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

ROBERTO JUNG DREBES

FIRMAMENT: Um Módulo de Injeção de Falhas de Comunicação para Linux

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Profª. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, outubro de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Drebes, Roberto Jung

FIRMAMENT: Um Módulo de Injeção de Falhas de Comunicação para Linux / Roberto Jung Drebes. – Porto Alegre: PPGC da UFRGS, 2005.

87 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientadora: Taisy Silva Weber.

1. Protocolos de comunicação. 2. Sistemas distribuídos. 3. Injeção de falhas. 4. Teste. 5. Validação experimental. I. Weber, Taisy Silva. II. Título.

“I may not have gone where I intended to go,
but I think I have ended up where I needed to be.”
—DOUGLAS ADAMS

AGRADECIMENTOS

Este trabalho não poderia ter sido realizada sem o apoio de diversas pessoas. A elas, por ajudarem de forma técnica ou teórica, dizerem as palavras certas nos momentos certos, ou simplesmente estarem junto a mim quando mais necessário, fica aqui um agradecimento especial:

- À minha orientadora, Profª. Dra. Taisy Silva Weber, agradeço pela "adoção" como bolsista de iniciação científica em 1999, e pelo acompanhamento à minha (ainda curta) carreira científica. Agradeço pela confiança incondicional em todo momento, mesmo quando nem eu depositava tanta confiança em mim e no meu trabalho. Taisy é um exemplo de pesquisadora e professora, mas mais importante, um exemplo de ser humano: sempre aberta a ajudar os amigos, alunos e demais pessoas que a cercam, de qualquer forma;
- À minha colega e amiga Gabriela Jacques-Silva (ou Jaques da Silva, quando fora de sua vida "artística"), parceira de tantas divagações sobre a computação e sobre a vida, pelo companheirismo, pela busca do artigo perfeito escrito a quatro (ou mais) mãos, e pelo exemplo de alguém que sabe o que quer, não se contenta com pouco e sempre batalha, e muito, em busca de seus objetivos;
- Aos amigos, Alberto Egon Schaeffer Filho, Camila Barbiani Salaverry, Denise de Oliveira, Guilherme Dal Pizzol, Guillermo Nudelman Hess, Émerson Barbiero Hernandez, Karina Girardi Roggia, Marcos Rafael Boschetti, Michelle Denise Leonhardt e Rafael Huff, pelas conversas no bar e nos corredores, pelas risadas, pelos almoços, filmes, festas. Todos vocês foram e são muito importantes para mim;
- Aos bolsistas de iniciação científica Felipe Mobus, Joana Matos Fonseca da Trindade, Júlio Gerchman, e Leonardo Golob, pelo auxílio em tarefas específicas no desenvolvimento deste trabalho: implementação de esqueletos de código, discussões e levantamento de questões pontuais nas versões preliminares da implementação da ferramenta, execução de testes e demais atividades operacionais no laboratório. Trabalho muitas vezes "braçal" e entediante, mas não menos importante;
- Ao Prof. Dr. Francisco "Fubica" Brasileiro e à Profª. Dra. Ingrid Jansch-Pôrto, pelos comentários e sugestões incorporados à versão final desta dissertação;
- Ao colega Fábio Olivé Leite, pelo trabalho anterior na ferramenta ComFIRM e pela indicação do caminho de continuidade do trabalho; e ao colega Reynaldo Novaes, representando de forma mais próxima a empresa HP Brasil, por todo apoio e parceria desta empresa junto ao projeto DepGriFE, dentro do qual este trabalho está inserido; e por fim
- À minha família, pelo mais importante: me prover de valores, caráter e ideais.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ARP	Address Resolution Protocol
ComFIRM	Communication Fault Injection through Operating System Resource Modification
CRC	Cyclic Redundancy Check
DProbes	Dynamic Probes
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports
FIRMASM	FIRMAMENT Assembly
FIRMVM	FIRMAMENT Virtual Machine
ICMP	Internet Control Message Protocol
IP	Internet Protocol
LWFI	LightWeigth Fault Injector
MPEG-4	Motion Picture Experts Group standard 4
MR-OSI	Modelo de Referência OSI
NEEDLE	Network Experimentation Environment using DProbes for Link Errors
PFI	Protocol Fault Injection
RFC	Request for Comments
RTC	Real-Time Clock
RTP	Real-Time Protocol
SCTP	Stream Control Transmission Protocol
SIP	Session Initiation Protocol
SWIFI	Software Implemented Fault Injection
TCP	Transmission Control Protocol
ToS	Type of Service
UDP	User Datagram Protocol
XDR	eXternal Data Representation

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A execução de testes é um passo essencial na adoção de novos protocolos de comunicação e sistemas distribuídos. A forma com que estes se comportam na presença de falhas, tão comuns em ambientes geograficamente distribuídos, deve ser conhecida e considerada. Testes sob condições de falha devem ser realizados e as implementações devem trabalhar dentro de sua especificação nestas condições, garantindo explicitamente o funcionamento dos seus mecanismos de detecção e recuperação de erros. Para a realização de tais testes, uma técnica poderosa é a injeção de falhas. Ferramentas de injeção de falhas permitem ao projetista ou engenheiro de testes medir a eficiência dos mecanismos de um sistema antes que o mesmo seja colocado em operação efetiva.

