área de memória de um *socket buffer*;

alloc_skb: Aloca memória para um novo `sk_buff`;

skb_clone: Duplica um *socket buffer*, criando duas áreas de controle que apontam para a mesma área de dados;

skb_copy: Retorna uma cópia de um `sk_buff`, com áreas de dados independentes. É mais lento pela cópia dos dados.

Envio de mensagens. Para esta descrição, é assumido que dois computadores têm entre si uma conexão TCP/IP já estabelecida. Em um computador, o processo dono da conexão faz uma chamada à função `write(socket,data,length);`, que acaba por chamar a função `sys_write` do *kernel*, uma rotina do sistema de arquivos. Esta chamada de sistema realiza uma série de testes de validade, como por exemplo se a área de dados passada como parâmetro é válida, se existe o descritor de arquivo apontado por `socket`, e assim por diante.

Passados estes testes, o código da chamada chama a rotina de escrita pertencente a este descritor de arquivo (o Linux apresenta um projeto orientado a objetos,

[3]Descritas em `include/linux/skbuff.h` e `net/core/skbuff.c`.

e o sistema de arquivos é um dos pontos onde isto se torna mais claro). No caso de um *socket*, esta rotina é a `sock_write`, que apenas realiza algumas funções administrativas. Ela procura pela estrutura *socket* ligada a este descritor de arquivo, e os parâmetros da operação de escrita são transferidos a uma estrutura de mensagem (`sk_buff`).

Como esta conexão pertence à família `INET` (que implementa o conjunto de protocolos TCP/IP), a rotina `sock_write` chama a função de transmissão `inet_sendmsg`, para a qual passa a estrutura do *socket*, da mensagem, e algumas *flags* específicas desta conexão. A função `inet_sendmsg` detecta através dos dados de conexão que se trata de uma mensagem do protocolo TCP, e redireciona os dados recebidos para a função `tcp_sendmsg`.

Uma série de outras funções são chamadas para construir cabeçalhos e calcular *checksums*, e finalmente é chamada a rotina `ip_queue_xmit`, que coloca a mensagem em uma fila de mensagens prontas para transmissão. Esta última função adiciona alguns valores que somente podem ser calculados agora, como o *checksum* do cabeçalho IP. Ela então passa o pacote pronto para a rotina `dev_queue_xmit`. Esta rotina recebe todos os pacotes prontos, provenientes de todos os protocolos de transporte (não apenas IP), e os envia aos dispositivos corretos [BEC 98].

A implementação do protocolo IPX, por exemplo, possui apenas uma camada entre a interface com os processos (via BSD *sockets*) e a rotina `dev_queue_xmit`, que envia os pacotes para os dispositivos de rede. O importante é que todo o tráfego de rede passa por esta última rotina, o que torna ela um alvo bastante interessante para inserção de uma ferramenta de injeção de falhas. Esta rotina é definida em `net/core/dev.c`.

Recepção de mensagens. Quando uma interface de rede devidamente configurada recebe uma mensagem, ela gera uma interrupção, que é tratada por uma rotina específica a este dispositivo. Esta rotina é responsável por transferir a mensagem da memória do dispositivo para uma estrutura de mensagem na memória principal, e então chamar a rotina que gerencia a recepção de todos os pacotes de rede de todos os dispositivos, `netif_rx`.

Esta rotina, que centraliza o processamento da recepção de mensagens, é chamada em um contexto de interrupção, portanto não pode demorar demasiadamente. Para que o *kernel* possa realizar um processamento mais demorado da mensagem sem demorar demais a retornar da interrupção, esta rotina apenas inclui a mensagem recebida em uma lista de pacotes a processar, e utiliza um mecanismo chamado *bottom half handler* para sinalizar a necessidade de processamento posterior desta lista. Este mecanismo será mais aprofundado nas próximas seções.

Quando o *bottom half handler* entra em ação, ele retira pacotes da lista, chamada *backlog*, e os processa de acordo com o protocolo a que pertence. Como o Linux mantém uma lista de protocolos suportados e seus identificadores, basta percorrer a lista procurando a rotina a ser chamada para cada protocolo.

A rotina `netif_rx` não deve ser alterada, pois ela é executada em contexto de interrupção. Neste contexto, as ações devem ser atômicas e bastante rápidas, de forma a ter um mínimo impacto no desempenho do sistema. Já o *bottom half handler* do subsistema de rede não tem nenhum aspecto crítico em seu contexto de execução, sendo o local ideal para a inclusão de código como uma ferramenta de injeção de falhas. A rotina que implementa o *bottom half handler* é a `net_bh`, no arquivo `net/core/dev.c`.

4.4 Programando em modo *kernel*

Quando se está alterando ou adicionando funcionalidade ao *kernel* do Linux, deve-se ter uma série de cuidados especiais. De início, mesmo um experiente programador em linguagem C deve ter cuidado porque, ainda que se esteja usando uma linguagem com funções e bibliotecas padronizadas, um *kernel* de sistema operacional funciona em um ambiente muito diferente daquele de um programa comum, e a primeira diferença que se nota é que a rica biblioteca de funções da linguagem C não está presente.

Para maior comodidade dos programadores, algumas funções mais utilizadas e importantes mesmo no contexto do *kernel*, como funções de manipulação de *strings*, são implementadas através de uma pequena biblioteca estática de funções de auxílio para todas as outras funções do *kernel*. O diretório `lib` contém os arquivos que implementam estas funções auxiliares.

O Linux não foi projetado em um quadro-negro, mas sim desenvolvido de forma evolucionária, e continua a se desenvolver. Cada função do *kernel* já foi repetidamente alterada e expandida para a remoção de erros e adição de novas características. Qualquer pessoa que já se envolveu pessoalmente em um grande projeto deste tipo sabe quão rápido o código fonte pode se tornar impossível de acompanhar e propenso a erros. Face a essa questão, Linus Torvalds, como coordenador do Projeto Linux, tem conseguido manter o *kernel* organizado de maneira a ser fácil de entender e constantemente removido idéias e práticas antigas que não mais se aplicam [BEC 98].

A despeito disso, o *kernel* do Linux não é de nenhuma forma um bom exemplo de programação estruturada. Existem "números mágicos" espalhados pelo código, ao invés de constantes declaradas em arquivos de cabeçalho, funções do tipo *inline*, ao invés de chamadas, instruções `goto`, ao invés do simples `break`, instruções em *assembly*, ao invés de código em C, e muitas outras características tão deselegantes quanto as citadas. Muitas destas características distintas da programação não-estruturada, contudo, foram deliberadamente incluídas. Muitas partes do *kernel* são críticas em termos de tempo, portanto o código é otimizado para um bom comportamento durante a execução, ao invés de boa legibilidade.

Visto a partir da perspectiva de um processo sendo executado no Linux, o *kernel* é um provedor de serviços. Processos individuais existem independentemente lado a lado, e não podem se afetar diretamente. A área de memória de cada processo é protegida de alterações por parte de outros processos.

O ponto de vista interno de um sistema Linux em funcionamento é bastante diferente. Apenas um programa — o sistema operacional — está sendo executado no computador, e pode acessar todos os recursos. As várias tarefas são desempenhadas por co-rotinas, ou seja, cada tarefa decide por si mesma se e quando passar o controle para outra tarefa. Uma conseqüência disso é que um erro de programação no *kernel* pode bloquear todo o sistema. Qualquer tarefa pode acessar todos os recursos de outras tarefas e modificá-los.

Certas partes de uma tarefa são executadas no modo menos privilegiado do processador, chamado modo usuário. Estas partes da tarefa se parecem do lado externo com processos (para alguém olhando de fora para dentro do *kernel*). Do ponto de vista destes processos, o sistema proporciona multitarefa real [BEC 98].

Alguns pontos devem ser observados nos algoritmos usados dentro do *kernel*.

Por exemplo, uma chamada de sistema de leitura de arquivo não pode simplesmente fazer a requisição ao disco e esperar a resposta. Enquanto o *kernel* espera, todos os processos esperam também, não há possibilidade de um processo interromper outro que está executando em modo privilegiado. Todos os laços empregados devem observar a possibilidade de não terminarem. Devem ser empregadas listas de processos que estão esperando por algum evento, como por um setor a ser lido de um disco. Quando uma tarefa do *kernel* detecta que uma operação pode demorar, ela deve automaticamente se inserir em uma fila de espera e entregar o processador a outra tarefa. Para ajudar nestas tarefas, o Linux implementa uma série de mecanismos, que serão estudados na próxima seção.

4.5 Mecanismos do *kernel*

O Linux proporciona mecanismos de suporte a funções que devem ser realizadas muito brevemente, porém não imediatamente. Por exemplo, o tratamento de uma interrupção pode ser rápido, ocasionando apenas algo como a leitura de um caracter do teclado. Mas também pode ser algo como ler e processar uma mensagem recebida através da rede, uma atividade claramente mais complexa. Através do mecanismo de *bottom half handlers*, atividades como este último exemplo podem ser divididas em um tratamento inicial, que necessita ser atômico, e uma atividade secundária, que deve ser executada logo mas não no contexto atual. A transferência da mensagem da placa de rede para a memória principal é a parte mais atômica e essencial. Processar esta mensagem e redirecioná-la para o gerenciador de protocolo adequado é uma atividade secundária, que pode ser executada depois do retorno da rotina de tratamento da interrupção.

Podem existir até 32 *bottom half handlers*. Para suportar este sistema, existem dois registros de 32 *flags*, que são usados para sinalizar se um *bottom half handler* está instalado e se deve ser executado quando possível (existe trabalho a realizar). Existe também um vetor de 32 ponteiros para *bottom half handlers* existentes no *kernel*. Cada bit nas *flags* corresponde ao ponteiro de mesma posição numérica no vetor. Estas posições são estáticas, não há a possibilidade de se incluir um *bottom half handler* em tempo de execução. Existem *bottom half handlers* predefinidos para o tratamento da interrupção do temporizador, tratamento do console, gerenciamento de tarefas do subsistema de rede, e um gerenciador genérico, que pode ser utilizado por qualquer *driver* que deseja deixar algum processamento para ser realizado mais tarde.

As *flags* são observadas ao final de cada chamada de sistema, antes de retornar ao processo chamador, e se algum bit indica a necessidade de executar um *bottom half handler* ele é executado imediatamente.

Outro mecanismo importante é o suporte a filas de tarefas (task queues). As filas de tarefas são a forma que o Linux suporta o adiamento de processamento genérico (em contraste com o adiamento específico dos *bottom half handlers*). Estas filas são comumente utilizadas em conjunto com os *bottom half handlers*; por exemplo, a fila de tarefas do temporizador é executada quando o *bottom half handler* do temporizador é acionado. Uma fila de tarefas é uma estrutura de dados simples, que consiste de uma lista encadeada simples de estruturas `tq_struct` cada uma contendo o endereço de uma rotina e um ponteiro para algum conjunto de dados. A

rotina será chamada quando o elemento da fila de tarefas for executado, e terá como parâmetro o ponteiro para o conjunto de dados.

Quando as filas de tarefas são processadas, o ponteiro para o primeiro elemento da fila é removido da fila e substituído por um ponteiro nulo. Então cada elemento da fila tem sua função chamada um de cada vez. Os elementos de dados na fila são normalmente dados alocados estaticamente, porém não existe nenhum mecanismo inerente para desalocar estes dados. A rotina de processamento das filas de tarefas simplesmente passa para o próximo elemento, portanto é necessário que cada rotina colocada em uma fila de tarefas saiba o que fazer com seus dados e se eles precisam ser desalocados ou não.

Um sistema operacional deve poder agendar algum trabalho para ser feito em algum momento no futuro. Qualquer processador que queira suportar um sistema operacional deve possuir um temporizador programável que periodicamente interrompe o processador. Esta interrupção periódica é chamada *clock tick* e age como um metrônomo, orquestrando as atividades do sistema operacional. O Linux tem uma visão muito simples da medição do tempo: ele mede o número de *clock ticks* desde a carga do sistema. Todos os tempos do sistema são baseados nesta medição, que é conhecida como `jiffies`, por causa da variável global homônima.

O Linux possui dois tipos de temporizadores, e ambos agendam rotinas para serem executadas em algum instante, porém sua implementação difere um pouco. O sistema antigo possui um vetor estático de 32 ponteiros para estruturas de dados `timer_struct` e um mapa de bits para indicar se cada uma delas está ativa. Este mecanismo está caindo em desuso, por ser pouco flexível e por não suportar o envio de dados às rotinas. O novo sistema utiliza uma lista encadeada de estruturas `timer_list`, que é mantida em ordem ascendente de prazo.

Ambos utilizam o tempo em `jiffies` como prazo, portanto um temporizador que queira agendar algo para 5s deve converter estes 5s em `jiffies` a adicionar este valor ao tempo atual do sistema. O *bottom half handler* do temporizador gerencia ambos os tipos de temporizadores. Quando o prazo de algum elemento já expirou ele é removido da lista e sua função é chamada. A função que coloca novos elementos nesta lista é a `add_timer`, que recebe como parâmetro apenas uma estrutura `timer_list` corretamente inicializada [RUS 97]. Este é o mecanismo utilizado na função de atraso de pacotes na ferramenta ComFIRM.

O último mecanismo importante para este trabalho é o de filas de espera. Existem muitas situações em que um processo deve esperar pela disponibilidade de algum recurso. Por exemplo, um processo pode fazer uma leitura de um bloco de um arquivo e este não estar disponível no cache de disco do Linux. Neste caso, ele deve esperar até que a operação de disco (demorada) esteja pronta para poder seguir sua execução. O Linux utiliza uma estrutura de dados muito simples, a `wait_queue`, que consiste de um ponteiro para uma `task_struct` e um ponteiro para a próxima `wait_queue` da fila. Um processo que deva esperar algum recurso é retirado da fila de processos prontos para executar, e colocado na fila de espera deste recurso. Quando o recurso estiver pronto, seu gerenciador se encarrega de recolocar todos os processos em espera na fila de processos prontos para execução [POM 99]. Este mecanismo é utilizado na ferramenta ComFIRM para colocar processos que estão lendo o registro (log) da ferramenta em espera caso não exista dado para ser lido.

4.6 Conclusão

Este capítulo demonstrou que o Sistema Operacional Linux é uma base bastante sólida para a implementação de ferramentas de software básico, por prover um sistema operacional robusto, ágil, completo e com todo o código fonte disponível para estudo e alteração. Foram apresentadas uma introdução ao Linux e suas características mais básicas, bem como foram aprofundados pontos que interessam a este trabalho.

O suporte a um sistema de arquivos virtual como o `proc` torna bastante fácil o gerenciamento de ferramentas internas ao *kernel*, por eliminar a necessidade de se criar novas chamadas de sistema, acessíveis apenas por programas específicos. A organização em camadas do subsistema de redes também facilita sua compreensão e alteração, porém alguns pontos ainda estão um tanto obscuros para alguém que não participou de sua concepção. Ainda assim, percebe-se que a comunidade de usuários do Linux está trabalhando para remediar este fato, criando vários projetos de documentação.

É importante ressaltar a filosofia por trás do Linux, um sistema operacional desenvolvido desde seu princípio com um espírito de troca e de cooperação. Não fosse a disponibilidade e possibilidade de alteração do código, este projeto não teria cativado centenas de desenvolvedores provenientes dos mais variados pontos do mundo. Cada pessoa que sente necessidade de alguma característica a implementa e contribui o código para o resto da comunidade, de forma a melhorar o sistema de todos.

Este mesmo espírito deve perpassar qualquer projeto que envolva Linux, e se possível deve-se sempre contribuir o código e os resultados de volta. Ainda que não se inclua alguma característica no *kernel* padrão, o código desenvolvido pode se tornar acessível e visível a milhares de usuários ao redor do mundo, aumentando as chances de se encontrar pessoas com iguais interesses e dispostas a testar, colaborar e melhorar aquilo que se produziu.

5 A ferramenta ComFIRM

Tendo já revisto o conhecimento teórico necessário à compreensão da ferramenta, pode-se apresentar sua proposta e implementação. Neste capítulo serão examinados o código adicionado ao núcleo e as formas com que este código interage com o resto do sistema operacional. É apresentada também uma interface gráfica simples, cujo principal propósito é demonstrar que a ferramenta pode ser utilizada de forma amigável. Apenas as partes do código que são interessantes serão incluídas no texto do capítulo. Os códigos completos, tanto da ferramenta quanto da interface, estão disponíveis respectivamente nos apêndices A e B.

5.1 Considerações iniciais

Pretende-se com este trabalho implementar uma ferramenta que reflita ao máximo as considerações tecidas em capítulos anteriores sobre injeção de falhas. Sendo assim, espera-se produzir uma ferramenta que seja bastante flexível na especificação dos experimentos, que permita sua reconfiguração em tempo de execução e que utilize uma linguagem simples para especificação de regras de injeção de falhas. Ao término deste capítulo espera-se ter demonstrado como a ferramenta ComFIRM atinge todos estes objetivos.

A ComFIRM se situa dentro do núcleo do Sistema Operacional Linux, no ponto mais baixo do tratamento de mensagens pelo subsistema de rede. Utilizando-se a figura 4.6 como exemplo, a ComFIRM está localizada na camada independente de dispositivo, interceptando as rotinas `dev_queue_xmit` e `netif_rx`. No código implementado, nota-se que estas rotinas foram renomeadas, e novas rotinas, que funcionam como filtros, foram colocadas em seu lugar. As novas rotinas eventualmente chamam as antigas para dar continuidade ao processamento das mensagens, após a injeção de falhas.

A implementação da ferramenta se deu de forma a interferir o mínimo possível com os arquivos existentes, para que a compreensão de seu funcionamento fosse facilitada. Os arquivos de código fonte existentes que foram alterados sofreram modificações apenas para efeito de ligação do código existente com o código da ComFIRM. Sendo assim, nota-se por exemplo que no arquivo `net/core/Makefile` (seção A.1) foi modificada apenas uma linha, incluindo o arquivo objeto `comfirm.o`, que contém o código da ferramenta, no processo de ligação do código do subsistema de rede.

No arquivo `include/linux/proc_fs.h`, foram incluídas algumas constantes (seção A.2) que representam os *i-nodes* dos arquivos virtuais utilizados pela ferramenta. Já no arquivo `kernel/sched.c`, vê-se uma pequena adição de código ao *bottom-half handler* do temporizador do sistema (seção A.3). A cada interrupção do temporizador o sinalizador deste *bottom-half handler* é ativado, e assim que a interrupção retorna esta rotina é chamada. Portanto, esta rotina foi escolhida para atualizar os temporizadores que a ComFIRM disponibiliza.

Na seção A.4 nota-se a última alteração de código existente, desta vez interceptando as rotinas de transmissão e recepção de pacotes. Vê-se claramente que a rotina `dev_queue_xmit` é renomeada para `old_dev_queue_xmit`, assim como `netif_rx` é renomeada para `old_netif_rx` e novas rotinas são colocadas em seu lu-

```
root@thor:~# cat /proc/net/ComFIRM_Log
00135d1a: ComFIRM version 0.5
00135ddc: tx rules have been erased
00136184: fault injection started
00136338: fault injection stopped
001364fe: ComFIRM has been reset
_
```

FIGURA 5.1 – Teste do registro da ferramenta ComFIRM

gar. Estas novas rotinas apenas verificam se a ComFIRM está ativada e se existem regras de transmissão ou recepção, chamando as rotinas originais diretamente caso alguma condição não seja satisfeita.

Caso as condições sejam satisfeitas, então as novas rotinas redirecionam a chamada para as rotinas de injeção de falhas, ativando a ferramenta. Como última alteração interessante neste arquivo, nota-se a chamada à função `ComFIRM_Init`, responsável pela inicialização das estruturas da ferramenta. Todo o resto do código da ferramenta está isolado em dois arquivos separados, `comfirm.h` e `comfirm.c`. O código presente nestes arquivos será visto mais adiante.

5.2 Recursos da ferramenta ComFIRM

A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) foi projetada para permitir a experimentação com vários elementos de seleção e manipulação de pacotes de rede, portanto ela possui alguns recursos que são usados na definição de regras para injeção de falhas, bem como recursos para seu controle e uso.

Os recursos mais aparentes são os quatro arquivos virtuais que ela disponibiliza, e que são usados para seu controle. Os arquivos ficam no diretório `/proc/net`. O arquivo `ComFIRM_Log` é somente de leitura, e disponibiliza informações sobre as operações da ferramenta e o resultado das injeções. Um experimento de injeção de falhas não é válido se não há um registro daquilo que foi feito, e este arquivo provê exatamente este registro. Quando um processo abre este arquivo, a ferramenta começa a registrar suas ações. Ao tentar ler deste arquivo, se não houver informação registrada este processo será bloqueado, portanto é possível, para colher dados de um teste rápido, simplesmente rodar `cat /proc/net/ComFIRM_Log` em um terminal (figura 5.1).

Já o arquivo `ComFIRM_Control`, com permissões somente para escrita, é a entrada de comandos para a ferramenta. Basta escrever comandos em texto simples neste arquivo para que a ferramenta os execute. Por exemplo, a figura 5.2 apresenta os comandos usados para obter as mensagens da figura 5.1. Vê-se que a ferramenta pode muito bem ser controlada através de linha de comando, ou mesmo por scripts automatizados.

Os arquivos `ComFIRM_TX_Rules` e `ComFIRM_RX_Rules` são usados para conter as regras de injeção de falhas, e seu uso será abordado mais adiante. Estes são os

```
root@thor:~# echo log version > /proc/net/ComFIRM_Control
root@thor:~# echo erase tx > /proc/net/ComFIRM_Control
root@thor:~# echo ComFIRM on > /proc/net/ComFIRM_Control
root@thor:~# echo ComFIRM off > /proc/net/ComFIRM_Control
root@thor:~# echo ComFIRM reset > /proc/net/ComFIRM_Control
root@thor:~# _
```

FIGURA 5.2 – Enviando comandos à ferramenta ComFIRM

TABELA 5.1 – Instruções de seleção da ferramenta ComFIRM

ComFIRM_Test_Bit	0x00
ComFIRM_Test_Byte	0x08
ComFIRM_Test_Word	0x10
ComFIRM_Test_Double	0x18
ComFIRM_Test_Counter	0x20
ComFIRM_Test_RandByte	0x28
ComFIRM_Test_Timer	0x30
ComFIRM_Test_Flag	0x38

recursos de interação da ferramenta com o seu usuário, ou com sua interface.

Para realizar a injeção de falhas, a ferramenta possui internamente instruções primitivas, como uma linguagem de máquina, e recursos extras que são: contadores, temporizadores e sinalizadores (*flags*). As instruções são primitivas de injeção de falhas ou de manipulação e teste destes recursos, e a utilização destas instruções em conjunto permite a construção de modelos de falhas bastante complexos. A tabela 5.1 mostra as instruções de seleção de pacotes disponíveis e seus códigos hexadecimais, e a tabela 5.2 os modificadores que podem ser usados em operações de teste.

Vê-se que é possível testar bits, bytes, words e doublewords da mensagem, bem como contadores, temporizadores, sinalizadores e bytes aleatórios. Com exceção do teste de bits da mensagem e dos sinalizadores, as operações de teste podem ser combinadas com os modificadores da tabela 5.2 para igualdade com algum valor e as relações maior que e menor que. Os testes de bit podem ser combinados com os modificadores para teste se o bit é zero ou um.

Estas são as possibilidades de seleção da ferramenta, e pela tabela 5.3 pode-se

TABELA 5.2 – Modificadores para instruções de seleção

ComFIRM_Test_Equal	0x00
ComFIRM_Test_Greater	0x01
ComFIRM_Test_Less	0x02
ComFIRM_Test_Bit0	0x00
ComFIRM_Test_Bit1	0x01

TABELA 5.3 – Instruções de manipulação da ferramenta ComFIRM

ComFIRM_Action_Bit	0x40
ComFIRM_Action_Byte	0x48
ComFIRM_Action_Word	0x50
ComFIRM_Action_Double	0x58
ComFIRM_Action_Drop	0x60
ComFIRM_Action_Delay	0x68
ComFIRM_Action_Dup	0x70
ComFIRM_Action_Counter	0x78
ComFIRM_Action_Timer	0x80
ComFIRM_Action_Flag	0x88
ComFIRM_Action_Dump	0x90

TABELA 5.4 – Modificadores para instruções de manipulação

ComFIRM_Action_Set	0x00
ComFIRM_Action_Inc	0x01
ComFIRM_Action_Dec	0x02
ComFIRM_Action_Bit0	0x00
ComFIRM_Action_Bit1	0x01
ComFIRM_Action_BitC	0x02
ComFIRM_Action_Timer_Mode_None	0x04
ComFIRM_Action_Timer_Mode_Inc	0x05
ComFIRM_Action_Timer_Mode_Dec	0x06

notar que as instruções de ação (manipulação de mensagens ou recursos da ferramenta) são similares. Algumas instruções de ação aceitam, também, sua combinação com modificadores (tabela 5.4), de modo a melhor especificar o que deve ser feito.

A ComFIRM suporta instruções de manipulação de bits, bytes, words e doublewords da mensagem, bem como descarte, atraso e duplicação, e ainda operações sobre os temporizadores, contadores e sinalizadores. Existe ainda uma instrução que causa a descarga do conteúdo da mensagem em hexadecimal no registro da ferramenta, o que é útil para depuração de protocolos. Os modificadores para ação são também bastante diferentes, sendo que se pode atribuir um valor, incrementar ou decrementar um valor, ou ainda forçar um bit (ou sinalizador) para zero, um ou seu complemento. Para os temporizadores é possível ainda reconfigurar seu modo de operação.

Todas estas informações estão no arquivo `include/linux/comfirm.h` (seção A.5). A próxima seção detalha a utilização destas instruções e dos recursos internos da ferramenta para a geração de experimentos de injeção de falhas de comunicação.

5.3 Instruções e regras

Como já foi dito, a ComFIRM injeta falhas segundo regras especificadas pelo usuário. Estas regras são formadas pela concatenação de instruções que serão interpretadas pela ComFIRM. Como a ferramenta está dentro do núcleo do sistema operacional, não se pode perder muito tempo na avaliação destas regras, portanto as instruções são codificadas em bytes, à exemplo de uma linguagem de máquina.

A avaliação das regras é seqüencial, sendo terminada quando alguma instrução de seleção retorna um valor falso ou ainda quando são utilizadas algumas instruções de manipulação de mensagens. Esta organização exige alguma reflexão antes de se executar um experimento, mas logo se parece bastante lógica.

Por exemplo, para se descartar o terceiro pacote transmitido por um computador, coloca-se na regra a instrução de incremento de um contador, seguida do teste da igualdade do valor deste contador com o número três, e então a instrução de descarte de mensagens. Estas três instruções são codificadas e então colocadas no arquivo virtual que contém as regras de injeção de falhas na transmissão.

Quando da transmissão do primeiro pacote após a ativação da ferramenta, a regra será avaliada da seguinte forma: a primeira instrução causa o incremento do contador, que assume o valor um; a seguir, a instrução de comparação deste contador com o valor três falha, e a avaliação da regra é terminada. O mesmo ocorre com o segundo pacote, apenas com o contador assumindo o valor dois. Assim que o terceiro pacote é transmitido, o contador assume o valor três (pela primeira instrução) e então desta vez o teste da segunda instrução retorna um valor verdadeiro; a avaliação continua com a execução da instrução de descarte da mensagem. Como não há mais instruções ou regras, o pacote é descartado e o processamento segue normalmente. A mensagem passou por todos os protocolos, porém foi descartada logo antes de atingir o dispositivo de comunicação, portanto a falha injetada é muito semelhante à uma perda da mensagem na própria rede. A seguir serão comentados o formato e a codificação de cada uma das instruções, bem como a formatação das regras, simples ou múltiplas.

5.3.1 Codificação das Instruções

Cada instrução consiste de um byte especificando o código e o modo da operação seguido de tantos bytes quantos forem necessários para especificar seus possíveis operandos. Dependendo de seu significado, os operandos podem ser codificados com o byte menos significativo primeiro (LSB), ou ao contrário, visto que o padrão de ordenação de bytes na rede é MSB. Nem todas as instruções possuem modos diferentes (por exemplo, descarte apenas descarta). A lista a seguir indica os códigos de cada instrução, com suas variações e operandos.

ComFIRM_Test_Bit `(0x00 AA AA)`**:** esta instrução permite que se teste se o valor de um bit da mensagem é zero ou um. O código de operação pode ser `0x00` para testar se o bit é zero, ou `0x01` para testar se é um. `AA AA` é um inteiro de 16 bits que indica qual bit da mensagem deve ser testado. Por exemplo, para testar se o bit número 100 da mensagem tem valor um, codifica-se `0x01 0x64 0x00`.

ComFIRM_Test_Byte `(0x08 AA AA BB)`**:** permite que se teste o valor de um byte da mensagem. O código de operação pode ser `0x08`, `0x09` ou `0x0a`,

para testar se o byte indicado pelo inteiro de 16 bits `AA AA` é respectivamente igual, maior ou menor que o operando `BB`. Exemplo: para testar se o byte de número 500 na mensagem tem valor 25, codifica-se `0x08 0xf4 0x01 0x19`.

ComFIRM_Test_Word `(0x10 AA AA BB BB)`**:** esta instrução testa o valor de uma palavra de 16 bits na mensagem (como uma porta UDP). Assim como a anterior, pode aparecer como `0x10`, `0x11` ou `0x12`, para testar se a palavra apontada por `AA AA` é igual, maior ou menor que a palavra fornecida em `BB BB`. É importante notar que a codificação do parâmetro `BB BB` se dá em *network byte order*, ou seja, MSB. Para se testar se a porta destino UDP de um pacote é a do serviço de DNS (número 53) utiliza-se `0x10 0x24 0x00 0x00 0x35`. `0x24 0x00` dá a posição desta informação dentro da mensagem, e `0x00 0x35` é o valor que se está testando (53 em hexadecimal MSB).

ComFIRM_Test_Double `(0x18 AA AA BB BB BB BB)`**:** esta função é essencialmente igual à anterior, porém operando em inteiros de 32 bits. Pode ser usada, por exemplo, para testar um endereço IP. Os códigos de operação são `0x18`, `0x19` e `0x1a` para testar se a palavra de 32 bits apontada por `AA AA` é igual, maior ou menor que o valor fornecido em `BB BB BB BB` (este parâmetro novamente está em MSB, por se tratar de um valor relativo à mensagem, que segue o *network byte order*).

ComFIRM_Test_Counter `(0x20 AA BB BB BB BB)`**:** esta operação testa o valor de um dos contadores fornecidos pela ferramenta. O parâmetro `AA` é um byte especificando qual contador deve ser testado, e `BB BB BB BB` é um valor de 32 bits, em LSB, para ser utilizado na comparação. Os códigos de operação são `0x20`, `0x21` e `0x22`, para testar se o contador é igual, maior ou menor que o valor fornecido.

ComFIRM_Test_RandByte `(0x28 AA)`**:** sorteia um byte aleatório, e testa se este byte é igual, maior ou menor que o valor fornecido em `AA`. Os códigos de operação para estes testes são respectivamente `0x28`, `0x29` e `0x2a`. Por exemplo, para um teste que seja verdadeiro em mais ou menos 10% dos casos, basta testar se o valor sorteado é menor que 25, com a seguinte seqüência: `0x2a 0x19` (como é sorteado um byte, este pode variar entre 0 e 255, sendo 25 uma aproximação de um décimo do valor total). A qualidade do gerador de números aleatórios disponível no núcleo do Linux garante uma distribuição adequadamente equilibrada dos valores sorteados.

ComFIRM_Test_Timer `(0x30 AA BB BB BB BB)`**:** funciona da mesma forma que o teste de contadores, porém permite testar o valor de um dos temporizadores fornecidos pela ferramenta. `AA` indica qual temporizador testar, e `BB BB BB BB` fornece o valor do teste. Os temporizadores são implementados como contadores que são incrementados ou decrementados a intervalos de 10ms. Sendo assim, para testar se já passou um segundo desde a ativação do temporizador 0 (se seu valor é maior que 100), codifica-se `0x31 0x00 0x64 0x00 0x00 0x00`. Os códigos de operação para teste de igual, maior ou menor são obtidos do código base da mesma forma que nas demais instruções.

ComFIRM_Test_Flag `(0x38 AA)`**:** última instrução de teste, se assemelha ao teste de bits da mensagem, porém o teste é feito sobre uma das 32 *flags* (sinal-

izadores) fornecidos. O parâmetro `AA` indica qual sinalizador testar, e os códigos de operação podem ser `0x38` ou `0x39` para testar se o sinalizador é zero ou um.

ComFIRM_Action_Bit `(0x40 AA AA)`: assim como o teste de bits, esta instrução recebe uma palavra de 16 bits indicando sobre qual bit da mensagem agir. O modo de operação indica se o bit deve ser zerado, ligado ou complementado (códigos de operação `0x40`, `0x41` e `0x42`).

ComFIRM_Action_Byte `(0x48 AA AA [BB])`: permite que se incremente, decremente ou atribua um valor ao byte de número `AA AA` da mensagem. O código de operação `0x48` necessita do parâmetro `BB`, que fornece o valor a ser atribuído ao byte. Já os códigos para incremento e decremento (`0x49` e `0x4a`) recebem apenas a palavra `AA AA`.

ComFIRM_Action_Word `(0x50 AA AA [BB BB])`: ação essencialmente igual à instrução anterior, porém permitindo agir sobre uma palavra de 16 bits da mensagem. Em caso de atribuição, é necessário observar que o parâmetro `BB BB` é codificado em MSB.

ComFIRM_Action_Double `(0x58 AA AA [BB BB BB BB])`: esta ação funciona também como a anterior, apenas trabalhando com inteiros de 32 bits dentro da mensagem.

ComFIRM_Action_Drop `(0x60)`: esta ação descarta a mensagem sendo analisada, e causa o término da avaliação das regras. Não há parâmetros.

ComFIRM_Action_Delay `(0x68 AA AA AA AA)`: atrasa o envio ou o recebimento da mensagem sendo analisada. O único parâmetro é um inteiro de 32 bits indicando o atraso em centésimos de segundo. Causa o término da avaliação das regras.

ComFIRM_Action_Dup `(0x70)`: esta instrução duplica a mensagem sendo avaliada, e causa o término da avaliação das regras. Não recebe parâmetros.

ComFIRM_Action_Counter `(0x78 AA [BB BB BB BB])`: ação sobre os contadores fornecidos pela ComFIRM. O parâmetro `AA` indica qual contador deve ser utilizado. O código para atribuição de um valor é `0x78`, e neste caso o parâmetro `BB BB BB BB` contém o novo valor do contador. Os códigos de operação para incremento e decremento de contadores são `0x79` e `0x7a`.

ComFIRM_Action_Timer `(0x80 AA [BB BB BB BB])`: esta instrução é mais complexa que as demais, por permitir configurar o modo de operação dos temporizadores. O parâmetro `AA` indica sobre qual temporizador agir. O código de operação `0x80` realiza a atribuição de um valor ao temporizador, caso em que o parâmetro `BB BB BB BB` contém seu novo valor. O código também pode ser `0x84` para desligar o temporizador, `0x85` para configurar o modo de incremento, e `0x86` para o modo de decremento. O estado normal dos temporizadores é desligado. A partir do momento que seu modo é reconfigurado, o temporizador entra em funcionamento, tendo seu valor aumentado ou diminuído a cada centésimo de segundo.

ComFIRM_Action_Flag `(0x88 AA)`**:** muito parecida com a ação sobre bits da mensagem, esta instrução permite a ação sobre os sinalizadores fornecidos pela ferramenta. O parâmetro `AA` indica sobre qual sinalizador agir, e o código de operação indica zerar, ligar ou complementar (respectivamente `0x88`, `0x89` e `0x8a`).

ComFIRM_Action_Dump `(0x90)`**:** interessante para depuração e investigação de protocolos, esta instrução causa uma descarga do conteúdo da mensagem em hexadecimal no registro da ComFIRM. A avaliação da regra segue normalmente. Esta instrução não recebe parâmetros.

Como se pode perceber, a codificação das instruções é bastante resumida, o que dificulta sua utilização direta. É difícil resumir ainda mais esta pequena linguagem, já que só é especificado aquilo que é necessário. Poder-se-ia alegar que contadores de 32 bits não são necessários, pois dificilmente se contará com esta ferramenta alguns bilhões de pacotes. Esta decisão foi tomada levando em consideração que os processadores em geral trabalham de forma mais eficiente com palavras do tamanho de seus registradores internos. Contadores de 16 bits, embora possam ser suficientes, poderiam ter um impacto negativo no desempenho do sistema.

Agora que as instruções primitivas de injeção de falhas de comunicação já foram vistas, inclusive com sua forma de codificação, a próxima subseção indica como compor estas instruções para formar regras, e como definir os conjuntos de regras a serem usados pela ferramenta.

5.3.2 Formatação das regras

Como já foi dito, as regras de injeção de falhas de comunicação da ferramenta ComFIRM são compostas pela simples concatenação dos bytes que representam cada uma das instruções desejadas. Mas em sua formatação ainda existe um pequeno passo adicional, que visa possibilitar a utilização de várias regras independentes para transmissão e/ou recepção.

Sendo assim, as regras na ComFIRM consistem de, primeiro, um byte que indica o tamanho da regra (também em bytes), seguido dos bytes que compõem as instruções desta regra. Isto permite que se crie uma seqüência de regras apenas pela concatenação de regras completas, terminadas por uma regra nula, representada por apenas um byte de tamanho zero.

Supondo-se que um dado experimento consista de duas regras, uma que apenas complementa o valor de um sinalizador e outra que descarta a mensagem sendo avaliada quando este sinalizador tem valor zero, estas regras seriam codificadas da seguinte maneira:

Primeira regra: esta consiste apenas da instrução de ação sobre sinalizadores, combinada com o modificador de complemento, aplicada ao sinalizador zero. Seu código fica portanto `0x8a 0x00`. Como ela possui dois bytes, sua formatação final fica `0x02 0x8a 0x00`.

Segunda regra: já esta possui duas instruções, a de teste do sinalizador e a de descarte. Utiliza-se a instrução de teste de sinalizador, combinada com o modificador de teste de bit igual a zero, aplicada ao sinalizador zero, seguida da instrução de ação de descarte, que não possui modificador ou parâmetro. Seu

código fica portanto `0x38 0x00 0x60`. Possuindo três bytes, sua formatação final fica `0x03 0x38 0x00 0x60`.

Já se tem as duas regras prontas, portanto basta agora concatená-las e adicionar uma regra nula para terminar a seqüência. Isso nos leva à seqüência final de bytes `0x02 0x8a 0x00 0x03 0x38 0x00 0x60 0x00`. Escrevendo-se tal conteúdo no arquivo `/proc/net/ComFIRM_TX_Rules`, por exemplo, e ativando-se a ferramenta, tem-se uma máquina que vai descartar pacotes transmitidos alternadamente. Cada pacote transmitido dispara a avaliação das regras. A primeira regra apenas inverte o estado de um sinalizador. A segunda regra então verifica se este sinalizador é zero e, caso seja, descarta o pacote. Quando este sinalizador apresentar o valor um, o teste vai falhar e a avaliação da segunda regra será terminada. Como não há regra após a segunda, o pacote simplesmente seguirá seu caminho pelos níveis mais baixos do subsistema de redes e será eventualmente transmitido.

Este formato simples e bastante compacto é um pouco complexo para ser usado manualmente em experimentos maiores, envolvendo mais regras e regras maiores, porém é ideal para ser avaliado pelo núcleo do sistema operacional. O byte contendo o tamanho da regra, no início desta, permite um rápido salto para o início da próxima regra, caso alguma instrução naquela venha a falhar. Espera-se que o limite implícito de 255 bytes por regra não venha a ser um empecilho na especificação de experimentos mais abrangentes. Salienta-se, no entanto, que embora cada regra tenha seu tamanho limitado, o número de regras concatenadas em uma seqüência está limitado apenas pela quantidade de memória presente na máquina, e também que é trivial aumentar, em uma nova versão da ComFIRM, este limite de 255 bytes.

5.4 Utilização da ferramenta

Agora que já se conhece as instruções implementadas, como combiná-las em regras, e a função básica dos arquivos virtuais disponibilizados pela ComFIRM, esta seção traz os últimos detalhes necessários ao uso da ferramenta. Suponha-se que, como exemplo, durante esta seção se deseje atrasar em 250ms um a cada cinco pacotes recebidos pela máquina que contém a ferramenta.

Começa-se pelo registro do experimento. Como já foi visto, para obter o registro das ações da ferramenta, basta que se leia o arquivo `/proc/net/ComFIRM_Log`. Esta é a única função deste arquivo. Na figura 5.1 nota-se um exemplo simples de obtenção do registro, e se percebe que cada linha é precedida de um número em hexadecimal. Este número é uma marca de tempo (*timestamp*) relativa, com precisão de 10ms, obtido através do contador de interrupções do temporizador da máquina. É interessante para se observar o intervalo de tempo entre ações da ferramenta.

As informações lidas do registro da ferramenta por um processo são geradas em vários pontos do código da ferramenta, porém estas informações só são armazenadas se houver algum processo lendo do arquivo virtual de registro. Ou seja, o processo que abre e lê o arquivo de registro recebe somente as informações que vierem a ser geradas após a abertura do arquivo. Não há armazenamento das mensagens quando não há um processo lendo o registro. Isso não implica que o processo que está controlando a ferramenta deverá estar lendo constantemente; desde que exista um processo com o arquivo de registro aberto, as informações são geradas e armazenadas, para serem enviadas a este processo na próxima operação de leitura.

```
root@thor:~# perl -e print '"\x13\x79\x00\x20\x00\x05\x00\x00\x00"
                            "\x78\x00\x00\x00\x00\x00\x68\x19\x00"
                            "\x00\x00\x00";' > ComFIRM_RX_Rules
root@thor:~# _
```

FIGURA 5.3 – Configurando regras de recepção

Depois de ativar o registro do experimento, é necessário especificar as regras de injeção de falhas de comunicação que serão usadas na transmissão e/ou recepção. A codificação e formatação de regras já foi alvo de seções anteriores, portanto aqui se tratará a forma de enviar estas informações para a ferramenta.

Os arquivos que se destinam a conter as seqüências de regras para transmissão e recepção são, respectivamente, `ComFIRM_TX_Rules` e `ComFIRM_RX_Rules`. Estes arquivos são virtuais, ou seja, as operações de leitura e gravação de dados nestes arquivos na verdade acontecem em estruturas de memória. Ao se escrever alguma coisa nestes arquivos, a ferramenta ComFIRM descarta as seqüências anteriores (se existirem), aloca memória suficiente para guardar as novas seqüências, e então as armazena nesta área. Leituras executadas sobre estes arquivos retornam o conteúdo atual da área de armazenamento de seqüências de regras.

As regras devem ser escritas nestes arquivos já na forma como serão utilizadas pela ferramenta, e a ferramenta vai confiar naquilo que for escrito (razão pela qual somente o superusuário pode escrever nestes arquivos). Se forem escritas seqüências incorretas ou inconsistentes de bytes, o funcionamento da ferramenta será incerto. Uma maneira prática de escrever as seqüências de bytes que implementam as regras desejadas é com algum programa que permita que se especifique os bytes em hexadecimal.

Para o experimento proposto no início desta seção, é necessário incrementar um contador, testar se seu valor é igual a cinco, e então zerar o contador e atrasar o pacote. Isto resulta na seguinte seqüência de bytes (já com o byte inicial de tamanho e a regra nula no final): `0x13 0x79 0x00 0x20 0x00 0x05 0x00 0x00 0x00 0x78 0x00 0x00 0x00 0x00 0x00 0x68 0x19 0x00 0x00 0x00 0x00`. Para se avaliar este conjunto de regras na recepção de pacotes, basta escrever a seqüência no arquivo `ComFIRM_RX_Rules`. A figura 5.3 demonstra como isso pode ser feito usando-se a linguagem Perl, aproveitando-se a possibilidade de se representar os bytes de uma seqüência em hexadecimal nesta linguagem.