Este trabalho apresenta o projeto, desenvolvimento e teste do injetor de falhas FIRMAMENT. Esta ferramenta executa, dentro do núcleo do sistema operacional, micro-programas, ou *faultlets*, sobre cada mensagem processada para a emulação de situações de falha de comunicação, utilizando uma abordagem de *scripts*.

A ferramenta é implementada como um módulo de núcleo do sistema operacional Linux, tendo acesso total aos fluxos de entrada e saída de pacotes de forma limpa e não intrusiva, permitindo o teste de sistemas baseados nos protocolos IPv4 e IPv6. Seu desempenho é significativo, já que a ferramenta evita que os mecanismos de injeção de falhas sejam invocados nos fluxos que não sejam de interesse aos testes, bem como dispensa a cópia de dados dos pacotes de comunicação a serem inspecionados e manipulados. A aplicabilidade da ferramenta, dada pela sua facilidade de integração a um ambiente de produção, é conseqüência de sua disponibilidade como um módulo de núcleo, podendo ser carregada como um *plugin* em um núcleo não modificado.

As instruções por FIRMAMENT suportadas lhe dão alto poder de expressão dos cenários de falhas. Estas instruções permitem a inspeção e seleção de mensagens de forma determinística ou estatística. Além disso, fornecem diversas ações a serem realizadas sobre os pacotes de comunicação e sobre as variáveis internas do injetor, fazendo-o imitar o comportamento de falhas reais, como descarte e duplicação de mensagens, atraso na sua entrega e modificação de seu conteúdo. Estas características tornam a ferramenta apropriada para a realização de experimentos sobre protocolos e sistemas distribuídos.

Palavras-chave: Protocolos de comunicação, sistemas distribuídos, injeção de falhas, teste, validação experimental.

FIRMAMENT: A Communication Fault Injection Module for Linux

ABSTRACT

The testing phase is essential in the adoption of new communication protocols and distributed systems. Their behavior under the presence of faults, so common in wide area networks, should be known and considered. Tests should be performed under faulty conditions, in which the implementations should follow their specifications, explicitly guaranteeing the correct actions of the target's error detection and recovery mechanisms. Fault injection is a powerful technique to perform such tests. A fault injector allows the designer or test engineer to measure the efficiency of the mechanisms of a system before it is put into effective operation.

This work presents the design, implementation and testing of the FIRMAMENT fault injector. Such tool executes, inside the operating system kernel, micro-programs, or *faultlets*, over each processed message in order to emulate communication fault situations, using a script approach.

The tool is implemented as a kernel module for the Linux operating system, having complete access to the inbound and outbound packet flows, in a clean and non-intrusive way. This allows the testing of targets based on the IPv4 and IPv6 protocols. Its performance is significant, since the tool avoids the invocation of the fault injection mechanisms for the flows which are not of interest to the testing, as well as exempting the copying of the communication packet data to be inspected and manipulated. The applicability of the tool, given by its ease of integration into a production environment, is a consequence of the availability of the tool as a kernel module, which can be loaded as a *plugin* in an existing unmodified kernel.

The instructions supported by FIRMAMENT make it powerful to express fault scenarios. Such instructions allow the inspection and selection of messages in a deterministic or statistic manner. Moreover, they support many actions to be performed over the communication packets and the internal fault injector variables, making it mimic the behavior of real faults, like message drop and duplication, late delivery and content modification. These features make the tool appropriate to perform experiments over protocols and distributed systems.

Keywords: communication protocols, distributed systems, fault injection, testing, experimental validation.

1 INTRODUÇÃO

Neste capítulo são apresentadas as motivações deste trabalho, através de conceitos-chave a ele relacionados, e os objetivos almejados levando em conta estas considerações. Entre as motivações estão o conceito de dependabilidade, sua importância e a necessidade de sua avaliação. Como técnica experimental para avaliação da dependabilidade são sugeridas as técnicas de injeção de falhas. Parte-se, em seguida, ao relacionamento da dependabilidade com os ambientes de sistemas distribuídos, introduzindo modelos de falhas tipicamente usados nesses sistemas e uma revisão histórica da aplicação de injeção de falhas em sistemas de comunicação. A partir dessas informações objetiva-se a construção de um injetor de falhas de comunicação de alto desempenho e poder de expressão do cenário de falhas para ambientes de comunicação.