Com o registro ativado e as regras de recepção no lugar, basta que ativemos a ferramenta. Aqui entra em uso o arquivo `ComFIRM_Control`, que não recebeu muita atenção até agora. Este é um arquivo somente para escrita, que recebe os comandos que se deseja passar à ferramenta. Linhas de texto escritas neste arquivo são passadas ao código da ComFIRM, que analisa a linha procurando por comandos a executar. Somente um comando pode ser enviado por vez. Os seguintes comandos são suportados atualmente:

ComFIRM on: ativa a ferramenta, iniciando o experimento de injeção de falhas. Se houverem regras de transmissão ou recepção, estas passarão a ser avaliadas para cada mensagem;

ComFIRM off: desativa a ferramenta. A avaliação das regras não ocorre mais, o subsistema de rede funciona normalmente como se a ferramenta não existisse;

ComFIRM reset: reinicia as estruturas da ferramenta, incluindo os contadores, temporizadores e sinalizadores. As regras são apagadas, e a ferramenta é desativada;

log action on: habilita o registro das ações da ferramenta. Isto é importante para se obter um registro detalhado das falhas injetadas durante o experimento;

log action off: desabilita o registro das ações da ferramenta;

log debug on: habilita o registro de informações extras de depuração. É interessante apenas para se verificar o funcionamento correto da ferramenta, ou obter excesso de detalhes sobre o experimento;

log debug off: desabilita o registro de informações de depuração;

log version: registra a versão da ferramenta em uso. Pode ser usado por um programa que manipule a ComFIRM diretamente para determinar se é compatível com a versão da ComFIRM no núcleo do sistema;

erase tx: apaga as regras de transmissão;

erase rx: apaga as regras de recepção.

Portanto, agora que já está tudo pronto para iniciar o experimento, basta executar "`echo ComFIRM on > /proc/net/ComFIRM_Control`" para que as regras entrem em ação. O próximo capítulo mostrará testes como este em mais detalhes, bem como experimentos mais interessantes.

5.5 A interface ComFIRM's Face

Pela descrição da seção anterior, pode-se notar que a ativação da ferramenta e a colheita de resultados através do registro são operações simples. O principal problema com a ativação manual da ferramenta é a codificação e correta formatação das instruções de injeção de falhas. Criar modelos de falha complexos pode requerer o uso de várias instruções e regras, compartilhando a manipulação dos recursos para obter o resultado desejado. A manipulação manual das seqüências de bytes geradas pode muito bem sair do controle do usuário, ou pelo menos tornar-se trabalhosa.

Para demonstrar que passar por tal situação não é de forma alguma a única forma de se utilizar a ferramenta, foi criado um protótipo de uma interface gráfica para a ComFIRM, chamada de ComFIRM's Face. Esta interface é programada na linguagem Perl [WAL 96], usando o módulo Perl/Tk [SRI 97] para construir suas janelas e objetos gráficos. A interface é bastante simples, e explora apenas algumas das possibilidades da ComFIRM. Ainda assim, modelos de falhas simples, determinísticos ou não, podem ser criados tanto para transmissão quanto para recepção.

A figura 5.4 mostra como esta interface se apresenta ao usuário. Durante todo o tempo em que estiver ativa, a interface estará lendo o arquivo de registro, portanto mesmo que a ferramenta esteja sendo controlada manualmente a interface registrará

FIGURA 5.4 – Interface gráfica experimental da ferramenta ComFIRM

o que for feito. A área de registro da interface é a caixa de texto na região inferior da janela. Esta caixa de texto, que apresenta barras de rolagem quando necessário, aceita as funções típicas de seleção e cópia, para posterior colagem em outro aplicativo ou editor de texto. Com este recurso, pode-se salvar pedaços importantes do registro da ferramenta.

A operação da ComFIRM's Face é bastante simples. A barra de menu no topo da janela apresenta as entradas "ComFIRM", "Log", "Apply changes" e "Help". O menu "ComFIRM" permite que se envie os comandos de ativação, desativação e reinicialização da ferramenta, bem como os comandos que apagam as regras de transmissão e recepção. Este menu permite ainda sair da interface gráfica. A entrada "Log" permite que se limpe a área de registro da ferramenta, que se ative ou desative o registro de mensagens de ação e de depuração, e também que se registre a versão da ferramenta.

O botão "Apply changes", que é ativo somente quando se faz alguma alteração nas configurações do experimento, realiza a configuração das regras da ferramenta para criar o modelo de falhas desejado. Finalmente, o menu "Help" apresenta apenas a opção "About", que abre uma janela de diálogo mostrando uma breve descrição da interface e informações sobre o autor.

Além da barra de menus e da área de registro, nota-se na interface quatro seções com elementos que permitem selecionar opções de montagem de modelos de injeção de falhas. A organização é extremamente simples, e a interface, em seu estado atual, é capaz de gerar apenas experimentos com uma única regra. Claro, pode-se configurar uma regra para recepção e outra para transmissão, mas não mais que isso. Além disso, a interface não lê o que já está programado na ferramenta, ela apenas escreve o que estiver configurado em seus botões e seleções.

A primeira seção, chamada "Stream direction", permite que se selecione se o modelo de falhas a aplicar será para transmissão ou recepção. Basta selecionar clicando sobre a opção desejada. A segunda seção, intitulada "Content-based selection" permite exercitar alguma seleção baseada em conteúdo, neste caso o protocolo em uso. Pode-se selecionar entre qualquer protocolo, TCP, UDP e ICMP. As três últimas opções necessitam que seja especificado na entrada ao lado da seleção o número da porta destino TCP ou UDP que será filtrada, ou, no caso do ICMP, o tipo da mensagem de controle desejado.

Já a terceira seção, "Stream-based selection", permite que se selecione mensagens baseado no fluxo de pacotes. Pode-se selecionar todos os pacotes, cada n-ésimo pacote, apenas o n-ésimo pacote, ou ainda $n\%$ dos pacotes. Para as três últimas opções é necessário especificar na entrada ao lado o valor de n. Por fim, a quarta seção permite que se selecione uma ação para aplicar nas mensagens selecionadas. É possível selecionar entre nada, descarte, duplicação e atraso. Para o caso do atraso, é necessário especificar seu valor em milisegundos na entrada ao lado. Para qualquer ação selecionada, pode-se opcionalmente ativar a opção "Dump contents", que descarrega no registro da ferramenta o conteúdo em hexadecimal da mensagem.

A operação da interface é bastante simples. Uma sessão típica consiste em selecionar nas seções de configuração do experimento o modelo de falhas que se quer utilizar, aplicar as mudanças, configurar as opções de registro desejadas, através do menu "Log", e então ativar a ferramenta, através da opção "Start" do menu "ComFIRM". Não é necessário nada mais; no momento em que se aplicam as regras e se ativa a ferramenta, o experimento já está em andamento. Para se finalizar um experimento, basta selecionar a opção "Stop" (ou "Reset") do menu "ComFIRM". O registro pode ser copiado para outras aplicações, a fim de registro permanente ou geração de um relatório do experimento, por exemplo.

A interface não suporta a criação de regras baseadas nos temporizadores e nos sinalizadores, portanto para tal a ferramenta deverá ser programada manualmente. Os temporizadores são interessantes para a criação de experimentos não determinísticos, onde se pode trabalhar com períodos de tempo de maior ou menor taxa de falhas. Os sinalizadores, por sua vez, são interessantes para quebrar regras muito grandes em regras menores. Pode-se, por exemplo, utilizar uma série de regras no início do conjunto que realizem vários testes, porém cuja única ação é atuar sobre os sinalizadores. Após esta série, regras que realmente injetem falhas podem testar apenas os sinalizadores que representem sua ativação ou disparo.

Como já foi dito, este protótipo de interface gráfica utiliza apenas uma fração das possibilidades da ComFIRM. Espera-se, no entanto, que este protótipo sirva de base para a criação de ferramentas mais abrangentes, provendo exemplos e sugestões sobre a forma com que se pode gerar regras dinamicamente. O próximo capítulo mostra vários testes e experimentos, e sempre que possível uma captura da configuração da ComFIRM's Face acompanha os relatos.

5.6 Conclusão

Este capítulo apresentou a ferramenta ComFIRM e demonstrou que ela é uma ferramenta de injeção de falhas de comunicação extremamente flexível, capaz de produzir uma grande variedade de modelos de falhas e que pode ser completamente reconfigurada em tempo de execução, o que confere uma agilidade muito grande ao condutor dos experimentos.

A ComFIRM se situa no núcleo do Sistema Operacional Linux, no nível mais baixo do seu subsistema de rede. Seu impacto no sistema é mínimo, visto que é ativada apenas pela recepção ou transmissão de mensagens pela rede. Capaz de selecionar pacotes a serem manipulados baseada em uma ampla gama de condições, a ComFIRM é uma ferramenta que permite até mesmo que experimentos sejam

conduzidos sem interferir na comunicação normal de uma estação conectada em rede.

Apesar de não ser demasiadamente difícil operar a ferramenta através de seus arquivos virtuais, a codificação e configuração manual das regras de injeção de falhas pode se tornar enfadonha e sujeita a erros. Para aumentar a facilidade de uso e confiança de funcionamento, foi mostrado também um protótipo de interface gráfica para a ComFIRM, chamada de ComFIRM's Face. Este protótipo permite alguma experimentação com a ferramenta, de forma bastante amigável, porém explora apenas uma fração das possibilidades da ComFIRM.

A ferramenta ComFIRM está estável, e vem funcionando há mais de um ano na máquina do autor. Sua funcionalidade atual é considerada suficiente para a realização de inúmeros experimentos, porém deve ser vista como uma pedra fundamental, a ser estendida e melhorada. O aspecto modular e pouco intrusivo da ferramenta é um ponto extremamente importante para a continuidade de seu desenvolvimento.

6 Testes e experimentos

Este capítulo tem o objetivo de demonstrar, através de testes simples e experimentos mais complexos, o funcionamento da ferramenta descrita no capítulo anterior. A primeira seção trata de testes simples, com e sem a interface gráfica, e visa apenas demonstrar que a ferramenta funciona e implementa aquilo que foi proposto. Já a segunda seção contém dois experimentos reais com sistemas que utilizam alguma forma de comunicação confiável.

6.1 Testes simples de funcionamento

Esta seção traz três testes de funcionamento, que visam demonstrar que a ferramenta funciona corretamente e implementa o que foi proposto. O primeiro teste consiste em descartar, e numa pequena variação posterior duplicar, um a cada dois pacotes de rede. Serão selecionados somente pacotes do protocolo UDP com porta de destino 12345. A ferramenta de rede NetCat[1] será utilizada para gerar os pacotes.

O segundo teste não é determinístico, pois usará a ferramenta para descartar 25% dos pacotes enviados. Para demonstrar o caráter não determinístico deste teste, o mesmo será executado três vezes, e as três recepções serão demonstradas separadamente. Será possível perceber a ação sobre pacotes diferentes a cada execução do experimento.

Finalmente, o terceiro teste vai programar a ferramenta de forma que a máquina apresente uma falha de perda de capacidade de comunicação (o nodo não envia nem recebe nada) 5 segundos após o recebimento de um pacote específico. Para os demais nodos envolvidos, a máquina em teste apresentará falha de *crash*, visto que não poderá mais se comunicar. Como este teste envolve instruções mais complexas, que não são suportadas pela interface, ele será codificado manualmente. Todos os passos para a configuração e ativação manual da ferramenta serão mostrados, bem como a forma de obter o registro dos eventos de injeção de falhas manualmente.

6.1.1 Primeiro teste

A figura 6.1 mostra como a interface ComFIRM's Face está configurada para este experimento. Seleciona-se a atuação sobre a transmissão, protocolo UDP, porta 12345, ação descarte. Para iniciar o experimento, clica-se em "Apply changes", e então "ComFIRM Start". Serão então enviados dez pacotes UDP para esta porta, com a utilização da ferramenta NetCat tanto para envio como para recepção.

As figuras 6.2 e 6.3 mostram os comandos de envio e recepção, onde se percebe que somente um a cada dois pacotes foi recebido. A figura 6.4 mostra as informações do registro da ferramenta.

Este é um teste bastante simples, executado com a transmissão manual de dez pacotes UDP, cada um contendo um número de 1 a 10. Através do registro da ferramenta pode-se notar que a cada dois incrementos do contador, este recebe o valor 0 novamente, e a mensagem é descartada. Já aqui pode-se fazer uma pequena alteração, mudando-se a ação desempenhada pela ferramenta para duplicação de

[1] Informações em `http://www.l0pht.com/~weld/netcat/`.

FIGURA 6.1 – Configuração da ComFIRM's Face para o primeiro teste

```
root@thor:~# nc -u localhost 12345
1
2
3
4
5
6
7
8
9
10
root@thor:~#
```

FIGURA 6.2 – Transmissão dos pacotes do primeiro teste

```
root@thor:~# nc -p 12345 -l -u
1
3
5
7
9
root@thor:~#
```

FIGURA 6.3 – Recepção dos pacotes do primeiro teste

```
1620902: logging of action events is on
1621874: fault injection started
1622519: counter 0 incremented
1622602: counter 0 incremented
1622602: counter 0 set to 0
1622602: message dropped
1622834: counter 0 incremented
1622913: counter 0 incremented
1622913: counter 0 set to 0
1622913: message dropped
1622950: counter 0 incremented
1622980: counter 0 incremented
1622980: counter 0 set to 0
1622980: message dropped
1623014: counter 0 incremented
1623045: counter 0 incremented
1623045: counter 0 set to 0
1623045: message dropped
1623075: counter 0 incremented
1623173: counter 0 incremented
1623173: counter 0 set to 0
1623173: message dropped
1643360: fault injection stopped
1644100: ComFIRM has been reset
```

FIGURA 6.4 – Registro da ferramenta durante o primeiro teste

```
root@thor:~# nc -p 12345 -l -u
1
2
2
3
4
4
5
6
6
7
8
8
9
10
10
root@thor:~#
```

FIGURA 6.5 – Recepção dos pacotes na variação do primeiro teste

pacotes. A figura 6.5 mostra apenas a parte de recepção do experimento refeito, já que é a única que contém mudança.

Nota-se claramente que, a cada dois pacotes recebidos, o segundo vem duplicado. Até agora estes testes foram feitos em uma estação de trabalho relativamente ocupada, sem nenhuma interrupção dos demais serviços atualmente em uso na máquina, como HTTP, ICQ, IRC e SMTP. Isto acontece devido à capacidade da ferramenta de selecionar apenas o protocolo em estudo (neste caso, aquele que usar a porta 12345 do protocolo UDP) para a injeção de falhas.

Para esta variação do primeiro teste, a única diferença no registro da máquina foi a troca das mensagens "message dropped" por "message duplicated", indicando que as mensagens estavam sendo duplicadas. E com a mesma facilidade que o experimento é iniciado e alterado, ele é terminado com a opção "Stop" do menu "ComFIRM", e o estado da ferramenta é reiniciado com a opção "Reset" (últimas linhas da figura 6.4).

Os demais testes, em princípio, envolverão todos este mesmo padrão de envio de pacotes UDP para esta porta específica, portanto, a não ser que exista alguma diferença interessante, os envios de pacotes não serão mais demonstrados em figuras. Com isto se espera que os experimentos sejam de mais fácil compreensão e que os resultados sejam mais evidentes.

6.1.2 Segundo teste

Como foi dito no início do capítulo, este teste envolve o descarte não determinístico de 25% dos pacotes. Será utilizada a mesma estratégia de envio de dez pacotes UDP, e a ferramenta será reiniciada a cada novo teste. A figura 6.6 mostra como a interface ComFIRM's Face foi configurada para este experimento.

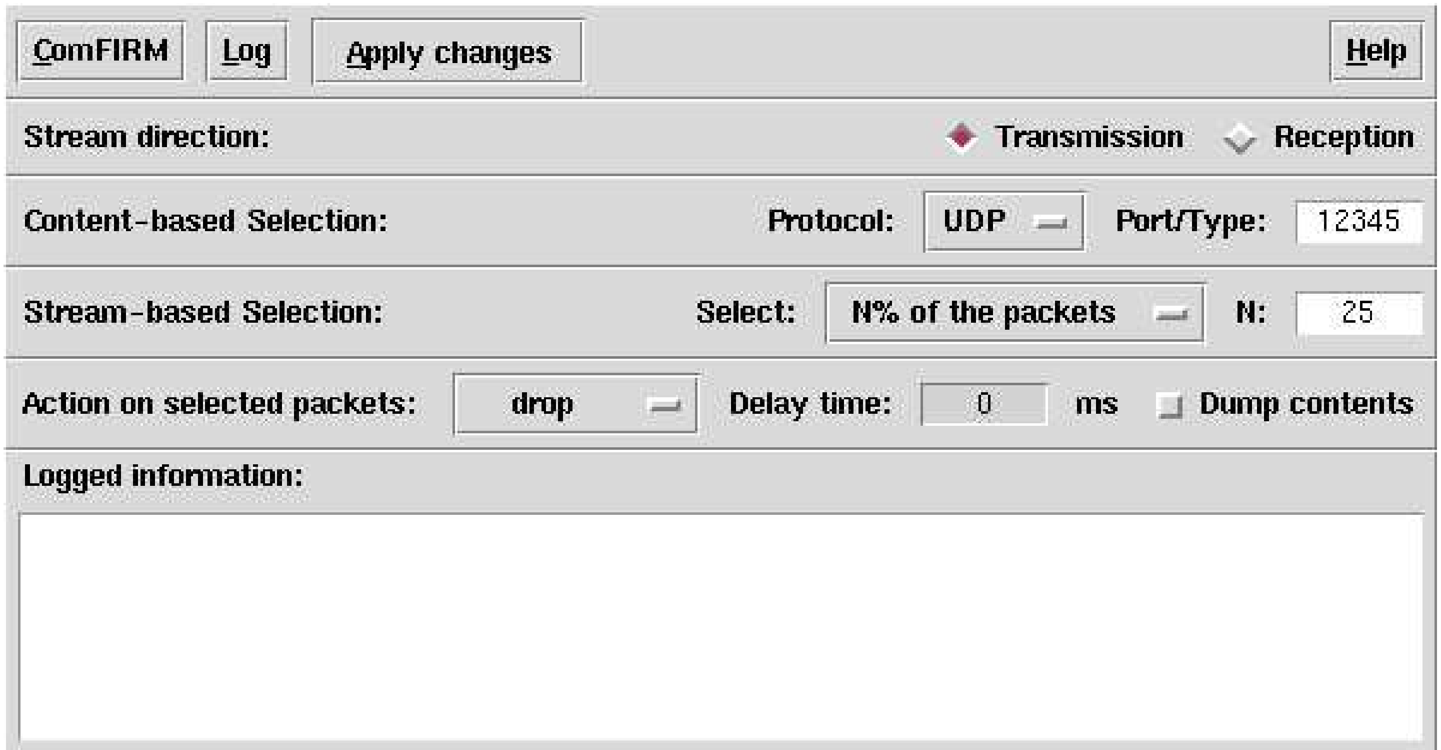

FIGURA 6.6 – Configuração da ComFIRM's Face para o segundo teste

A primeira execução deste teste, cuja recepção está demonstrada na figura 6.7 demonstra que na verdade 40% dos pacotes foram perdidos, porém isto apenas indica que dez pacotes é uma amostragem muito pequena para um teste probabilístico como este. Os demais poderão apresentar índices menores, que façam com que a média final fique em torno de 25%.

Já a segunda rodada deste teste exibiu uma taxa de descarte de mensagens de 30%, o que já está mais próximo dos 25% que foram configurados (figura 6.8). Finalmente, a figura 6.9 mostra uma recepção de 80% dos pacotes, o que indica que 20% das mensagens transmitidas foram descartadas. Nota-se claramente que este tipo de teste precisa de um maior fluxo de mensagens para ser efetivo. Ainda assim, por ser fácil perceber que mensagens diferentes foram selecionadas a cada execução do teste, verifica-se que a simulação não determinística de taxa de falhas funciona

```
root@thor:~# nc -p 12345 -l -u
2
4
7
8
9
10
root@thor:~#
```

FIGURA 6.7 – Recepção dos pacotes na primeira rodada do segundo teste

```
root@thor:~# nc -p 12345 -l -u
2
4
5
6
7
9
10
root@thor:~#
```

FIGURA 6.8 – Recepção dos pacotes na segunda rodada do segundo teste

```
root@thor:~# nc -p 12345 -l -u
1
3
4
5
6
7
9
10
root@thor:~#
```

FIGURA 6.9 – Recepção dos pacotes na terceira rodada do segundo teste

corretamente.

6.1.3 Terceiro teste

Este último teste não será realizado com a ajuda da interface gráfica, por utilizar instruções que esta não suporta. Para este teste, a ferramenta será programada para que a máquina apresente uma falha de perda de capacidade de comunicação (o nodo não envia nem recebe nada) 5 segundos após o recebimento de uma mensagem UDP para a porta 1234. Para os demais nodos envolvidos, a máquina em teste apresentará falha de *crash*, visto que não poderá mais se comunicar. Como este teste envolve instruções mais complexas, que não são suportadas pela interface, ele será codificado manualmente. Todos os passos para a configuração e ativação manual da ferramenta serão mostrados, bem como a forma de obter o registro dos eventos de injeção de falhas manualmente.

Para construir tal cenário de falhas, serão utilizadas uma regra de transmissão e duas regras de recepção, que funcionarão em conjunto. Serão necessários um sinalizador e um temporizador, além das instruções de descarte e de detecção da mensagem responsável por ativar o temporizador. O sinalizador servirá como um bloqueio para que as regras de descarte não sejam ativadas antes da configuração do temporizador, bem como para que a regra de configuração do temporizador seja

executada apenas uma vez quando da recepção do pacote sendo aguardado.

Tanto para transmissão quanto para recepção, será criada uma regra que, caso o sinalizador 0 tenha o valor 1, e o temporizador 0 tenha o valor 0, o pacote é descartado. Para completar o experimento, adiciona-se uma regra de recepção que testa se o protocolo da mensagem é UDP, se a porta destino é 1234 e se o sinalizador 0 tem valor 0, e então configura o temporizador 0 para 5 segundos, modo decrescente e coloca o valor 1 no sinalizador 0. Esta manipulação do sinalizador tem o significado de "comutar" o "conjunto" de regras sendo utilizado, visto que com o sinalizador em 1 esta regra não será mais avaliada, e as outras duas agora passarão a ser. Sendo assim, quando o temporizador chegar a zero, cinco segundos após sua ativação, qualquer pacote recebido ou transmitido será descartado.

As regras necessárias serão as seguintes:

Transmissão: testa se o sinalizador 0 tem valor 1, testa se o temporizador 0 tem valor 0 e descarta a mensagem;

Recepção 1: testa se o sinalizador 0 tem valor 1, testa se o temporizador 0 tem valor 0 e descarta a mensagem;

Recepção 2: testa se o sinalizador 0 tem valor 0, testa se o byte na posição 9 tem valor 17 (UDP), testa se a word na posição 0016h tem o valor 1234 (04d2h), configura o temporizador 0 para 5 segundos, configura o temporizador zero para o modo decrescente e então atribui o valor 1 ao sinalizador 0.

Codificando-se cada uma delas, obtém-se as seguintes seqüências hexadecimais (o primeiro byte representa o tamanho da regra):

Transmissão: 09 39 00 30 00 00 00 00 00 60

Recepção 1: 09 39 00 30 00 00 00 00 00 60

Recepção 2: 15 38 00 08 09 00 11 10 16 00 04 d2 80 00 f4 01 00 00 86 00 89 00

A figura 6.10 mostra como as seqüências são colocadas nos arquivos virtuais de regras e a ferramenta é ativada. Nota-se que foi incluído um byte nulo no final de cada seqüência, para indicar o término das regras, e também que foi adotada uma forma um pouco mais compacta para representar os bytes em hexadecimal. Após a ativação da ferramenta com o comando `ComFIRM on` o teste é conduzido em outras janelas, e então os comandos `off` e `reset` são enviados para desligar e reiniciar a ComFIRM, terminando a sessão de injeção de falhas.

A figura 6.11 mostra a forma com que se enviou dez pacotes UDP para a porta 1234 da máquina `thor`, um por segundo, e cada um com seu número de seqüência. A máquina utilizada para o envio destes pacotes é chamada `gus`, e está no mesmo segmento de rede. Já a figura 6.12 mostra a recepção dos pacotes, onde se vê que somente cinco pacotes foram aceitos, visto que o primeiro disparou o temporizador e então cinco segundos após este a máquina apresentou falha de *crash* (de comunicação).

Neste teste, a ferramenta foi configurada para descartar realmente qualquer pacote enviado ou recebido, mesmo aqueles que não faziam parte da seqüência de dez pacotes enviados para a porta UDP 1234. Sendo assim, no registro da ferramenta,

```
# perl -e 'print pack("H*", "09390030000000000006000");' \
        > ComFIRM_TX_Rules
# perl -e 'print pack("H*", "1538000809001110160004d28000f4010000" .
        "86008900093900300000000000006000");' > ComFIRM_RX_Rules
# echo log action on > ComFIRM_Control
# echo ComFIRM on > ComFIRM_Control
... execução do teste ...
# echo ComFIRM off > ComFIRM_Control
# echo ComFIRM reset > ComFIRM_Control
```

FIGURA 6.10 – Programação e ativação da ferramenta ComFIRM para o terceiro teste

```
root@gus:~$ exec 3>/dev/udp/thor/1234
root@gus:~$ for i in `seq 10`; do echo $i >&3; sleep 1; done
root@gus:~$ exec 3<&-
```

FIGURA 6.11 – Envio dos pacotes no terceiro teste

mostrado na figura 6.13, nota-se que após a expiração do temporizador mais de cinco pacotes foram descartados, pois mesmo a sessão de login via rede para a máquina `gus` foi afetada. Claro, como a ferramenta foi desligada logo após a realização do experimento, a sessão remota se recuperou, por utilizar o protocolo TCP.

Este teste demonstra como se pode utilizar conjuntos de regras de transmissão e recepção que interajam entre si, através dos recursos da própria ferramenta. Não seria difícil utilizar um sinalizador, por exemplo, para alternar entre atraso de pacotes no envio e na recepção, ou criar cenários complexos onde a transmissão de um certo padrão de pacotes cause a recepção duplicada de alguns outros.

Espera-se que com estes três testes bastante interessantes tenha ficado clara a flexibilidade que a ferramenta ComFIRM proporciona. Tendo finalizado então seus testes de funcionamento, onde ela demonstrou estar estável e permitir que se realize testes sem atrapalhar o uso normal da máquina em uso, passa-se agora para a realização de dois experimentos reais, onde o tráfego não mais será gerado

```
root@thor:~# nc -u -p 1234 -l
1
2
3
4
5
```

FIGURA 6.12 – Recepção dos pacotes no terceiro teste

```
root@thor:/proc/net# cat ComFIRM_Log
0010a4ba: logging of action events is on
0010a913: fault injection started
0012324d: timer 0 set to 500
0012324d: timer 0's mode set to 2
0012324d: flag 0 set to 1
00123445: message dropped
001234aa: message dropped
0012350f: message dropped
00123574: message dropped
001235d9: message dropped
0012363f: message dropped
00123653: message dropped
0012367b: message dropped
001236cb: message dropped
0012376b: message dropped
001238ab: message dropped
00123b2b: message dropped
00123c30: fault injection stopped
00123ed8: ComFIRM has been reset
```

FIGURA 6.13 – Registro da ferramenta no terceiro teste

manualmente, mas por ferramentas que realmente utilizam comunicação confiável.

6.2 Experimentos

Esta seção apresentará dois experimentos realizados com uma ferramenta que utiliza protocolos de comunicação confiável. O objetivo é demonstrar a utilização da ferramenta ComFIRM em situações reais, onde se precisa validar os protocolos de comunicação confiável implementados em uma dada aplicação.

Os experimentos utilizarão o gerenciador de *clusters* para o Sistema Operacional Linux chamado Heartbeat [ROB 2000], sendo que o primeiro testará sua capacidade para se recuperar de falhas de omissão. Já o segundo experimento testará sua capacidade de eliminar mensagens repetidas.

6.2.1 Primeiro experimento

Este experimento visa validar o protocolo de comunicação confiável implementado no Heartbeat, que é baseado em NACKs e tolera perda de mensagens através de pedidos de retransmissão. Para se testar esta sua característica, a ferramenta ComFIRM foi configurada de forma a descartar uma a cada 5 mensagens enviadas pelo Heartbeat.

A figura 6.14 mostra como a ferramenta ComFIRM foi configurada para este propósito. O Heartbeat foi configurado de forma a controlar um *cluster* de duas máquinas, `thor` e `gus`, e a ferramenta ComFIRM foi usada na máquina `thor`. Foi

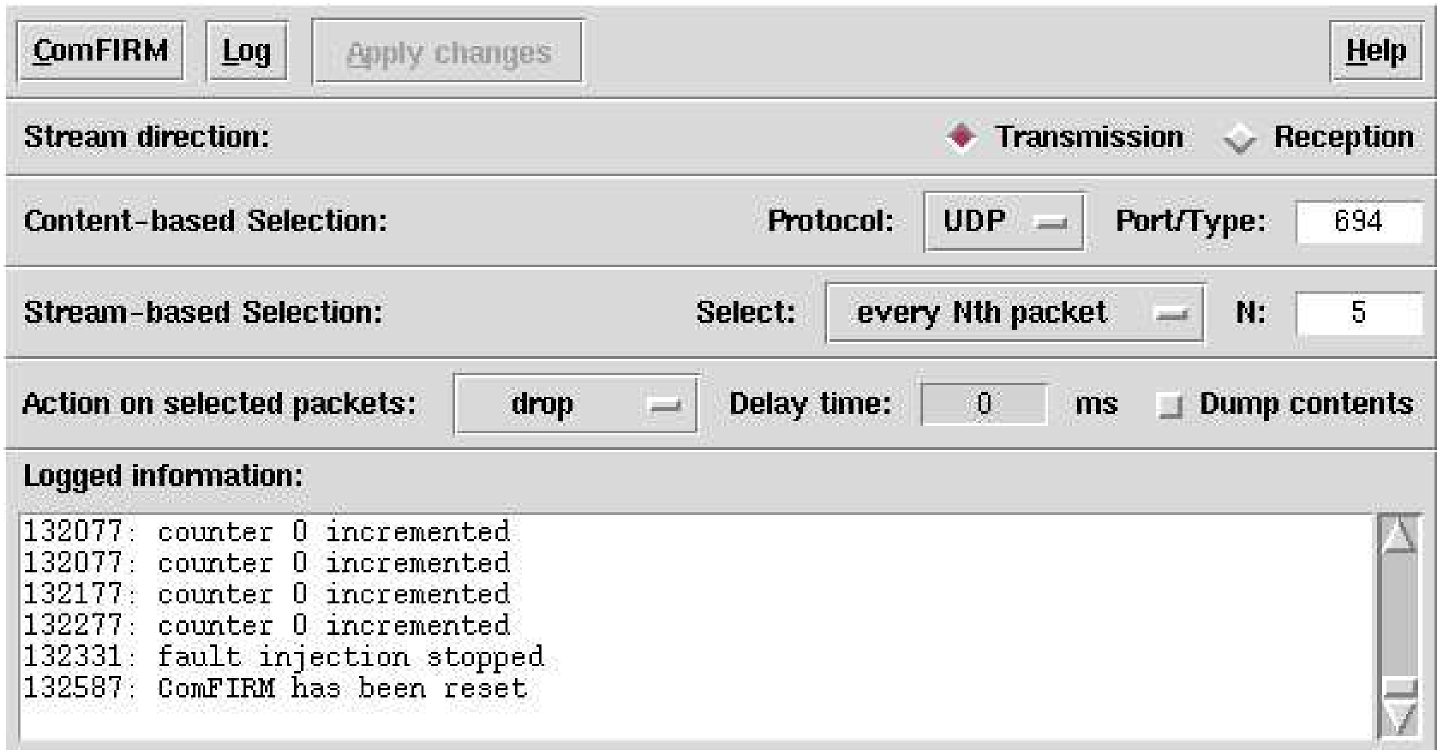

FIGURA 6.14 – Configuração da ComFIRM's Face para o primeiro experimento

selecionado o protocolo UDP e porta 694, que são usados pelo Heartbeat para suas comunicações.

Depois de configurada e ativada a ferramenta, o Heartbeat foi iniciado nas duas máquinas, e pode-se notar pela figura 6.15, contendo o registro do Heartbeat na máquina `gus`, que foram detectadas perdas de pacotes várias vezes. Nota-se que a máquina `gus` percebe uma falha a cada quatro pacotes que recebe, visto que a retransmissão dos pacotes perdidos pela máquina `thor` também são contados pela ferramenta.

Com isso já se percebe que o mecanismo implementado pelo Heartbeat, embora simples, funciona a contento. As mensagens "`info: No pkts missing from thor!`" indicam o término da recuperação após algum pedido de retransmissão.

Este experimento foi configurado e realizado em um intervalo de tempo de 7 minutos, o que demonstra a facilidade de uso da ferramenta ComFIRM.

Para completar a demonstração deste experimento, a figura 6.16 mostra o registro colhido da interface gráfica. Pode-se notar claramente o padrão de cinco incrementos do contador, sua reinicialização e o descarte da mensagem. O período de tempo preciso entre as ações da ferramenta também é interessante de ser notado, pois mostra que o Heartbeat possui uma resolução de tempo menor do que 10ms para suas transmissões (10ms é o intervalo de tempo entre interrupções do temporizador usado pelo Linux em plataformas Intel), e que o uso da ComFIRM não atrasou o seu funcionamento normal.

```
gus heartbeat[6426]: info: Node thor: status up
gus heartbeat[6426]: info: Node thor: status active
gus heartbeat[6426]: info: local resource transition completed.
gus heartbeat[6426]: info: remote resource transition completed.
gus heartbeat: info: Acquiring resource group: gus 10.0.17.26
gus heartbeat[6426]: ERROR: 1 lost packet(s) for [thor] [174:176]
gus heartbeat[6426]: info: No pkts missing from thor!
gus heartbeat[6426]: ERROR: 1 lost packet(s) for [thor] [178:180]
gus heartbeat[6426]: info: No pkts missing from thor!
gus heartbeat[6426]: ERROR: 1 lost packet(s) for [thor] [182:184]
gus heartbeat[6426]: info: No pkts missing from thor!
gus heartbeat[6426]: ERROR: 1 lost packet(s) for [thor] [186:188]
gus heartbeat[6426]: info: No pkts missing from thor!
gus heartbeat[6426]: ERROR: 1 lost packet(s) for [thor] [190:192]
gus heartbeat[6426]: info: No pkts missing from thor!
```

FIGURA 6.15 – Registro do Cluster Manager Heartbeat durante o primeiro experimento

6.2.2 Segundo experimento

Este segundo experimento objetiva demonstrar que o protocolo de comunicação utilizado pelo gerenciador de *clusters* Heartbeat elimina o problema de mensagens repetidas, através de um esquema de autenticação que também protege contra ataques de *replay* de agentes maliciosos na rede.

Neste experimento o Heartbeat foi configurado exatamente como no experimento anterior, com as máquinas `gus` e `thor` compondo o *cluster*. A ferramenta ComFIRM, situada na máquina `thor` foi configurada para que duplicasse um a cada três pacotes enviados pelo Heartbeat, como demonstrado na figura 6.17.

O registro colhido na máquina `gus` das ações do Heartbeat (figura 6.18) demonstra a detecção de vários pacotes com problemas de autenticação (a recepção de uma mensagem na ordem correta e com o código de autenticação correto invalida qualquer outra subseqüente duplicata). Como o Heartbeat consegue recuperar tal situação, se pode perceber no registro também as mensagens que decorrem da execução normal de suas funções.

Logo após o início do envio dos *heartbeats* pela porta UDP 694, pode-se observar que ambas as máquinas são percebidas como ativas. Aí mesmo já começam a chegar mensagens repetidas, que são registradas pelo Heartbeat como "*Invalid authentication type [2] in message!*". Pode-se notar que mesmo com o grande número de mensagens duplicadas detectadas, as demais mensagens mostram que a aplicação segue sendo executada corretamente, inclusive com a aquisição do grupo de recursos configurado.

Depois de uma breve espera para constatação de que realmente o Heartbeat estava operando corretamente, foi enviado o comando para que ele terminasse sua execução, para completar o experimento. Pelas últimas três mensagens pode-se perceber que mesmo com a duplicação de uma a cada três mensagens, o sistema foi capaz de corretamente terminar suas transações, realizando um *shutdown* limpo e

```
128914: logging of action events is on
129961: fault injection started
129977: counter 0 incremented
130077: counter 0 incremented
130177: counter 0 incremented
130277: counter 0 incremented
130377: counter 0 incremented
130377: counter 0 set to 0
130377: message dropped
130477: counter 0 incremented
130477: counter 0 incremented
130577: counter 0 incremented
130677: counter 0 incremented
130777: counter 0 incremented
130777: counter 0 set to 0
130777: message dropped
130877: counter 0 incremented
130877: counter 0 incremented
130977: counter 0 incremented
131077: counter 0 incremented
131177: counter 0 incremented
131177: counter 0 set to 0
131177: message dropped
131277: counter 0 incremented
131277: counter 0 incremented
131377: counter 0 incremented
131477: counter 0 incremented
131577: counter 0 incremented
131577: counter 0 set to 0
131577: message dropped
131677: counter 0 incremented
131677: counter 0 incremented
131777: counter 0 incremented
131877: counter 0 incremented
131977: counter 0 incremented
131977: counter 0 set to 0
131977: message dropped
132077: counter 0 incremented
132077: counter 0 incremented
132177: counter 0 incremented
132277: counter 0 incremented
132331: fault injection stopped
132587: ComFIRM has been reset
```

FIGURA 6.16 – Registro da ferramenta ComFIRM durante o primeiro experimento

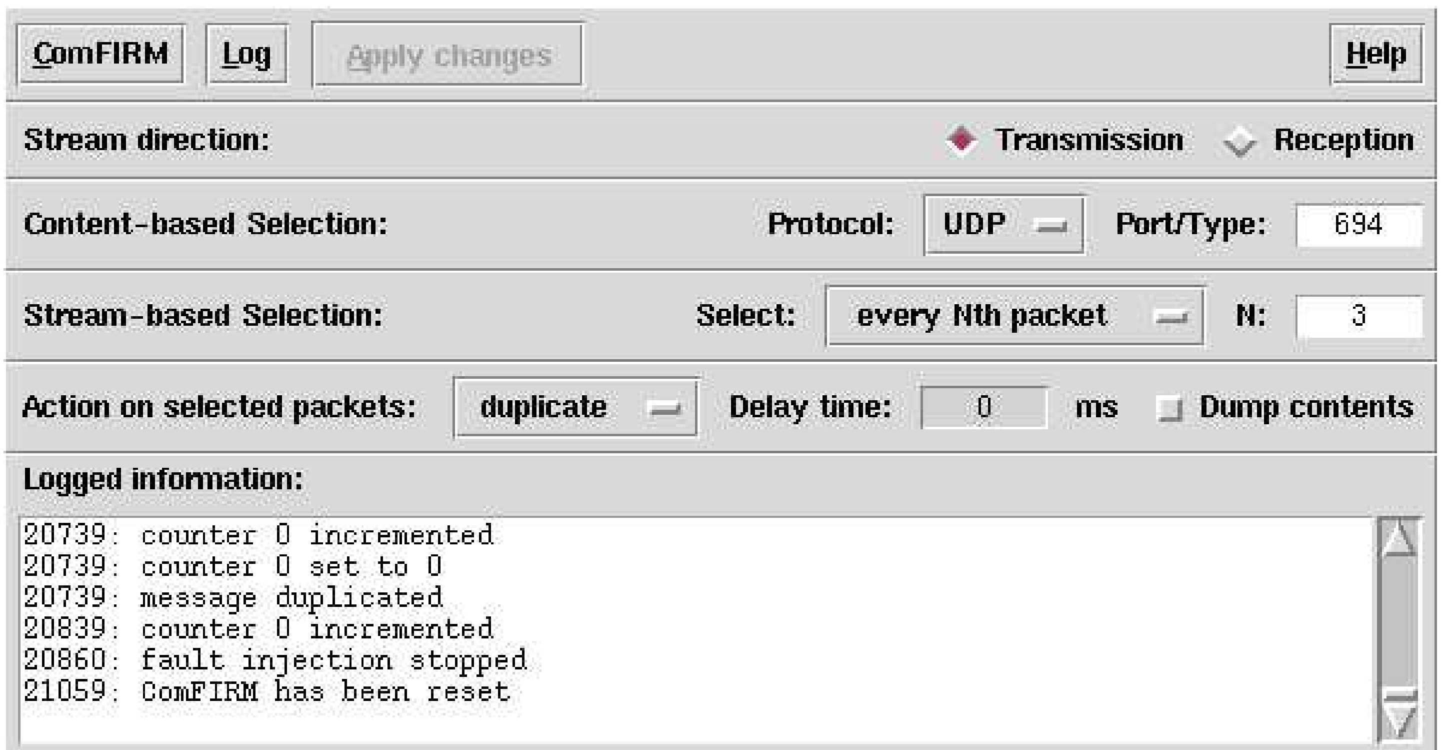

FIGURA 6.17 – Configuração da ComFIRM's Face para o segundo experimento

```
gus heartbeat: Configuration validated. Starting heartbeat 0.4.8e
gus heartbeat: info: Heartbeat generation: 1
gus heartbeat: UDP heartbeat started on port 694 interface eth0
gus heartbeat: info: Local status now set to: 'active'
gus heartbeat: info: Node thor: status active
gus heartbeat: ERROR: Invalid authentication type [2] in message!
gus last message repeated 9 times
gus heartbeat: info: local resource transition completed.
gus heartbeat: info: remote resource transition completed.
gus heartbeat: ERROR: Invalid authentication type [2] in message!
gus heartbeat: ERROR: Invalid authentication type [2] in message!
gus last message repeated 9 times
gus heartbeat: info: Acquiring resource group: gus 10.0.17.26
gus heartbeat: ERROR: Invalid authentication type [2] in message!
gus last message repeated 31 times
gus last message repeated 14 times
...
gus heartbeat: info: Heartbeat shutdown in progress.
gus heartbeat: info: Releasing resource group: gus 10.0.17.26
gus heartbeat: info: Heartbeat shutdown complete.
```

FIGURA 6.18 – Registro do Cluster Manager Heartbeat durante o segundo experimento

```
12339: logging of action events is on
13224: fault injection started
17240: counter 0 incremented
17240: counter 0 incremented
17306: counter 0 incremented
17306: counter 0 set to 0
17306: message duplicated
...
20539: counter 0 incremented
20639: counter 0 incremented
20739: counter 0 incremented
20739: counter 0 set to 0
20739: message duplicated
20839: counter 0 incremented
20860: fault injection stopped
21059: ComFIRM has been reset
```

FIGURA 6.19 – Registro da ferramenta ComFIRM durante o segundo experimento

sem problemas.

Para ilustração dos acontecimentos percebidos na máquina `thor`, a figura 6.19 mostra as mensagens geradas pela ferramenta, com algumas simplificações para poupar espaço. Com este experimento pode-se concluir que o *cluster manager* Heartbeat realmente suporta a operação em redes onde aconteça a duplicação de mensagens, bem como é imune a ataques de *replay*, onde um agente malicioso na rede tenta capturar e enviar novamente em outros momentos algumas mensagens.