1.1 Motivação

1.1.1 Avaliação de dependabilidade

A adoção de sistemas computacionais nas mais diversas tarefas do cotidiano e a conseqüente dependência posta nesta aplicação gera uma demanda crescente por sistemas com a característica de *dependabilidade*. Apesar do conceito ser complexo (LAPRIE, 1998), dependabilidade abrange duas características principais: *disponibilidade*, ou seja, a capacidade de um sistema estar pronto para uso quando exigido; e *confiabilidade*, a capacidade do sistema permanecer em serviço, mesmo na presença de falhas (*faults*).

Infelizmente, erros de especificação e projeto, bem como falhas de hardware, podem ter conseqüências desastrosas durante a vida útil de um sistema computacional. Técnicas de tolerância a falhas são utilizadas para aumentar sua dependabilidade. O desenvolvedor pode, por exemplo, incorporar ao projeto do sistema técnicas de detecção e recuperação de erros ou mascaramento através de réplicas. Ao adicionar estes mecanismos à especificação e ao projeto, o sistema fica sujeito a novos tipos de falhas: as que afetam os mecanismos de tolerância a falhas em si. Portanto, a sua implementação deve ser validada de forma cuidadosa. Deve-se garantir que os mecanismos de recuperação sejam capazes de mascarar a ocorrência de falhas e que as falhas não mascaradas levem o sistema através de um processo de falhas conhecido e esperado.

A avaliação dos mecanismos de tolerância a falhas pode ser vista como um dos aspectos de uma avaliação de dependabilidade mais ampla, que busca obter métricas que

indicam tanto quantitativa quanto qualitativamente como um sistema se comporta durante sua vida útil, relativo à ocorrência de falhas. A avaliação da dependabilidade não depende exclusivamente da avaliação dos mecanismos de tolerância a falhas: a qualidade dos componentes, por exemplo, pode refletir-se na dependabilidade sem o uso de técnicas de tolerância a falhas. Realisticamente, entretanto, sistemas de missão crítica não podem depositar toda sua confiança de funcionamento na qualidade de seus componentes. Como as técnicas de tolerância a falhas não podem ser abandonadas, a sua avaliação é indispensável.

Além disso, a qualidade dos componentes costuma ser indicada por medidas estatísticas como o tempo médio entre falhas[1] (MTBF, *mean time between failures*), ou tempo médio para a ocorrência da primeira falha (MTTF, *mean time to failure*). Pela característica aleatória e imprevisível das falhas, esses valores para componentes encontrados no mercado costumam ser da ordem de centenas de milhares de horas, o que torna impraticável a espera passiva por suas falhas para estudo do comportamento do sistema. A observação simultânea de um grande número de componentes pode, estatisticamente, acelerar a ocorrência de falhas, mas os custos envolvidos nesse processo tornam-se muito altos: ganha-se em previsibilidade temporal, pois o tempo é reduzido, mas perde-se previsibilidade local, pois o número de componentes a ser observado é muito maior.

As técnicas de avaliação de dependabilidade empregam um modelo do sistema, ou ainda um protótipo ou versão final, para o estudo de seu comportamento na presença de falhas. As *técnicas analíticas* estão relacionadas ao conceito de modelagem e simulação. Nelas, procedimentos matemáticos são executados sobre um modelo matemático do sistema para obtenção das métricas de dependabilidade. Um modelo é sempre uma aproximação da realidade, e sua simplificação, ao mesmo tempo que facilita e agiliza o processo, pode ter efeitos não antecipados na validação, caso ele não considere todas as características relacionadas à dependabilidade. A busca de um modelo nem tão simples que desconsidere essas características fundamentais, nem tão complexo que torne a simulação impraticável, é um desafio por si só, entretanto simulações tendem a ser mais controláveis e possuir maior reproducibilidade.

Já nas *técnicas experimentais*, a idéia é observar um protótipo ou versão pronta do sistema na presença de falhas, e, a partir daí, levantar conclusões sobre sua dependabilidade. Como visto anteriormente, empregar técnicas relacionadas ao uso, isto é, simplesmente utilizar o sistema e aguardar a ocorrência natural de falhas, é impraticável, devido aos longos períodos envolvidos e a dificuldade de reproduzir situações de teste.