6.3 Conclusão

Este capítulo demonstrou que a ferramenta ComFIRM cumpre satisfatoriamente seus objetivos, apresentando uma abordagem flexível, poderosa e efetiva para a injeção de falhas de comunicação. Foram realizados três testes satisfatórios, onde o modo de operação da ferramenta, tanto via interface gráfica quanto manualmente, foi explicado e documentado.

Os dois experimentos realizados demonstram a utilização da ferramenta em situações reais, onde não houve nenhuma preparação feita na aplicação sendo testada. Através de seus registros, bem como pelo registro da ComFIRM, foi possível perceber as falhas sendo injetadas, o que valida a utilização da ferramenta. Como efeito colateral, já se pode afirmar também que a ferramenta de gerência de *clusters* Heartbeat possui protocolos de comunicação confiável que toleram falhas de omissão e duplicação de pacotes.

Ainda existem muitas possibilidades de melhorias e extensões à ferramenta, as quais serão exploradas no próximo capítulo. Espera-se que a ferramenta se torne um padrão para injeção de falhas de comunicação no Sistema Operacional Linux, e que evolua cada vez mais.

7 Conclusões

Este capítulo finaliza este trabalho resumindo o que foi feito, o que se aprendeu, quais foram os problemas enfrentados, como foram resolvidos, quais são os trabalhos futuros e as outras possibilidades de se fazer injeção de falhas de comunicação sobre o Linux, que foram percebidos durante a execução deste trabalho.

7.1 Resultados

Este trabalho apresentou um estudo abrangente sobre as possibilidades da técnica de validação experimental de protocolos de comunicação confiável através da injeção de falhas de comunicação. Foram estudadas as técnicas de injeção de falhas por hardware, software e simulação, e foi estudada de forma especialmente aprofundada a injeção de falhas por software.

Foram estudadas as formas de se implementar injetores de falhas em software, em vários níveis de atuação e interferência com o objeto em teste. Como este trabalho objetivou a implementação de um injetor de falhas de comunicação no nível do sistema operacional, foi dada especial atenção a esta abordagem. Conclui-se que a injeção de falhas neste nível é, do ponto de vista de um processo, indiferente da ocorrência de falhas reais, o que torna esta técnica bastante interessante.

Dependendo do nível onde se coloca o injetor de falhas dentro do sistema operacional, pode-se injetar falhas mesmo em código do próprio sistema. Este é o caso da ferramenta implementada, a ComFIRM, que pode até mesmo validar a implementação de um protocolo com confiabilidade inerente, como o TCP. Ao se injetar falhas em uma conexão TCP, a própria implementação do protocolo se encarrega da correção das falhas, e tais eventos jamais chegam ao nível de aplicação (salvo, claro, em caso de *crash*, onde depois de um longo período de espera o processo é avisado da impossibilidade de estabelecer comunicação).

Foram estudadas também as formas de se especificar um experimento de injeção de falhas, e os modelos de falhas comumente encontrados na bibliografia de Tolerância a Falhas. Modelos como os de Christian [CHR 86] são bastante fáceis de se produzir com a ferramenta resultante deste trabalho. Um estudo da ferramenta ORCHESTRA foi conduzido, por ela atingir muitos dos objetivos esperados para a ComFIRM, apenas em um ambiente operacional diferente.

O núcleo do Sistema Operacional Linux foi estudado, em sua arquitetura e detalhes de implementação. Como a ferramenta ComFIRM foi proposta estando situada no núcleo do Linux, foram estudados os vários mecanismos do núcleo que seriam necessários à sua implementação. Para se ter uma visão adequada do conjunto de subsistemas que forma o Linux, foram estudadas suas arquiteturas conceitual e concreta.

Especialmente importante foi o estudo do Virtual Filesystem Switch e da implementação do sistema de arquivos virtual `proc`, bem como do Subsistema de Rede e de sua estrutura de dados mais importante, o `sk_buff`. Também foram estudadas as várias funções e rotinas de apoio à manipulação destas estruturas, e os vários mecanismos que o *kernel* implementa para facilitar o desenvolvimento de novas funcionalidades, como *task queues* e *bottom-half handlers*.

Foi especificada e implementada uma ferramenta de injeção de falhas de co-

municação situada dentro do núcleo do Sistema Operacional Linux, chamada ComFIRM, assim como uma interface gráfica experimental destinada a facilitar o uso da ComFIRM. A ComFIRM — *Communication Fault Injection through Operating System Resource Modification* — é uma ferramenta flexível e poderosa, permitindo a especificação de experimentos de injeção de falhas de comunicação através de regras de transmissão e recepção, compostas de instruções primitivas de seleção e manipulação de mensagens.

A operação da ferramenta se dá através de quatro arquivos virtuais situados normalmente no diretório `/proc/net`. A ferramenta possui um arquivo para registro do experimento, um para comandos de controle da ferramenta, um para abrigar as regras a serem avaliadas na transmissão de um pacote e um para as regras de recepção. A manipulação de tais arquivos é idêntica a de arquivos comuns de um sistema de arquivos real, e a interface gráfica na verdade apenas facilita a utilização destes mesmos arquivos.

Foram realizados alguns testes, os quais comprovaram o correto funcionamento da ferramenta, tanto usando a interface gráfica como através da configuração manual da ferramenta. Depois de concluídos os testes, foram conduzidos experimentos que comprovaram a eficácia do uso da ferramenta na validação experimental de aplicações que implementam protocolos de comunicação confiável.

A ferramenta gerada é licenciada como software livre, sob a licença GNU General Public License, de forma a atingir o maior público possível. Espera-se que o código gerado seja utilizado por pesquisadores que escolherem o Sistema Operacional Linux como plataforma para seus projetos envolvendo comunicação confiável. Como resultado do licenciamento do código como software livre, e sua distribuição através do site `http://www.inf.ufrgs.br/~olive`, espera-se que o código seja continuamente expandido e melhorado, e que seja considerado uma boa adição ao patrimônio de software livre já existente.

7.2 Experiência adquirida

Durante a execução deste trabalho foram estudados assuntos bastante diversos, desde o puramente teórico até detalhes muito específicos da implementação do núcleo do Sistema Operacional Linux. Absorver a grande quantidade de informações necessárias à concretização deste trabalho com certeza é uma experiência transformadora.

Estudando as três formas de injeção de falhas, por hardware, software e simulação, pode-se perceber como o universo dos sistemas distribuídos é uma grande ilusão. Processos e máquinas criam conexões virtuais, através do envio de mensagens por dispositivos de comunicação, e se relacionam apenas através destas mensagens. Qualquer manipulação destas mensagens tem impacto direto no funcionamento dos processos, e estes nada tem a fazer contra isso. Sua visão do mundo exterior passa pela "janela" do sistema operacional, e este pode aplicar a alteração que quiser no ambiente do processo.

A injeção de falhas por hardware é trabalhosa e cara, e dificilmente será vista sendo usada em grande escala em ambientes acadêmicos com falta de recursos, como é a situação em geral no Brasil. Já as outras duas possibilidades não precisam carregar o peso do desenvolvimento de hardware específico, e portanto podem ser

aplicadas diretamente.

Através da injeção de falhas por simulação se pode validar a especificação de um protocolo antes mesmo de sua implementação. Uma vez validado e implementado o protocolo, a injeção de falhas por software permite sua validação experimental, garantindo que a implementação atinja o previsto na implementação. E com os avanços dos sistemas operacionais livres existentes hoje em dia, se pode, por exemplo, simular em apenas uma máquina com Linux uma rede virtual de outras máquinas, também virtuais, através do User-Mode Linux[1].

O User-Mode Linux é um *kernel* que roda como processo de usuário, criando uma máquina dentro de outra. Assim todo um experimento envolvendo várias máquinas virtuais pode ser conduzido em apenas uma máquina física. *Crashes* nas máquinas virtuais não afetam a máquina física.

A injeção de falhas de comunicação por software não é um procedimento difícil de ser realizado, por poder acontecer em momentos bem definidos dentro da execução de um processo, e por operar em pacotes de dados bem encapsulados. Quando um processo transmite uma mensagem, ele encapsula um conjunto de dados e o repassa ao sistema operacional para transmissão. O que acontece durante a operação de transmissão é transparente ao processo, e isto dá uma grande oportunidade de se manipular a mensagem sendo transmitida em um contexto separado daquele do processo.

Ao utilizar uma chamada de sistema em um sistema operacional multitarefa, já se sabe que o funcionamento normal acarreta o bloqueio do processo por algum tempo, até que a chamada retorne e o escalonador volte a selecionar este processo para execução. Durante este tempo o processo está congelado, e pode ficar neste estado por tanto tempo quanto se desejar. Sendo assim, não importa o que o sistema operacional faz enquanto ele espera, nem importa que o sistema operacional deliberadamente cause uma falha na operação, enquanto "mente" ao processo que tudo está bem.

Adicionar funcionalidades ao núcleo do Linux se mostrou ser uma tarefa razoavelmente fácil, porém com muitas armadilhas que somente a experiência permite evitar. Nas primeiras versões da ComFIRM, o atraso de pacotes funcionava na verdade como um *crash* programado da máquina. É difícil saber onde colocar pedaços de código não triviais, como o código da ComFIRM que atualiza os temporizadores. Até hoje o autor se pergunta se não há uma maneira mais "limpa" de fazer isto.

Por outro lado, existe muita documentação sobre o funcionamento do núcleo do Linux disponível de forma impressa e eletrônica. Na pior das hipóteses, uma leitura cuidadosa do código fonte resolve qualquer dúvida. Também é notável a disposição que os demais programadores do núcleo do Linux têm para ajudar pessoas que estão iniciando. Sempre há alguém disposto a resolver dúvidas, problemas e eventualmente até corrigir o código de um iniciante.

O sentimento que fica ao se conceber e implementar um projeto interessante e liberá-lo como software livre é o de poder mudar o mundo, ou pelo menos de fazer parte de algo maior. É certamente uma experiência quase religiosa. Aquilo que se produz como software livre é patrimônio da humanidade, e mesmo que não venha a ser largamente usado, pode influenciar outros projetos, e ter uma vida muito longa. Nada disso seria possível sem a idéia do software livre. Software livre é poder.

[1]Informações em `http://user-mode-linux.sourceforge.net`.

7.3 Dificuldades encontradas

As dificuldades encontradas durante a realização deste trabalho foram quase totalmente práticas. Algumas questões teóricas ainda esperam uma resposta adequada, claro. A ferramenta ComFIRM é poderosa e flexível, porém determinadas operações ainda podem ser demasiadamente complexas.

Por exemplo, as instruções de seleção baseada no conteúdo da mensagem são interessantes, e realmente permitem que se tome decisões baseadas em qualquer bit, byte, word ou doubleword da mensagem. Com isto se pode selecionar pacotes de protocolos específicos, e até mesmo pacotes com esta ou aquela *flag* que se deseja testar. Mas o protocolo IP, por exemplo, tem um cabeçalho de tamanho variável; algumas de suas opções podem estender o cabeçalho em alguns bytes.

Isso quer dizer que os testes com deslocamentos absolutos usados na ComFIRM nem sempre testarão aquilo que se pensa estar testando. Isto foi uma decisão consciente de projeto, ao situar a ferramenta ComFIRM em um ponto independente de protocolos do núcleo do Linux. Abordagens mais intimamente ligadas a um protocolo específico podem, obviamente, testar com precisão qualquer um dos campos em seu cabeçalho, porém também seriam menos flexíveis. Uma das possibilidades de trabalhos futuros citados é exatamente a colocação da ferramenta em pontos mais próximos dos protocolos de rede e transporte, de forma a facilitar suas capacidades de “foco” em um protocolo específico. Um usuário que necessite da flexibilidade atual poderá sempre utilizar a ferramenta em seu corrente estado.

Outra dificuldade encontrada, com relação à parte mais teórica da injeção de falhas de comunicação baseada em regras, foi com relação ao conjunto de instruções mínimo necessário à utilização da ferramenta. Sempre há a vontade de incluir mais e mais instruções, cada vez mais complexas. Um exemplo que quase foi implementado é uma instrução de procura de cadeias de caracteres dentro do pacote sendo analisado. Tais instruções, embora interessantes e úteis, teriam um *overhead* muito alto principalmente nas regras de recepção, onde se está num contexto de interrupção. Espera-se que com o conjunto atualmente implementado na ComFIRM se possa realizar a maioria dos testes de injeção de falhas.

Passando-se às dificuldades de implementação, pode-se citar como grande fator complicador o correto entendimento da operação dos `sk_buffs` e suas listas (ou a falta deste entendimento). Principalmente em estudos anteriores, utilizando as versões 2.0.x do Linux, a operação com estas estruturas dependia do entendimento da função das muitas listas de `sk_buffs` internas ao núcleo. Retirar um pacote das muitas listas das quais ele poderia fazer parte ao, por exemplo, descartar um pacote, era uma operação bastante complexa.

Os *kernels* da versão 2.2.x são mais bem organizados neste ponto, e portanto um dos poucos pontos mais complexos da implementação da versão atual da ComFIRM foi o atraso de pacotes. Este por muito tempo causava o *crash* da máquina quando da tentativa de transmitir ou receber o pacote depois da expiração do temporizador. Ainda hoje este ponto da implementação é considerado um pouco instável, embora a ferramenta já venha sendo usada há meses, e por vezes com *uptime* da máquina superior a algumas semanas, sem nenhum sinal de problemas causados pela ferramenta.

Com o tempo o Linux está se tornando mais e mais depurado, e suas estruturas internas mais organizadas e documentadas. Sendo assim, um esforço de

implementação de funcionalidades em núcleos mais novos com certeza será menos conturbado.

A escolha de projeto de requerer um byte nulo como marcador de final das regras, e ainda requerer que o usuário o inclua foi um erro. Quando se deseja um teste rápido através da configuração manual da ferramenta, freqüentemente se esquece deste byte, e se é brindado com *crashes* dos mais variados tipos. Uma melhoria óbvia, e que será a primeira alteração da ferramenta após o término deste trabalho, será fazer com que a própria ferramenta aloque um byte a mais quando da escrita das regras, e nele coloque o valor zero, forçando o final da avaliação das regras mesmo quando o usuário se esquece dele.

7.4 Trabalhos futuros

É comum se ouvir nos círculos da Ciência da Computação que um software nunca está terminado, sempre há mais alguma característica a se adicionar. O código da ComFIRM está estável e funcional, porém existem vários pontos ainda a se melhorar ou estender.

Uma melhoria que já foi citada na seção anterior é a de não requerer que o usuário se lembre de adicionar um byte nulo no final das regras. É muito fácil fazer com que o código aloque um byte a mais que o especificado na escrita das regras e o preencha com o valor zero. Caso o usuário já o tenha colocado, se desperdiça um byte, o que é irrisório. Caso tenha sido esquecido, elimina-se uma boa possibilidade de *crash* da máquina. Realmente não há razão que deponha contra esta adição.

O código da ComFIRM foi criado em uma máquina monoprocessada, e o autor não tem experiência em programação para máquinas multiprocessadas. Isto quer dizer que o código da ComFIRM possivelmente contém várias condições de corrida e outros problemas típicos da falta de sincronização adequada em tal ambiente. Sendo assim, outra necessidade de trabalho futuro é a revisão do código da ComFIRM adicionando semáforos e *spinlocks* onde for necessário.

O sistema de registro (*log*) da ferramenta funciona corretamente, porém pode ser muito melhorado. Atualmente cada linha de texto gerada é colocada em uma estrutura de tamanho fixo, e as estruturas são ordenadas em uma lista ligada. Observando-se a implementação de *pipes* e arquivos do tipo FIFO pode-se obter boas idéias para uma nova implementação do sistema de registro, que se aproxime mais de modelos já utilizados por estes outros tipos de arquivos que apresentam características semelhantes.

Os arquivos virtuais situados em `/proc/net` apresentam somente a funcionalidade básica para o funcionamento da ferramenta. As regras devem ser escritas todas de uma vez, em apenas uma operação de escrita, a partir do deslocamento zero do arquivo. Isto limita um pouco a flexibilidade e a usabilidade da ferramenta, portanto esta funcionalidade deve ser melhorada. Novamente um estudo mais detalhado da implementação dos demais arquivos do `/proc` deve ser suficiente para se pensar uma maneira mais natural de implementar os arquivos virtuais da ComFIRM.

Outra maneira interessante de expandir a funcionalidade da ComFIRM está na adição de novos comandos ao seu conjunto já existente. De imediato pode-se pensar em comandos mais específicos para os protocolos mais usados, como TCP, UDP, ICMP e IP. Um comando que testasse o endereço IP de origem ou destino, e

um comando que testasse portas UDP ou TCP seriam adições interessantes, se diferenciando dos demais testes do conteúdo da mensagem por conhecerem os protocolos, portanto não sofrendo do problema de cabeçalhos com tamanho variável.

Como a ferramenta é flexível, tais comandos precisariam apenas ser inseridos nos arquivos de cabeçalho e de código da ferramenta, sem interferir com nenhum outro código. Com o esquema escolhido de codificação das instruções, ainda restam valores que possam ser usados para representar as novas instruções, e as antigas não seriam atrapalhadas pelas novas.

Uma outra possibilidade bastante interessante é a de tornar a ComFIRM um módulo do novo código de *firewalling* do Linux, o NetFilter. Este código permite que se defina regras para a filtragem de pacotes, de uma maneira muito parecida com a da ComFIRM. Regras são criadas com uma série de condições de seleção, e então uma (e apenas uma) operação de manipulação da mensagem. As condições de seleção são todas específicas para seus protocolos, e portanto podem achar a informação que necessitam onde estiver, o que é uma das deficiências da ComFIRM.

Por outro lado, o NetFilter não possui nenhum tipo de seleção baseada no fluxo de mensagens ou em elementos externos, portanto se a ComFIRM pudesse ser colocada como um módulo de manipulação do pacote, poder-se-ia selecionar pacotes com os selecionadores do NetFilter e então passá-los à avaliação de regras da ComFIRM. Tal conjunto seria de uma flexibilidade e poder muito interessantes. O NetFilter garantiria o foco no protocolo em estudo, e a ComFIRM proveria seus contadores, sinalizadores e temporizadores para a seleção baseada nestes outros elementos.

Pode-se pensar ainda muitas outras extensões e adaptações para a ComFIRM, posto que o código é modular e enxuto. Como o seu código é distribuído livremente sob a licença pública GPL, espera-se que ela seja bem difundida e que seja usada e reutilizada por muito tempo.

Anexo 1 Código da ferramenta ComFIRM

Nas seguintes seções é apresentado o código gerado para implementar a ferramenta ComFIRM, em sua versão 0.5, dentro do núcleo do Sistema Operacional Linux. Esta versão foi considerada estável e completa o suficiente para a elaboração deste texto. O código abaixo relacionado está em formato *diff*, quando houve alteração de arquivos existentes, visando apresentar somente as diferenças entre o código original do Linux (versão 2.2.9) e o código com a ferramenta inserida. Os arquivos que foram criados têm seus conteúdos simplesmente listados.

O leitor mais atento poderá encontrar pequenas diferenças entre o código aqui encontrado e o código presente nos arquivos *diff* reais, porém tais alterações têm apenas o intuito de melhor formatar o código para impressão e visualização por humanos, não incorrendo em qualquer alteração sintática ou semântica.

A.1 net/core/Makefile

```
--- /usr/src/linux-2.2.9/net/core/Makefile       Tue Dec 29 17:21:49 1998
+++ /usr/src/linux-2.2.9-comfirm/net/core/Makefile       Fri Jun 18 04:12:41 1999
@@ -9,7 +9,7 @@

 O_TARGET := core.o

-O_OBJS        := sock.o skbuff.o iovec.o datagram.o scm.o
+O_OBJS        := sock.o skbuff.o iovec.o datagram.o scm.o comfirm.o

 ifeq ($(CONFIG_SYSCTL),y)
 ifeq ($(CONFIG_NET),y)
```

A.2 include/linux/proc_fs.h

```
--- /usr/src/linux-2.2.9/include/linux/proc_fs.h       Fri May 21 02:30:00 1999
+++ /usr/src/linux-2.2.9-comfirm/include/linux/proc_fs.h       Wed Dec 29 01:08:52 1999
@@ -146,6 +146,10 @@
        PROC_NET_IPFW_CHAIN_NAMES,
        PROC_NET_AT_AARP,
        PROC_NET_BRIDGE,
+       PROC_NET_COMFIRM_CONTROL,
+       PROC_NET_COMFIRM_TX_RULES,
+       PROC_NET_COMFIRM_RX_RULES,
+       PROC_NET_COMFIRM_LOG,
        PROC_NET_LAST
 };
```

A.3 kernel/sched.c

```
--- /usr/src/linux-2.2.9/kernel/sched.c       Mon May 10 13:55:21 1999
+++ /usr/src/linux-2.2.9-comfirm/kernel/sched.c       Tue Aug 31 02:33:03 1999
@@ -40,6 +40,8 @@

 #include <linux/timex.h>

+#include <linux/comfirm.h>
+
 /*
  * kernel variables
  */
@@ -1520,9 +1522,18 @@
```

```
 static void timer_bh(void)
 {
+        int i;
+
         update_times();
         run_old_timers();
         run_timer_list();
+
+        if (ComFIRM_Active)                /* Run ComFIRM timers */
+          for (i = 0; i < ComFIRM_Max_Timers; i++)
+            if (ComFIRM_Timers[i].mode == ComFIRM_Timer_Mode_Inc)
+              ComFIRM_Timers[i].time++;
+            else if (ComFIRM_Timers[i].mode == ComFIRM_Timer_Mode_Dec && ComFIRM_Timers[i].time > 0)
+              ComFIRM_Timers[i].time--;
 }

 void do_timer(struct pt_regs * regs)
```

A.4 net/core/dev.c

```
--- /usr/src/linux-2.2.9/net/core/dev.c        Thu Mar 25 14:23:34 1999
+++ /usr/src/linux-2.2.9-comfirm/net/core/dev.c        Sun Jul 25 23:57:36 1999
@@ -93,6 +93,7 @@
 #ifdef CONFIG_PLIP
 extern int plip_init(void);
 #endif
+#include <linux/comfirm.h>

 NET_PROFILE_DEFINE(dev_queue_xmit)
 NET_PROFILE_DEFINE(net_bh)
@@ -573,7 +574,7 @@
         netif_rx(newskb);
 }

-int dev_queue_xmit(struct sk_buff *skb)
+int old_dev_queue_xmit(struct sk_buff *skb)
 {
         struct device *dev = skb->dev;
         struct Qdisc  *q;
@@ -634,6 +635,18 @@
         return 0;
 }

+/*
+ * This dummy wrapper is used in ComFIRM so that we don't need to
+ * mess with the intricate logic/timing of dev_queue_xmit.
+ */
+
+int dev_queue_xmit(struct sk_buff *skb) {
+  if (ComFIRM_Active && ComFIRM_TX_Rules_Size > 0) /* Only call ComFIRM if it's active and has TX rules */
+    return ComFIRM_Apply_TX_Rules(skb);
+  else
+    return old_dev_queue_xmit(skb);
+}
+

 /*=======================================================================
                         Receiver rotutines
@@ -753,7 +766,7 @@
  *        (protocol) levels.  It always succeeds.
  */

-void netif_rx(struct sk_buff *skb)
+void old_netif_rx(struct sk_buff *skb)
 {
 #ifndef CONFIG_CPU_IS_SLOW
         if(skb->stamp.tv_sec==0)
@@ -792,6 +805,18 @@
         kfree_skb(skb);
 }
```

```
+/*
+ * This dummy wrapper is used in ComFIRM so that we don't need to
+ * mess with the intricate logic/timing of netif_rx.
+ */
+
+void netif_rx(struct sk_buff *skb) {
+  if (ComFIRM_Active && ComFIRM_RX_Rules_Size > 0) /* Only call ComFIRM if it's active and has RX Rules */
+    ComFIRM_Apply_RX_Rules(skb);
+  else
+    old_netif_rx(skb);
+}
+
 #ifdef CONFIG_BRIDGE
 static inline void handle_bridge(struct sk_buff *skb, unsigned short type)
 {
@@ -2020,6 +2045,6 @@
 #ifdef CONFIG_IP_PNP
         ip_auto_config();
 #endif
-
+        ComFIRM_Init();
        return 0;
 }
```

A.5 include/linux/comfirm.h

```
/*
 * ComFIRM Header File
 * Author: Fábio Olivé Leite <olive@inf.ufrgs.br>
 * Advisor: Taisy Silva Weber <taisy@inf.ufrgs.br>
 *
 * Copyright 1999, 2000 by Fábio Olivé Leite.
 * This code is licensed under the GNU General Public License V. 2.
 * Consult http://www.gnu.org/ for details.
 *
 * Version 0.5 - Remember to increase this when updating,
 *               and also check the "log version" command
 */

#ifndef _COMFIRM_H
#define _COMFIRM_H

#include <linux/skbuff.h>

#define ComFIRM_Log_Open      0x00000001 /* The file ComFIRM_Log is open                    */
#define ComFIRM_Log_Selection 0x00000002 /* Log message selection information via ComFIRM_Log */
/* After some consideration I've decided logging selection information is not very useful, */
/* so it won't be added. Let's keep the #define, anyway. */
#define ComFIRM_Log_Action    0x00000004 /* Log message actions via ComFIRM_Log             */
#define ComFIRM_Log_Debug     0x00000008 /* Log debugging messages via syslog               */

int ComFIRM_Apply_TX_Rules(struct sk_buff *);
void ComFIRM_Apply_RX_Rules(struct sk_buff *);
void ComFIRM_Init(void);
int old_dev_queue_xmit(struct sk_buff *);
void old_netif_rx(struct sk_buff *);
void get_random_bytes(void *, int);

extern int ComFIRM_Active;
extern int ComFIRM_TX_Rules_Size;
extern int ComFIRM_RX_Rules_Size;

/* Rule format: nn aa..zz where
 *   nn is a byte with the size of the rule and
 *   aa..zz are the bytes that compose the rule
 * To use several rules just stick them together and end with a null rule (just 00)
 */

#define ComFIRM_Inst_Type_Mask 0xf8 /* That is, five bits compose the opcode */
#define ComFIRM_Inst_Mode_Mask 0x07 /*  and three bits for the "opmode" :) */
```

```
#define ComFIRM_Test_Bit       0x00
#define ComFIRM_Test_Byte      0x08
#define ComFIRM_Test_Word      0x10
#define ComFIRM_Test_Double    0x18
#define ComFIRM_Test_Counter   0x20
#define ComFIRM_Test_RandByte  0x28
#define ComFIRM_Test_Timer     0x30
#define ComFIRM_Test_Flag      0x38

#define ComFIRM_Action_Bit     0x40
#define ComFIRM_Action_Byte    0x48
#define ComFIRM_Action_Word    0x50
#define ComFIRM_Action_Double  0x58
#define ComFIRM_Action_Drop    0x60
#define ComFIRM_Action_Delay   0x68
#define ComFIRM_Action_Dup     0x70
#define ComFIRM_Action_Counter 0x78
#define ComFIRM_Action_Timer   0x80
#define ComFIRM_Action_Flag    0x88
#define ComFIRM_Action_Dump    0x90

#define ComFIRM_Test_Equal     0x00
#define ComFIRM_Test_Greater   0x01
#define ComFIRM_Test_Less      0x02
#define ComFIRM_Test_Bit0      0x00
#define ComFIRM_Test_Bit1      0x01

#define ComFIRM_Action_Set     0x00
#define ComFIRM_Action_Inc     0x01
#define ComFIRM_Action_Dec     0x02
#define ComFIRM_Action_Bit0    0x00
#define ComFIRM_Action_Bit1    0x01
#define ComFIRM_Action_BitC    0x02
#define ComFIRM_Action_Timer_Mode_None 0x04
#define ComFIRM_Action_Timer_Mode_Inc  0x05
#define ComFIRM_Action_Timer_Mode_Dec  0x06

/*
 * Timers and their defines
 */

struct ComFIRM_Timer {
  int time;
  char mode;
};

#define ComFIRM_Timer_Mode_None 0x00
#define ComFIRM_Timer_Mode_Inc  0x01
#define ComFIRM_Timer_Mode_Dec  0x02

extern struct ComFIRM_Timer ComFIRM_Timers[];

/*
 * Static limits
 */

#define ComFIRM_Max_Counters 8
#define ComFIRM_Max_Timers   8
#define ComFIRM_Max_Flags    32

#endif /* _COMFIRM_H */
```

A.6 net/core/comfirm.c

```
/*
 * ComFIRM Main Source File (All the fun happens here)
 * Author: Fábio Olivé Leite <olive@inf.ufrgs.br>
 * Advisor: Taisy Silva Weber <taisy@inf.ufrgs.br>
 *
 * Copyright 1999, 2000 by Fábio Olivé Leite.
```

```
 * This code is licensed under the GNU General Public License V. 2.
 * Consult http://www.gnu.org/ for details.
 *
 * Version 0.5 - Remember to increase this when updating,
 *              and also check the "log version" command
 */

#include <linux/skbuff.h>
#include <linux/proc_fs.h>
#include <linux/comfirm.h>
#include <asm/uaccess.h>
#include <linux/sched.h>
#include <linux/wrapper.h>
#include <linux/timer.h>
#include <asm/uaccess.h>

/* Internal control variables and structures */
int ComFIRM_Active;
static int ComFIRM_Log_Flag;
static struct wait_queue *ComFIRM_Log_Open_WaitQ = NULL;
static struct wait_queue *ComFIRM_Log_Read_WaitQ = NULL;
struct ComFIRM_Delay_Parameters {
  struct sk_buff *skb;
  struct timer_list *timer;
};
/* The following struct is Very Ugly (TM). */
/* Coding a more elegant/efficient log system is a post-dissertation thing */
static struct ComFIRM_Log_Line {
  char line[100];
  struct ComFIRM_Log_Line *next;
} *ComFIRM_Log_First, *ComFIRM_Log_Last;

/* Rule storage stuff */
int ComFIRM_TX_Rules_Size;
int ComFIRM_RX_Rules_Size;
static unsigned char *ComFIRM_TX_Rules;
static unsigned char *ComFIRM_RX_Rules;

/* External selection resources */
static int ComFIRM_Counters[ComFIRM_Max_Counters]; /* Counters are no longer a special struct */
struct ComFIRM_Timer ComFIRM_Timers[ComFIRM_Max_Timers];
static int ComFIRM_Flags;

/* Some needed forward declarations */
static struct proc_dir_entry proc_net_comfirm_tx_rules;
static struct proc_dir_entry proc_net_comfirm_rx_rules;

/* Functions */
static void ComFIRM_Reset(void) {
  int i;

  /* Deactivate */
  ComFIRM_Active = 0;

  /* Erase rules */
  proc_net_comfirm_tx_rules.size = ComFIRM_TX_Rules_Size = 0;
  if (ComFIRM_TX_Rules != NULL) { kfree(ComFIRM_TX_Rules); ComFIRM_TX_Rules = NULL; }
  proc_net_comfirm_rx_rules.size = ComFIRM_RX_Rules_Size = 0;
  if (ComFIRM_RX_Rules != NULL) { kfree(ComFIRM_RX_Rules); ComFIRM_RX_Rules = NULL; }

  /* Keep only logfile open status */
  ComFIRM_Log_Flag &= ComFIRM_Log_Open;
  /* Maybe eliminate pending log lines */

  /* Reinitialize external selection resources */
  for (i = 0; i < ComFIRM_Max_Counters; i++)
    ComFIRM_Counters[i] = 0;
  for (i = 0; i < ComFIRM_Max_Timers; i++) {
    ComFIRM_Timers[i].time = 0;
    ComFIRM_Timers[i].mode = ComFIRM_Timer_Mode_None;
  }
  ComFIRM_Flags = 0;
}
```

```
/* This has got to be the weirdest log system in the world */
static void ComFIRM_Log(char *str) {
  struct ComFIRM_Log_Line *logline;

  if (ComFIRM_Log_Flag & ComFIRM_Log_Open) {
    logline = kmalloc(sizeof(struct ComFIRM_Log_Line), GFP_KERNEL);
    if (logline == NULL) {
      printk(KERN_NOTICE "ComFIRM: missed log line due to malloc problems!\n");
      return;
    }
    sprintf(logline->line, "%08lx: %s\n", jiffies, str);
    logline->next = NULL;
    if (ComFIRM_Log_First == NULL) {
      ComFIRM_Log_First = ComFIRM_Log_Last = logline;
      /* wake up read queue */
      wake_up(&ComFIRM_Log_Read_WaitQ);
    } else {
      ComFIRM_Log_Last->next = logline;
      ComFIRM_Log_Last = logline;
    }
  }
}

static unsigned char ComFIRM_Apply_Rule(struct sk_buff *skb, unsigned char rulesize,
                                        unsigned char *rulepointer, int *param){
  unsigned char inst, mode;
  int op1, op2; /* Instruction operands */
  int *intp;
  short *shortp;
  char logline[80]; /* This sucks. Think of a better way to do it. */

  while (rulesize > 0) {
    inst = *rulepointer & ComFIRM_Inst_Type_Mask;
    mode = *rulepointer & ComFIRM_Inst_Mode_Mask;
    switch (inst) {
    case ComFIRM_Test_Bit: {
      /* Mode says which test, only param is a short telling which bit to test */
      op1 = *((short *)(rulepointer + 1)) >> 3;
      op2 = *((short *)(rulepointer + 1)) & 7;
      switch (mode) {
      case ComFIRM_Test_Bit0: {
        if (skb->data[op1] & (1 << op2))
          return 0;
        break;
      }
      case ComFIRM_Test_Bit1: {
        if (!(skb->data[op1] & (1 << op2)))
          return 0;
        break;
      }
      }
      rulepointer += 3;
      rulesize -=3;
      break;
    }
    case ComFIRM_Test_Byte: {
      /* Mode says which test, params are a short telling which byte to test and a byte value */
      op1 = *((short *)(rulepointer + 1));
      op2 = *(rulepointer + 3);
      switch (mode) {
      case ComFIRM_Test_Equal: {
        if (skb->data[op1] != op2)
          return 0;
        break;
      }
      case ComFIRM_Test_Greater: {
        if (skb->data[op1] <= op2)
          return 0;
        break;
      }
      case ComFIRM_Test_Less: {
        if (skb->data[op1] >= op2)
```

```
      return 0;
    break;
  }
  }
  rulepointer += 4;
  rulesize -= 4;
  break;
}
case ComFIRM_Test_Word: {
  /* Mode says which test, params are a short telling which word to test and */
  /* a word value in network byte order */
  op1 = *((short *)(rulepointer + 1));
  op2 = *((short *)(rulepointer + 3));
  switch (mode) {
  case ComFIRM_Test_Equal: {
    if (*((short *)(skb->data + op1)) != op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Greater: {
    if (*((short *)(skb->data + op1)) <= op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Less: {
    if (*((short *)(skb->data + op1)) >= op2)
      return 0;
    break;
  }
  }
  rulepointer += 5;
  rulesize -= 5;
  break;
}
case ComFIRM_Test_Double: {
  /* Mode says which test, params are a short telling which double to test and */
  /* a double value in network byte order */
  op1 = *((short *)(rulepointer + 1));
  op2 = *((int *)(rulepointer + 3));
  switch (mode) {
  case ComFIRM_Test_Equal: {
    if (*((int *)(skb->data + op1)) != op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Greater: {
    if (*((int *)(skb->data + op1)) <= op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Less: {
    if (*((int *)(skb->data + op1)) >= op2)
      return 0;
    break;
  }
  }
  rulepointer += 7;
  rulesize -= 7;
  break;
}
case ComFIRM_Test_Counter: {
  /* Mode says which test, params are a byte telling which counter to test and a double value */
  op1 = *(rulepointer + 1);
  op2 = *((int *)(rulepointer + 2));
  switch (mode) {
  case ComFIRM_Test_Equal: {
    if (ComFIRM_Counters[op1] != op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Greater: {
    if (ComFIRM_Counters[op1] <= op2)
      return 0;
```

```
    break;
  }
  case ComFIRM_Test_Less: {
    if (ComFIRM_Counters[op1] >= op2)
      return 0;
    break;
  }
  }
  rulepointer += 6;
  rulesize -= 6;
  break;
}
case ComFIRM_Test_RandByte: { /* Mode says which test, only param is a byte value */
  op1 = *(rulepointer + 1);
  op2 = 0; get_random_bytes(&op2, 1); /* Fetch a random byte from the random number generator */
  switch (mode) {
  case ComFIRM_Test_Equal:
    if (op2 != op1)
      return 0;
    break;
  case ComFIRM_Test_Greater:
    if (op2 <= op1)
      return 0;
    break;
  case ComFIRM_Test_Less:
    if (op2 >= op1)
      return 0;
    break;
  }
  rulepointer += 2;
  rulesize -= 2;
  break;
}
case ComFIRM_Test_Timer: { /* Looks incredibly like a counter test... */
  op1 = *(rulepointer + 1);
  op2 = *((int *)(rulepointer + 2));
  switch (mode) {
  case ComFIRM_Test_Equal: {
    if (ComFIRM_Timers[op1].time != op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Greater: {
    if (ComFIRM_Timers[op1].time <= op2)
      return 0;
    break;
  }
  case ComFIRM_Test_Less: {
    if (ComFIRM_Timers[op1].time >= op2)
      return 0;
    break;
  }
  }
  rulepointer += 6;
  rulesize -= 6;
  break;
}
case ComFIRM_Test_Flag: { /* Mode blah blah, only param is a byte saying which flag */
  op1 = *(rulepointer + 1);
  switch (mode) {
  case ComFIRM_Test_Bit0: {
    if (ComFIRM_Flags & (1 << op1))
      return 0;
    break;
  }
  case ComFIRM_Test_Bit1: {
    if (!(ComFIRM_Flags & (1 << op1)))
      return 0;
    break;
  }
  }
  rulepointer += 2;
  rulesize -= 2;
```

```
  break;
}
case ComFIRM_Action_Bit: { /* Mode tells what to do, param is a short saying which bit */
  op1 = *((short *)(rulepointer + 1)) >> 3;
  op2 = *((short *)(rulepointer + 1)) & 7;
  switch (mode) {
  case ComFIRM_Action_Bit0: {
    skb->data[op1] &= ~(1 << op2);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "bit %d of byte at offset %d set to 0", op1, op2);
      ComFIRM_Log(logline);
    }
    break;
  }
  case ComFIRM_Action_Bit1: {
    skb->data[op1] |= (1 << op2);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "bit %d of byte at offset %d set to 1", op1, op2);
      ComFIRM_Log(logline);
    }
    break;
  }
  case ComFIRM_Action_BitC: {
    skb->data[op1] ^= (1 << op2);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "bit %d of byte at offset %d flipped", op1, op2);
      ComFIRM_Log(logline);
    }
    break;
  }
  }
  rulepointer += 3;
  rulesize -=3;
  break;
}
case ComFIRM_Action_Byte: {
  /* Params are a short telling which byte and an optional byte value (for Action_Set) */
  op1 = *((short *)(rulepointer + 1));
  switch (mode) {
  case ComFIRM_Action_Set: {
    skb->data[op1] = *(rulepointer + 3);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "byte at offset %d set to %d", op1, *(rulepointer + 3));
      ComFIRM_Log(logline);
    }
    rulepointer += 4;
    rulesize -= 4;
    break;
  }
  case ComFIRM_Action_Inc: {
    skb->data[op1]++;
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "byte at offset %d incremented", op1);
      ComFIRM_Log(logline);
    }
    rulepointer += 3;
    rulesize -= 3;
    break;
  }
  case ComFIRM_Action_Dec: {
    skb->data[op1]--;
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "byte at offset %d decremented", op1);
      ComFIRM_Log(logline);
    }
    rulepointer += 3;
    rulesize -= 3;
    break;
  }
  }
  break;
}
case ComFIRM_Action_Word: {
```

```
    /* Params are a short telling which word and an optional word value in network byte order */
    op1 = *((short *)(rulepointer + 1));
    switch (mode) {
    case ComFIRM_Action_Set: {
      *((short *)(skb->data + op1)) = *((short *)(rulepointer + 3));
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "word at offset %d set to %d", op1, *((short *)(rulepointer + 3)));
        ComFIRM_Log(logline);
      }
      rulepointer += 5;
      rulesize -= 5;
      break;
    }
    case ComFIRM_Action_Inc: {
      shortp = (short *)(skb->data + op1);
      (*shortp)++;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "word at offset %d incremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 3;
      rulesize -= 3;
      break;
    }
    case ComFIRM_Action_Dec: {
      shortp = (short *)(skb->data + op1);
      (*shortp)--;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "word at offset %d decremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 3;
      rulesize -= 3;
      break;
    }
    }
    break;
  }
  case ComFIRM_Action_Double: { /* I think you can get it yourself. Remeber the network byte order */
    op1 = *((short *)(rulepointer + 1));
    switch (mode) {
    case ComFIRM_Action_Set: {
      *((int *)(skb->data + op1)) = *((int *)(rulepointer + 3));
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "doubleword at offset %d set to %d", op1, *((int *)(rulepointer + 3)));
        ComFIRM_Log(logline);
      }
      rulepointer += 7;
      rulesize -= 7;
      break;
    }
    case ComFIRM_Action_Inc: {
      intp = (int *)(skb->data + op1);
      (*intp)++;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "doubleword at offset %d incremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 3;
      rulesize -= 3;
      break;
    }
    case ComFIRM_Action_Dec: {
      intp = (int *)(skb->data + op1);
      (*intp)--;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "doubleword at offset %d decremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 3;
      rulesize -= 3;
      break;
    }
```

```
    }
    break;
  }
  case ComFIRM_Action_Drop: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action)
      ComFIRM_Log("message dropped");
    return inst;
  }
  case ComFIRM_Action_Dup: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action)
      ComFIRM_Log("message duplicated");
    return inst;
  }
  case ComFIRM_Action_Delay: {
    *param = *((int *)(rulepointer + 1)); /* Delay specified in 100ths of a second (i386 jiffy) */
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "message delayed by %d ms", (*param) * 10);
      ComFIRM_Log(logline);
    }
    return inst;
  }
  case ComFIRM_Action_Counter: { /* Looks incredibly like a double action, except in host byte order */
    op1 = *(rulepointer + 1);
    switch (mode) {
    case ComFIRM_Action_Set: {
      ComFIRM_Counters[op1] = *((int *)(rulepointer + 2));
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "counter %d set to %d", op1, ComFIRM_Counters[op1]);
        ComFIRM_Log(logline);
      }
      rulepointer += 6;
      rulesize -= 6;
      break;
    }
    case ComFIRM_Action_Inc: {
      ComFIRM_Counters[op1]++;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "counter %d incremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 2;
      rulesize -= 2;
      break;
    }
    case ComFIRM_Action_Dec: {
      ComFIRM_Counters[op1]--;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "counter %d decremented", op1);
        ComFIRM_Log(logline);
      }
      rulepointer += 2;
      rulesize -= 2;
      break;
    }
    }
    break;
  }
  case ComFIRM_Action_Timer: { /* Looks like a counter action with more options */
    op1 = *(rulepointer + 1);
    switch (mode) {
    case ComFIRM_Action_Set: {
      ComFIRM_Timers[op1].time = *((int *)(rulepointer + 2));
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
        sprintf(logline, "timer %d set to %d", op1, ComFIRM_Timers[op1].time);
        ComFIRM_Log(logline);
      }
      rulepointer += 6;
      rulesize -= 6;
      break;
    }
    case ComFIRM_Action_Timer_Mode_None: {
      ComFIRM_Timers[op1].mode = ComFIRM_Timer_Mode_None;
      if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
```

```
      sprintf(logline, "timer %d's mode set to %d", op1, ComFIRM_Timers[op1].mode);
      ComFIRM_Log(logline);
    }
    rulepointer += 2;
    rulesize -= 2;
    break;
  }
  case ComFIRM_Action_Timer_Mode_Inc: {
    ComFIRM_Timers[op1].mode = ComFIRM_Timer_Mode_Inc;
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "timer %d's mode set to %d", op1, ComFIRM_Timers[op1].mode);
      ComFIRM_Log(logline);
    }
    rulepointer += 2;
    rulesize -= 2;
    break;
  }
  case ComFIRM_Action_Timer_Mode_Dec: {
    ComFIRM_Timers[op1].mode = ComFIRM_Timer_Mode_Dec;
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "timer %d's mode set to %d", op1, ComFIRM_Timers[op1].mode);
      ComFIRM_Log(logline);
    }
    rulepointer += 2;
    rulesize -= 2;
    break;
  }
  }
  break;
}
case ComFIRM_Action_Flag: { /* Param is a byte telling which flag to act on */
  op1 = *(rulepointer + 1);
  switch (mode) {
  case ComFIRM_Action_Bit0: {
    ComFIRM_Flags &= ~(1 << op1);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "flag %d set to 0", op1);
      ComFIRM_Log(logline);
    }
    break;
  }
  case ComFIRM_Action_Bit1: {
    ComFIRM_Flags |= (1 << op1);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "flag %d set to 1", op1);
      ComFIRM_Log(logline);
    }
    break;
  }
  case ComFIRM_Action_BitC: {
    ComFIRM_Flags ^= (1 << op1);
    if (ComFIRM_Log_Flag & ComFIRM_Log_Action) {
      sprintf(logline, "flag %d flipped", op1);
      ComFIRM_Log(logline);
    }
    break;
  }
  }
  rulepointer += 2;
  rulesize -= 2;
  break;
}
case ComFIRM_Action_Dump: { /* No parameters, dumps the packet's contents in hex via ComFIRM_Log */
  /* Log packet size */
  sprintf(logline, "dumping packet, %d bytes follow", skb->len);
  ComFIRM_Log(logline);
  /* Log packet contents in hex, 16 bytes at a time */
  op1 = skb->len;
  op2 = 0;
  while (op1 > 0) {
    sprintf(logline, "%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x",
            skb->data[op2], skb->data[op2+1], skb->data[op2+2], skb->data[op2+3],
            skb->data[op2+4], skb->data[op2+5], skb->data[op2+6], skb->data[op2+7],
```

```
                skb->data[op2+8], skb->data[op2+9], skb->data[op2+10], skb->data[op2+11],
                skb->data[op2+12], skb->data[op2+13], skb->data[op2+14], skb->data[op2+15]);
        if (op1 < 16)
          logline[(op1 * 3) - 1] = 0; /* Last line may be shorter than 16 bytes */
        ComFIRM_Log(logline);
        op1 -= 16;
        op2 += 16;
      }
      rulepointer++;
      rulesize--;
      break;
    }
    }
  }
  return 0;
}

void ComFIRM_Delayed_TX(struct ComFIRM_Delay_Parameters *dp) {
  if (ComFIRM_Log_Flag & ComFIRM_Log_Action)
    ComFIRM_Log("sending delayed message");
  old_dev_queue_xmit(dp->skb);
  kfree(dp->timer);
  kfree(dp);
}

int ComFIRM_Apply_TX_Rules(struct sk_buff *skb) {
  unsigned char rulesize, *rulepointer, result;
  struct sk_buff *skb2;
  struct timer_list *tl;
  struct ComFIRM_Delay_Parameters *dp;
  int d;

  rulesize = ComFIRM_TX_Rules[0];
  rulepointer = ComFIRM_TX_Rules + 1;
  while (rulesize > 0) {
    result = ComFIRM_Apply_Rule(skb, rulesize, rulepointer, &d);
    switch (result) {
    case ComFIRM_Action_Drop: {
      kfree_skb(skb);
      return 0;
    }
    case ComFIRM_Action_Delay: {
      dp = kmalloc(sizeof(struct ComFIRM_Delay_Parameters), GFP_KERNEL);
      tl = kmalloc(sizeof(struct timer_list), GFP_KERNEL);
      dp->timer = tl;
      dp->skb = skb;
      tl->prev = tl->next = NULL;
      tl->data = (unsigned long)dp;
      tl->function = (void *)ComFIRM_Delayed_TX;
      tl->expires = jiffies + d;
      add_timer(tl);
      return 0;
    }
    case ComFIRM_Action_Dup: {
      skb2 = skb_clone(skb, GFP_ATOMIC);
      old_dev_queue_xmit(skb2);
      old_dev_queue_xmit(skb);
      return 0;
    }
    }
    rulepointer += rulesize + 1;
    rulesize = *(rulepointer - 1);
  }
  return old_dev_queue_xmit(skb);
}

void ComFIRM_Delayed_RX(struct ComFIRM_Delay_Parameters *dp) {
  if (ComFIRM_Log_Flag & ComFIRM_Log_Action)
    ComFIRM_Log("receiving delayed message");
  old_netif_rx(dp->skb);
  kfree(dp->timer);
  kfree(dp);
}
```

```
void ComFIRM_Apply_RX_Rules(struct sk_buff *skb) {
  unsigned char rulesize, *rulepointer, result;
  struct sk_buff *skb2;
  struct timer_list *tl;
  struct ComFIRM_Delay_Parameters *dp;
  int d;

  rulesize = ComFIRM_RX_Rules[0];
  rulepointer = ComFIRM_RX_Rules + 1;
  while (rulesize > 0) {
    result = ComFIRM_Apply_Rule(skb, rulesize, rulepointer, &d);
    switch (result) {
    case ComFIRM_Action_Drop: {
      kfree_skb(skb);
      return;
    }
    case ComFIRM_Action_Delay: {
      dp = kmalloc(sizeof(struct ComFIRM_Delay_Parameters), GFP_KERNEL);
      tl = kmalloc(sizeof(struct timer_list), GFP_KERNEL);
      dp->timer = tl;
      dp->skb = skb;
      tl->prev = tl->next = NULL;
      tl->data = (unsigned long)dp;
      tl->function = (void *)ComFIRM_Delayed_RX;
      tl->expires = jiffies + d;
      add_timer(tl);
      return;
    }
    case ComFIRM_Action_Dup: {
      skb2 = skb_clone(skb, GFP_ATOMIC);
      old_netif_rx(skb2);
      old_netif_rx(skb);
      return;
    }
    }
    rulepointer += rulesize + 1;
    rulesize = *(rulepointer - 1);
  }
  old_netif_rx(skb);
}

static ssize_t comfirm_read (struct file *file, char *buf, size_t len, loff_t *ppos) {
  struct ComFIRM_Log_Line *logline;
  int i, is_sig=0;

  switch (file->f_dentry->d_inode->i_ino) {
  case PROC_NET_COMFIRM_CONTROL: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("reading from ComFIRM_Control! This shouldn't happen!");
    return -EPERM; /* This operation is not permitted */
  }
  case PROC_NET_COMFIRM_TX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("reading from ComFIRM_TX_Rules");
    if (*ppos >= ComFIRM_TX_Rules_Size)
      return 0;
    if (ComFIRM_TX_Rules_Size > 0 && len >= ComFIRM_TX_Rules_Size) {
      copy_to_user(buf, ComFIRM_TX_Rules, ComFIRM_TX_Rules_Size);
      *ppos += ComFIRM_TX_Rules_Size;
      return ComFIRM_TX_Rules_Size;
    }
    break;
  }
  case PROC_NET_COMFIRM_RX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("reading from ComFIRM_RX_Rules");
    if (*ppos >= ComFIRM_RX_Rules_Size)
      return 0;
    if (ComFIRM_RX_Rules_Size > 0 && len >= ComFIRM_RX_Rules_Size) {
      copy_to_user(buf, ComFIRM_RX_Rules, ComFIRM_RX_Rules_Size);
      *ppos += ComFIRM_RX_Rules_Size;
      return ComFIRM_RX_Rules_Size;
```

```
    }
    break;
  }
  case PROC_NET_COMFIRM_LOG: {
    if ((file->f_flags & O_NONBLOCK) && (ComFIRM_Log_First == NULL))
      return -EAGAIN; /* The process does not want to wait */
    /* Put reading process to sleep if there's no log info at the moment */
    while (ComFIRM_Log_First == NULL) {
      interruptible_sleep_on(&ComFIRM_Log_Read_WaitQ);
      for(i=0; i<_NSIG_WORDS && !is_sig; i++)
        is_sig = current->signal.sig[i] & ~current->blocked.sig[i];
      if (is_sig)
        return -EINTR;
    }
    /* There's something to read */
    if (ComFIRM_Log_First == ComFIRM_Log_Last) {
      /* There's only one line */

      /* copy Log_First to user */
      for (i = 0; i < len && ComFIRM_Log_First->line[i]; i++)
        put_user(ComFIRM_Log_First->line[i], buf + i);

      kfree(ComFIRM_Log_First);
      ComFIRM_Log_First = ComFIRM_Log_Last = NULL;
    } else {
      /* There's more than one line */

      /* copy Log_First to user */
      for (i = 0; i < len && ComFIRM_Log_First->line[i]; i++)
        put_user(ComFIRM_Log_First->line[i], buf + i);

      logline = ComFIRM_Log_First;
      ComFIRM_Log_First = ComFIRM_Log_First->next;
      kfree(logline);
    }
    return i;
    break;
  }
  default:
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("comfirm_read called for an unknown file!");
  }

  return 0;
}

static ssize_t comfirm_write (struct file *file, const char *buf, size_t len, loff_t *ppos) {
  if (*ppos > 0)
    return -ESPIPE; /* Don't allow writes that are not at the beggining of the file */
  switch (file->f_dentry->d_inode->i_ino) {
  case PROC_NET_COMFIRM_CONTROL: {
    if (!strncmp(buf, "ComFIRM", 7)) {
      if (!strncmp(buf + 8, "on", 2)) {
        ComFIRM_Active = 1;
        ComFIRM_Log("fault injection started");
      } else if (!strncmp(buf + 8, "off", 3)) {
        ComFIRM_Active = 0;
        ComFIRM_Log("fault injection stopped");
      } else if (!strncmp(buf + 8, "reset", 5)) {
        ComFIRM_Reset();
        ComFIRM_Log("ComFIRM has been reset");
      } else {
        ComFIRM_Log("unknown parameter to ComFIRM command!");
      }
    } else if (!strncmp(buf, "log", 3)) {
      if (!strncmp(buf + 4, "action", 6)) {
        if (!strncmp(buf + 11, "on", 2)) {
          ComFIRM_Log_Flag |= ComFIRM_Log_Action;
          ComFIRM_Log("logging of action events is on");
        } else if (!strncmp(buf + 11, "off", 3)) {
          ComFIRM_Log_Flag &= ~ComFIRM_Log_Action;
          ComFIRM_Log("logging of action events is off");
        } else {
```

```
        ComFIRM_Log("unknown parameter to log action command!");
      }
      /* Logging of selection events was found to be useless, so it's disabled. */
      /*      } else if (!strncmp(buf + 4, "selection", 9)) { */
      /*        if (!strncmp(buf + 14, "on", 2)) { */
      /*          ComFIRM_Log_Flag |= ComFIRM_Log_Selection; */
      /*          ComFIRM_Log("logging of selection events is on"); */
      /*        } else if (!strncmp(buf + 14, "off", 3)) { */
      /*          ComFIRM_Log_Flag &= ~ComFIRM_Log_Selection; */
      /*          ComFIRM_Log("logging of selection events is off"); */
      /*        } else { */
      /*          ComFIRM_Log("unknown parameter to log selection command!"); */
      /*        } */
    } else if (!strncmp(buf + 4, "debug", 5)) {
      if (!strncmp(buf + 10, "on", 2)) {
        ComFIRM_Log_Flag |= ComFIRM_Log_Debug;
        ComFIRM_Log("debugging messages are on");
      } else if (!strncmp(buf + 10, "off", 3)) {
        ComFIRM_Log_Flag &= ~ComFIRM_Log_Debug;
        ComFIRM_Log("debugging messages are off");
      } else {
        ComFIRM_Log("unknown parameter to log debug command!");
      }
    } else if (!strncmp(buf + 4, "version", 7)) {
      ComFIRM_Log("ComFIRM version 0.5");
    } else {
      ComFIRM_Log("unknown parameter to log command!");
    }
  } else if (!strncmp(buf, "erase", 5)) {
    if (!strncmp(buf + 6, "tx", 2)) {
      proc_net_comfirm_tx_rules.size = ComFIRM_TX_Rules_Size = 0;
      if (ComFIRM_TX_Rules != NULL) { kfree(ComFIRM_TX_Rules); ComFIRM_TX_Rules = NULL; }
      ComFIRM_Log("tx rules have been erased");
    } else if (!strncmp(buf + 6, "rx", 2)) {
      proc_net_comfirm_rx_rules.size = ComFIRM_RX_Rules_Size = 0;
      if (ComFIRM_RX_Rules != NULL) { kfree(ComFIRM_RX_Rules); ComFIRM_RX_Rules = NULL; }
      ComFIRM_Log("rx rules have been erased");
    } else {
      ComFIRM_Log("unknown parameter to erase command!");
    }
  } else {
    ComFIRM_Log("unknown command written to ComFIRM_Control!");
  }
  break;
}
case PROC_NET_COMFIRM_TX_RULES: {
  if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
    ComFIRM_Log("writing to ComFIRM_TX_Rules");
  if (ComFIRM_TX_Rules != NULL) {
    kfree(ComFIRM_TX_Rules); /* Get rid of old rules */
    ComFIRM_TX_Rules = NULL;
  }
  if (len > 0) {
    ComFIRM_TX_Rules = kmalloc(len, GFP_KERNEL);
    copy_from_user(ComFIRM_TX_Rules, buf, len);
  }
  proc_net_comfirm_tx_rules.size = ComFIRM_TX_Rules_Size = *ppos = len;
  break;
}
case PROC_NET_COMFIRM_RX_RULES: {
  if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
    ComFIRM_Log("writing to ComFIRM_RX_Rules");
  if (ComFIRM_RX_Rules != NULL) {
    kfree(ComFIRM_RX_Rules); /* Get rid of old rules */
    ComFIRM_RX_Rules = NULL;
  }
  if (len > 0) {
    ComFIRM_RX_Rules = kmalloc(len, GFP_KERNEL);
    copy_from_user(ComFIRM_RX_Rules, buf, len);
  }
  proc_net_comfirm_rx_rules.size = ComFIRM_RX_Rules_Size = *ppos = len;
  break;
}
```

```
  case PROC_NET_COMFIRM_LOG: {
    ComFIRM_Log("writing to ComFIRM_Log! This shouldn't happen!");
    return -EPERM; /* This operation is not permitted */
  }
  default:
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("comfirm_write called for an unknown file!");
  }

  return len; /* At least make the caller believe everything is fine */
}

static int comfirm_open (struct inode *ino, struct file *filep) {
  int i, is_sig=0;

  switch (ino->i_ino) {
  case PROC_NET_COMFIRM_CONTROL: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("opening ComFIRM_Control");
    break;
  }
  case PROC_NET_COMFIRM_TX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("opening ComFIRM_TX_Rules");
    break;
  }
  case PROC_NET_COMFIRM_RX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("opening ComFIRM_RX_Rules");
    break;
  }
  case PROC_NET_COMFIRM_LOG: {
    if ((filep->f_flags & O_NONBLOCK) && (ComFIRM_Log_Flag & ComFIRM_Log_Open))
      return -EAGAIN; /* The process does not want to wait */
    while (ComFIRM_Log_Flag & ComFIRM_Log_Open) { /* Someone is already listening for log data */
      interruptible_sleep_on(&ComFIRM_Log_Open_WaitQ);
      for(i=0; i<_NSIG_WORDS && !is_sig; i++)
        is_sig = current->signal.sig[i] & ~current->blocked.sig[i];
      if (is_sig)
        return -EINTR;
    }
    ComFIRM_Log_Flag |= ComFIRM_Log_Open; /* Allow logging only when there's someone listening */
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("opening ComFIRM_Log");
    break;
  }
  default:
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("comfirm_open called for an unknown file!");
  }

  return 0;
}

static int comfirm_close (struct inode *ino, struct file *file) {
  struct ComFIRM_Log_Line *logline;

  switch (ino->i_ino) {
  case PROC_NET_COMFIRM_CONTROL: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("closing ComFIRM_Control");
    break;
  }
  case PROC_NET_COMFIRM_TX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("closing ComFIRM_TX_Rules");
    break;
  }
  case PROC_NET_COMFIRM_RX_RULES: {
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("closing ComFIRM_RX_Rules");
    break;
  }
```

```
  case PROC_NET_COMFIRM_LOG: {
    ComFIRM_Log_Flag &= ~ComFIRM_Log_Open; /* Stop logging because there's no one listening */
    /* Flush remaining log lines */
    ComFIRM_Log_Last = NULL;
    while (ComFIRM_Log_First != NULL) {
      logline = ComFIRM_Log_First;
      ComFIRM_Log_First = ComFIRM_Log_First->next;
      kfree(logline);
    }
    wake_up(&ComFIRM_Log_Open_WaitQ);
    break;
  }
  default:
    if (ComFIRM_Log_Flag & ComFIRM_Log_Debug)
      ComFIRM_Log("comfirm_close called for an unknown file!");
  }

  return 0;
}

static struct file_operations comfirm_fops =
{
    NULL,         /*  Seek              */
    comfirm_read, /*  Read              */
    comfirm_write,/*  Write             */
    NULL,         /*  Readdir           */
    NULL,         /*  Poll              */
    NULL,         /*  IOctl             */
    NULL,         /*  MMAP              */
    comfirm_open, /*  Open              */
    NULL,         /*  Flush             */
    comfirm_close,/*  Release           */
    NULL,         /*  Fsync             */
    NULL,         /*  Fasync            */
    NULL,         /*  CheckMediaChange  */
    NULL,         /*  Revalidate        */
    NULL,         /*  Lock              */
};

static struct inode_operations proc_net_comfirm_inode_operations = {
        &comfirm_fops,            /* default property file-ops */
        NULL,                       /* create */
        NULL,                       /* lookup */
        NULL,                       /* link */
        NULL,                       /* unlink */
        NULL,                       /* symlink */
        NULL,                       /* mkdir */
        NULL,                       /* rmdir */
        NULL,                       /* mknod */
        NULL,                       /* rename */
        NULL,                       /* readlink */
        NULL,                       /* follow_link */
        NULL,                       /* readpage */
        NULL,                       /* writepage */
        NULL,                       /* bmap */
        NULL,                       /* truncate */
        NULL                       /* permission */
};

static struct proc_dir_entry proc_net_comfirm_control = {
        PROC_NET_COMFIRM_CONTROL, 15, "ComFIRM_Control",
        S_IFREG | S_IWUSR, 1, 0, 0,
        0, &proc_net_comfirm_inode_operations
};

static struct proc_dir_entry proc_net_comfirm_tx_rules = {
        PROC_NET_COMFIRM_TX_RULES, 16, "ComFIRM_TX_Rules",
        S_IFREG | S_IWUSR | S_IRUSR, 1, 0, 0,
        0, &proc_net_comfirm_inode_operations
};

static struct proc_dir_entry proc_net_comfirm_rx_rules = {
        PROC_NET_COMFIRM_RX_RULES, 16, "ComFIRM_RX_Rules",
```

```
        S_IFREG | S_IWUSR | S_IRUSR, 1, 0, 0,
        0, &proc_net_comfirm_inode_operations
};

static struct proc_dir_entry proc_net_comfirm_log = {
        PROC_NET_COMFIRM_LOG, 11, "ComFIRM_Log",
        S_IFREG | S_IRUSR, 1, 0, 0,
        0, &proc_net_comfirm_inode_operations
};

void ComFIRM_Init(void) {
  int i;

  printk(KERN_INFO "ComFIRM v0.5 by Fabio Olive Leite (olive@inf.ufrgs.br)\n");
  /* Initial state is inactive */
  ComFIRM_Active = 0;

  /* Initialize rule storage resources */
  proc_net_comfirm_tx_rules.size = ComFIRM_TX_Rules_Size = 0;
  ComFIRM_TX_Rules = NULL;
  proc_net_comfirm_rx_rules.size = ComFIRM_RX_Rules_Size = 0;
  ComFIRM_RX_Rules = NULL;

  /* Initial log flags */
  ComFIRM_Log_Flag = 0;
  ComFIRM_Log_First = NULL;
  ComFIRM_Log_Last = NULL;

  /* Initialize external selection resources */
  for (i = 0; i < ComFIRM_Max_Counters; i++)
    ComFIRM_Counters[i] = 0;
  for (i = 0; i < ComFIRM_Max_Timers; i++) {
    ComFIRM_Timers[i].time = 0;
    ComFIRM_Timers[i].mode = ComFIRM_Timer_Mode_None;
  }
  ComFIRM_Flags = 0;

  /* Register files on /proc/net */
  proc_net_register(&proc_net_comfirm_control);
  proc_net_register(&proc_net_comfirm_tx_rules);
  proc_net_register(&proc_net_comfirm_rx_rules);
  proc_net_register(&proc_net_comfirm_log);
}
```

Anexo 2 Código da interface ComFIRM's Face

Abaixo está o código em linguagem Perl da interface gráfica simplificada para a ferramenta ComFIRM, chamada ComFIRM's Face. Seu principal propósito é demonstrar a possibilidade de se utilizar de forma amigável a ferramenta ComFIRM, através de uma interface que abstraia a manipulação dos arquivos virtuais e a geração de *bytecodes* para as regras. Esta interface serve apenas como exemplo, pois, embora suas opções sejam suficientes para experimentos simples, não explora em sua totalidade os recursos oferecidos pela ComFIRM.

```
#!/usr/bin/perl -w

# ComFIRM's Face, a GUI for ComFIRM
# Author: Fábio Olivé Leite <olive@inf.ufrgs.br>
# Advisor: Taisy Silva Weber <taisy@inf.ufrgs.br>
#
# Copyright 1999, 2000 by Fábio Olivé Leite
# This code is licensed under the GNU General Public License V. 2.
# Consult http://www.gnu.org/ for details.

use strict;
use Tk;
use FileHandle;

# Main window

my $top = MainWindow->new();
$top->title("ComFIRM's Face - Version 0.1");

# Globals

my $Log_Debug_Messages = 0;
my $Log_Action_Events = 0;
my $Protocol = "any";
my $OldProtocol = $Protocol;
my $PortOrType = 0;
my $StreamType = "all packets";
my $OldStreamType = $StreamType;
my $StreamN = 0;
my $Action = "none";
my $OldAction = $Action;
my $Dump = 0;
my $Delay = 0;
my $Direction = "tx";

open CONTROL, ">/proc/net/ComFIRM_Control" or die "Can't open ComFIRM_Control: $!\n";
CONTROL->autoflush(1);

# The log is a bit more complex...

my $LOG = new FileHandle "/proc/net/ComFIRM_Log", O_RDONLY | O_NONBLOCK;
if (!defined $LOG) {
    die "Can't open ComFIRM_Log: $!\n";
}

# Menus

my $menu_bar = $top->Frame(relief => "raised",
                                  borderwidth => 1);
$menu_bar->pack(side => "top", anchor => "w", fill => "x");

my $ComFIRM_mb = $menu_bar->Menubutton(text => "ComFIRM",
                                                relief => "groove",
                                                underline => 0);
$ComFIRM_mb->pack(side => "left", padx => 4, pady => 4);
$ComFIRM_mb->command(-label => "Start",
                            underline => 0,
                            command => sub {print CONTROL "ComFIRM on\n"});
```

```
$ComFIRM_mb->command(-label => "Stop",
                            underline => 3,
                            command => sub {print CONTROL "ComFIRM off\n"});
$ComFIRM_mb->command(-label => "Reset",
                            underline => 1,
                            command => sub {
                                print CONTROL "ComFIRM reset\n";
                                $Log_Debug_Messages = 0;
                                $Log_Action_Events = 0;
                            });
$ComFIRM_mb->separator();
$ComFIRM_mb->command(-label => "Erase TX",
                            command => sub {print CONTROL "erase tx\n"},
                            underline => 6);
$ComFIRM_mb->command(-label => "Erase RX",
                            command => sub {print CONTROL "erase rx\n"},
                            underline => 6);
$ComFIRM_mb->separator();
$ComFIRM_mb->command(-label => "Exit ComFIRM's Face",
                               underline => 1,
                               command => [$top => 'destroy']);

my $Log_mb = $menu_bar->Menubutton(text => "Log",
                                        relief => "groove",
                                        underline => 0);
$Log_mb->pack(side => "left", padx => 4, pady => 4);
$Log_mb->command(-label => "Clear log messages",
                        underline => 0,
                        command => \&clear_the_log);
$Log_mb->separator();
$Log_mb->checkbutton(-label => "Log Action Events",
                            underline => 4,
                            variable => \$Log_Action_Events,
                            command => sub {
                                if ($Log_Action_Events == 1) {
                                    print CONTROL "log action on\n";
                                } else {
                                    print CONTROL "log action off\n";
                                }
                            });
$Log_mb->checkbutton(-label => "Log Debug Messages",
                            underline => 4,
                            variable => \$Log_Debug_Messages,
                            command => sub {
                                if ($Log_Debug_Messages == 1) {
                                    print CONTROL "log debug on\n";
                                } else {
                                    print CONTROL "log debug off\n";
                                }
                            });
$Log_mb->separator();
$Log_mb->command(-label => "Log ComFIRM Version",
                        underline => 12,
                        command => sub {print CONTROL "log version\n"});

my $Apply_button = $menu_bar->Button(text => "Apply changes",
                                          relief => "groove",
                                          underline => 0,
                                          command => \&apply_rules);
$top->bind('<Alt-a>', \&apply_rules);
$top->bind('<Alt-A>', \&apply_rules);
$Apply_button->pack(side => "left",
                         padx => 4,
                         pady => 4);

my $Help_mb = $menu_bar->Menubutton(text => "Help",
                                         relief => "groove",
                                         underline => 0);
$Help_mb->pack(side => "right", padx => 4, pady => 4);
$Help_mb->command(-label => "About...",
                         underline => 0,
                         command => \&about_box);
```

```
# Experiment description stuff

my $frame_sel0 = $top->Frame(relief => "raised",
                                borderwidth => 1);
$frame_sel0->pack(side => "top", fill => "x");

$frame_sel0->Label(text => "Stream direction:")->pack(side => "left",
                                                       padx => 4,
                                                       pady => 4,
                                                       anchor => "w");
$frame_sel0->Radiobutton(text => "Reception",
                          variable => \$Direction,
                          value => "rx",
                          command => \&port_change)->pack(side => "right",
                                                           padx => 4,
                                                           pady => 4,
                                                           anchor => "w");
$frame_sel0->Radiobutton(text => "Transmission",
                          variable => \$Direction,
                          value => "tx",
                          command => \&port_change)->pack(side => "right",
                                                           padx => 4,
                                                           pady => 4,
                                                           anchor => "w");

my $frame_sel1 = $top->Frame(relief => "raised",
                                borderwidth => 1);
$frame_sel1->pack(side => "top", fill => "x");

$frame_sel1->Label(text => "Content-based Selection:")->pack(side => "left",
                                                              padx => 4,
                                                              pady => 4,
                                                              anchor => "w");

my $pe = $frame_sel1->Entry(relief => "sunken",
                             -border => 1,
                             -textvariable => \$PortOrType,
                             -justify => "c",
                             # this is a bit of a hack, using xscrollcommand
                             # to track value changes...
                             -xscrollcommand => \&port_change,
                             -bg => "lightgray",
                             -state => "disabled",
                             -width => 7)->pack(side => "right",
                                                 padx => 4,
                                                 pady => 4,
                                                 anchor => "w");

$frame_sel1->Label(text => "Port/Type:")->pack(side => "right",
                                                padx => 4,
                                                pady => 4,
                                                anchor => "w");

$frame_sel1->Optionmenu(options => [qw(any TCP UDP ICMP)],
                         relief => "groove",
                         -command => \&protocol_change,
                         -variable => \$Protocol)->pack(side => "right",
                                                         padx => 4,
                                                         pady => 4,
                                                         anchor => "w");

$frame_sel1->Label(text => "Protocol:")->pack(side => "right",
                                              padx => 4,
                                              pady => 4,
                                              anchor => "w");

my $frame_sel2 = $top->Frame(relief => "raised",
                                borderwidth => 1);
$frame_sel2->pack(side => "top", fill => "x");

$frame_sel2->Label(text => "Stream-based Selection:")->pack(side => "left",
                                                             padx => 4,
                                                             pady => 4,
```

```
                                                            anchor => "w");

my $ne = $frame_sel2->Entry(relief => "sunken",
                                   -border => 1,
                                   -textvariable => \$StreamN,
                                   -justify => "c",
                                   # this is a bit of a hack, using xscrollcommand
                                   # to track value changes...
                                   -xscrollcommand => \&port_change,
                                   -bg => "lightgray",
                                   -state => "disabled",
                                   -width => 7)->pack(side => "right",
                                                       padx => 4,
                                                       pady => 4,
                                                       anchor => "w");

$frame_sel2->Label(text => "N:")->pack(side => "right",
                                        padx => 4,
                                        pady => 4,
                                        anchor => "w");

$frame_sel2->Optionmenu(options => ["all packets", "every Nth packet",
                                    "the Nth packet", "N% of the packets"],
                                relief => "groove",
                                -command => \&stream_change,
                                -variable => \$StreamType)->pack(side => "right",
                                                                  padx => 4,
                                                                  pady => 4,
                                                                  anchor => "w");

$frame_sel2->Label(text => "Select:")->pack(side => "right",
                                             padx => 4,
                                             pady => 4,
                                             anchor => "w");

my $frame_act = $top->Frame(relief => "raised",
                                   borderwidth => 1);
$frame_act->pack(side => "top", fill => "x");

$frame_act->Label(text => "Action on selected packets:")->pack(side => "left",
                                                                padx => 4,
                                                                pady => 4,
                                                                anchor => "w");

$frame_act->Checkbutton(text => "Dump contents",
                             command => \&port_change,
                             variable => \$Dump)->pack(side => "right",
                                                        padx => 4,
                                                        pady => 4,
                                                        anchor => "w");

$frame_act->Label(text => "ms")->pack(side => "right",
                                       padx => 4,
                                       pady => 4,
                                       anchor => "w");

my $de = $frame_act->Entry(relief => "sunken",
                                  -border => 1,
                                  -textvariable => \$Delay,
                                  -justify => "c",
                                   # this is a bit of a hack, using xscrollcommand
                                   # to track value changes...
                                  -xscrollcommand => \&port_change,
                                  -bg => "lightgray",
                                  -state => "disabled",
                                  -width => 7)->pack(side => "right",
                                                      padx => 4,
                                                      pady => 4,
                                                      anchor => "w");

$frame_act->Label(text => "Delay time:")->pack(side => "right",
                                                padx => 4,
                                                pady => 4,
```

```
                                                    anchor => "w");

$frame_act->Optionmenu(options => ["none", "drop", "duplicate", "delay"],
                               relief => "groove",
                               -command => \&action_change,
                               -variable => \$Action)->pack(side => "right",
                                                                 padx => 4,
                                                                 pady => 4,
                                                                 anchor => "w");

# Log window

my $log_zone = $top->Frame(relief => "raised",
                                  borderwidth => 1);
$log_zone->pack(side => "bottom", anchor => "w", fill => "both", expand => "y");

my $log = $log_zone->Scrolled('Text', relief => "sunken",
                              border => 1,
                              state  => "disabled",
                              bg => "white",
                              width  => 80,
                              height => 7,
                              -scrollbars => "oe");
$log->pack(side => "bottom", padx => 4, pady => 4,
                fill => "both", expand => "y");

$log_zone->Label(text => "Logged information:")->pack(side => "top",
                                                           padx => 4,
                                                           anchor => "w");

# Main loop

my $timer = $top->repeat(100, \&get_the_log);

MainLoop();

# Functions

sub get_the_log() {
    my $line;
    my @lines = <$LOG>;
    if (@lines) {
        $log->configure(state => "normal");
        foreach $line (@lines) {
            $line =~ s/(.{8})/hex $1/e; # Makes that hex timestamp more readable
            $log->insert("end", $line);
        }
        $log->configure(state => "disabled");
        $log->see("end");
    }
}

sub clear_the_log() {
    $log->configure(state => "normal");
    $log->delete("1.0", "end");
    $log->configure(state => "disabled");
    $log->see("end");
}

sub about_box() {
    $top->messageBox(-icon => "info",
                         -type => "OK",
                         -font => "helvetica",
                         -title => "About ComFIRM's Face",
                         -message => "This is ComFIRM's Face Version 0.1, a very simple interface for ComFIRM.
It should be noted that ComFIRM itself is capable of much more than what is shown here.\n\n
By Fabio Olive Leite \(olive\@inf.ufrgs.br\)");
}

sub protocol_change() {
    unless ($Protocol eq $OldProtocol) {
        $Apply_button->configure(state => "normal");
        $OldProtocol = $Protocol;
```

```
        if ($Protocol eq "any") {
            $pe->configure(state => "disabled",
                           bg => "lightgray");
        } else {
            $pe->configure(state => "normal",
                           bg => "white");
        }
    }
}

sub stream_change() {
    unless ($StreamType eq $OldStreamType) {
        $Apply_button->configure(state => "normal");
        $OldStreamType = $StreamType;
        if ($StreamType eq "all packets") {
            $ne->configure(state => "disabled",
                           bg => "lightgray");
        } else {
            $ne->configure(state => "normal",
                           bg => "white");
        }
    }
}

sub action_change() {
    unless ($Action eq $OldAction) {
        $Apply_button->configure(state => "normal");
        $OldAction = $Action;
        if ($Action eq "delay") {
            $de->configure(state => "normal",
                           bg => "white");
        } else {
            $de->configure(state => "disabled",
                           bg => "lightgray");
        }
    }
}

# This function did something else than what it does now, but it became
# useful for easily enabling the apply button.
sub port_change() {
    $Apply_button->configure(state => "normal");
}

# This is where the rules get generated
sub apply_rules() {
    my @bytes; # This will hold the bytes for the rule

    # Content based selection
    unless ($Protocol eq "any") {
        push @bytes, 0x10, 0x0c, 0x00, 0x08, 0x00; # Test for IP
        if ($Protocol eq "TCP") {
            push @bytes, 0x08, 0x17, 0x00, 0x06; # Test for TCP
            push @bytes, 0x10, 0x24, 0x00, unpack "C2", pack "n", $PortOrType; # Test for the given port
        } elsif ($Protocol eq "UDP") {
            push @bytes, 0x08, 0x17, 0x00, 0x11; # Test for UDP
            push @bytes, 0x10, 0x24, 0x00, unpack "C2", pack "n", $PortOrType; # Test for the given port
        } elsif ($Protocol eq "ICMP") {
            push @bytes, 0x08, 0x17, 0x00, 0x01; # Test for ICMP
            push @bytes, 0x08, 0x22, 0x00, $PortOrType; # Test for the given type
            # Who the heck knows by heart all ICMP message types anyway?
        }
    }

    # Stream based selection
    if ($StreamType eq "every Nth packet") {
        # Increment a counter, test it, zero it
        push @bytes, 0x79, 0x00, 0x20, 0x00, unpack "C4", pack "L", $StreamN;
        push @bytes, 0x78, 0x00, 0x00, 0x00, 0x00, 0x00;
    } elsif ($StreamType eq "the Nth packet") {
        # Increment a counter, test it
        push @bytes, 0x79, 0x00, 0x20, 0x00, unpack "C4", pack "L", $StreamN;
    } elsif ($StreamType eq "N% of the packets") {
```

```
        # Get a random byte and test wether it's less than N*255/100
        push @bytes, 0x2a, int $StreamN*255/100;
    }

    # Actions
    if ($Dump) {
        push @bytes, 0x90;
    }
    if ($Action eq "drop") {
        push @bytes, 0x60;
    } elsif ($Action eq "duplicate") {
        push @bytes, 0x70;
    } elsif ($Action eq "delay") {
        push @bytes, 0x68, unpack "C4", pack "L", $Delay/10;
    }

    # Write the stuff
    open RULES, ($Direction eq "tx") ?
    ">/proc/net/ComFIRM_TX_Rules" : ">/proc/net/ComFIRM_RX_Rules" or
    die "Can't open rule file: $!\n";
    print RULES pack "C*", scalar @bytes, @bytes, 0;
    close RULES;
    $Apply_button->configure(state => "disabled");
}
```

Bibliografia

[ARL 93] ARLAT, J. et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. **IEEE Transactions on Computers**, New York, v. 42 n. 8, p. 913–923, Aug. 1993.

[BAR 96] BARCELOS, Patrícia P. **ADC - Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável**. Porto Alegre: CPGCC/UFRGS, 1996.

[BAR 99] BARCELOS, Patrícia P.; LEITE, Fábio O.; WEBER, Taisy S. Implementação de um Injetor de Falhas de Comunicação. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, SCTF, 8., 1999, Campinas. **Anais...** Campinas: Unicamp, 1999.

[BEC 98] BECK, Michael; BÖHME, Harald; DZIADZKA, Mirko; KUNITZ, Ulrich; MAGNUS, Robert; VERWORNER, Dirk. **Linux Kernel Internals**. 2nd ed. Harlow: Addison-Wesley, 1998.

[BEN 96] BENTSON, Randolph. **Inside Linux**: A Look at Operating System Development. Seattle: Specialized Systems Consultants, 1996.

[BOW 98a] BOWMAN, Ivan. **Conceptual Architecture of the Linux Kernel**. Waterloo: Department of Computer Science, University of Waterloo, 1998, Trabalho de Aula. Disponível em: <http://plg.uwaterloo.ca/~itbowman/CS746G/a1/>. Acesso em: 12 maio 1999.

[BOW 98b] BOWMAN, Ivan; SIDDIQI, Saheem; TANUAN, Meyer. **Concrete Architecture of the Linux Kernel**. Waterloo: Department of Computer Science, University of Waterloo, 1998, Trabalho de Aula. Disponível em: <http://plg.uwaterloo.ca/~itbowman/CS746G/a2/>. Acesso em: 14 maio 1999.

[CAR 95a] CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Assessing the Effects of Communication Faults on Parallel Applications. In: INTERNATIONAL COMPUTER AND DEPENDABILITY SYMPOSIUM, 1995, Erlangen, Germany. **Proceedings...** New York:IEEE Computer Society Press, 1995.

[CAR 98a] CARREIRA, J.; MADEIRA, H.; SILVA, J. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. **Transactions on Software Engineering**, New York, v. 24 n. 2, p. 125–136, Feb. 1998.

[CAR 98b] CARD, Rémy; DUMAS, Éric; MÉVEL, Franck. **The Linux Kernel Book**. Chichester: John Wiley & Sons, 1998.

[CHR 86] CHRISTIAN, F. et al. Clock Synchronization in the Presence of Omission and Performance Faults, and Processor Joins. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, 1986, Viena, Austria. **Proceedings...** New York: IEEE Computer Society Press, 1986.

[DAW 95] DAWSON, S.; JAHANIAN, F. Probing and Fault Injection of Protocol Implementations. In: INT. CONF. ON DISTRIBUTED COMPUTER SYSTEMS, 1995, Vancouver, Canada. **Proceedings...** New York:IEEE Computer Society Press, 1995.

[DAW 96a] DAWSON, S.; JAHANIAN, F.; MITTON, T. **Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool**. Ann Arbor: University of Michigan, Department of Electrical Engineering and Computer Science, 1996. (Tecnical Report, n. 298).

[DAW 96b] DAWSON, S.; JAHANIAN, F.; MITTON, T. **ORCHESTRA:** A Fault Injection Environment for Distributed Systems. Ann Arbor: University of Michigan, Department of Electrical Engineering and Computer Science, 1996. (Technical Report, n. 318).

[DAW 98] DAWSON, S. **Message Level Fault Injection in Distributed Systems**. Ann Arbor: University of Michigan, 1998. Tese de Doutorado.

[GON 00] GONÇALVES, Luis C. R.; WEBER, Taisy S. Injeção de Falhas via Depuradores. In: FÓRUM INTERNACIONAL SOFTWARE LIVRE, 1., 2000, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 2000.

[HAD 93] HADZILACOS, Vassos; TOUEG, Sam. Fault-tolerant broadcasts and related problems. In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed. Reading: Addison Wesley, 1993.

[HSU 97] HSUEH, M.; TSAI, T.; IYER, R. Fault Injection Techniques and Tools. **IEEE Computer**, New York, v. 30, n. 4, p. 75–82, Apr. 1997.

[JAL 94] JALOTE, Pankaj. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice-Hall, 1994.

[KAN 95] KANAWATI, G.; KANAWATI, N.; ABRAHAM, J. FERRARI: A Flexible Software-Based Fault and Error Injection System. **IEEE Transactions on Computers**, New York, v. 44, n. 2, p. 248–260, Feb. 1995.

[LEI 99] LEITE, Fábio O. **Injeção de Falhas de Comunicação no Sistema Operacional Linux**: trabalho individual. Porto Alegre: PPGC da UFRGS, 1999.

[PAR 98] PARKER, S.; SCHMECHEL, C. **rfc2398 - Some Testing Tools for TCP Implementors**. FIXME: Network Working Group, The Inter-

net Society, 1998. Request for Comments, 2398 (FYI 33). Disponível em: <http://rfc.net/rfc2398.html>. Acesso em: 9 abr. 1999.

[POM 99] POMERANTZ, Ori. **Linux Kernel Module Programming Guide**. Disponível em: <http://www.linuxdoc.org/LDP/lkmpg/mpg.html>. Acesso em: 08 abr. 1999.

[PRA 96] PRADHAN, Dhiraj. **Fault-Tolerant Computer System Design**. Upper Saddle RIver: Prentice-Hall, 1996.

[ROB 2000] ROBERTSON, Alan. **Linux-HA Heartbeat System Design**. Palestra proferida no Atlanta Linux Showcase, 14 de outubro, 2000. Texto disponível em: <http://www.linux-ha.org/comm>. Acesso em: 08 nov. 2000.

[ROS 98] ROSA, A. A; MARTINS, E. Using reflexive programming to inject faults into object-oriented systems. In: IFIP INT. WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, 1998, Johannesburg, South Africa. **Proceedings...** Johannesburg: University of the Witwatersrand, 1998.

[RUS 97] RUSLING, David. **The Linux Kernel**. Disponível em: <http://www.ssc.com/linux/LDP/LDP/tlk/tlk.html>. Acesso em: 18 maio 1999.

[SOT 96] SOTOMA, Irineu; WEBER, Taisy S. Construção de Ferramentas de Injeção de Falhas em Sistemas Distribuídos. In: SIMPÓSIO REGIONAL DE TOLERÂNCIA A FALHAS, 1., 1996, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1996.

[SOT 97] SOTOMA, Irineu. **AFIDS - Arquitetura para Injeção de Falhas em Sistemas Distribuídos**. Porto Alegre: CPGCC/UFRGS, 1997. Dissertação de Mestrado.

[SRI 97] SRINIVASAN, Sriram. **Advanced Perl Programming**. Sebastopol: O'Reilly, 1997.

[STA 00] STALLMAN, Richard. **The Free Software Movement and the GNU/Linux Operating System**. Palestra proferida no 1º Fórum Gnu/Linux de Curitiba, 30 de abril de 2000.

[TEI 94] TEIXEIRA JR, Carlos A.; WEBER, Taisy S. **FIX - Uma Ferramenta para Recuperação de Processos e Análise de Propagação de Falhas no Sistema UNIX**: trabalho individual. Porto Alegre: CPGCC/UFRGS, 1994.

[WAL 96] WALL, Larry; CHRISTIANSEN, Tom; SCHWARTZ, Randal. **Programming Perl**. 2nd ed. Sebastopol, O'Reilly, 1996.