1.1.2 Injeção de falhas

Uma técnica bastante comum que objetiva a validação experimental é a *injeção de falhas* (CLARK; PRADHAN, 1995). Nela, adota-se uma atitude ativa perante o sistema, introduzindo de forma controlada falhas de um dado cenário em um sistema de teste, o alvo, e observando seu comportamento. Assim, existe a vantagem de se estar trabalhando sobre um sistema real, tendo portanto uma avaliação *efetiva* de sua dependabilidade. Por não se estar sujeito à variabilidade temporal e local das falhas naturais, é possível observar o sistema sob a ocorrência de falhas específicas em momentos específicos.

[1] Os conceitos de MTBF e MTTF são anteriores ao trabalho de Laprie (1998), referindo-se a *failures* como as manifestações das falhas não tratadas.

O uso da injeção de falhas fornece parâmetros indicativos da dependabilidade. A cobertura mostra a porcentagem (associada ao tipo e localização) das falhas que são tratadas e conseqüentemente não se manifestam ao usuário. A latência de falhas (e de erros) expressa a diferença temporal entre a ocorrência de uma falha (ou erro) e o seu erro (ou defeito) correspondente. Com a injeção de falhas também é possível determinar a propagação das falhas e erros, localizando os gargalos de dependabilidade, bem como os locais onde essas falhas e erros são tratados. É possível ainda verificar se há variação de desempenho sob a ocorrência de falhas. Em sistemas não tolerantes a falhas, a injeção de falhas também pode ser utilizada para avaliar o processo de manifestação das falhas quando estas ocorrem. Com isto, torna-se possível o projeto de sistemas com maior dependabilidade e e mais efetividade.

Desde os anos 70, a injeção de falhas é usada para medir a dependabilidade de sistemas tolerantes a falhas. Somente na década seguinte, entretanto, ela começou a ser utilizada em pesquisas experimentais com o objetivo inicial de estudar a propagação de erros e avaliar novos mecanismos de detecção de falhas.

As ferramentas de injeção de falhas costumam ser divididas em duas grandes famílias, relacionadas à forma como são implementadas. Os injetores por *hardware* adicionam componentes físicos ao sistema, como ponteiras de prova ativas, podendo então atuar diretamente no nível elétrico ou causando distúrbios eletro-magnéticos que levam à ocorrência de falhas.

Já os injetores por *software* (SWIFI, *software implemented fault injection*) (CARREIRA; MADEIRA; SILVA, 1995) são porções de código que emulam falhas corrompendo o estado de um sistema alvo. O estado do alvo é modificado, forçando o sistema a funcionar como se falhas tivessem ocorrido. Durante uma rodada de testes, o injetor basicamente interrompe ou muda, utilizando quaisquer ganchos ou mecanismos de desvio disponíveis, a execução do alvo de forma controlada, emulando falhas através da inserção de erros em seus diversos componentes. Assim, o conteúdo de estruturas lógicas como registradores, posições de memória, áreas de código ou sinalizadores pode ser modificado. De fato, injetores de falhas por software são na realidade injetores de erros, devido ao nível que atuam, mas esta distinção de nomenclatura não é usual, sendo tanto os injetores implementados por software quanto por hardware chamados de *injetores de falhas*.

Uma campanha completa de injeção de falhas inclui as seguintes fases: inicialmente, um cenário de falhas para o teste do experimento deve ser especificado. Isto inclui decisões a respeito da localização e tipo das falhas e momento de injeção. Em seguida, a injeção das falhas efetivamente deve ocorrer. Durante esta fase, dados do experimento são coletados e como fase final estes devem ser analisados para obtenção das medidas de dependabilidade do sistema alvo.

1.1.3 Intrusividade

Intrusividade é a influência direta sobre o sistema de suporte causada pela instrumentação necessária para a injeção de falhas (DEPENDABILITY BENCHMARKING PROJECT, 2001). Essa influência não pode ser evitada – já que a modificação do sistema é inerente à sua instrumentação – restando aos desenvolvedores de ferramentas de injeção de falhas buscarem sua minimização.

Existem duas óticas pelas quais a intrusividade pode ser considerada. A *intrusividade espacial* está associada à forma estrutural na qual o sistema de suporte necessita ser alterado para instrumentação. A este aspecto estão associados os mecanismos utilizados para interceptação do fluxo de execução do alvo, bem como o espaço de memória adicional utilizado pelo código de injeção de falhas. Por exemplo, uma ferramenta que exija a modificação direta de seu código fonte para instrumentação apresenta uma alta intrusividade espacial, enquanto a utilização de recursos de depuração indicam um caminho para minimizá-la.

A *intrusividade temporal* está relacionada à sobrecarga associada ao tempo de processamento do alvo, sendo resultado da execução das rotinas de injeção de falhas e/ou das trocas de contexto entre o injetor e o alvo. Ela é particularmente importante em sistemas de tempo real (DAWSON et al., 1996), onde a previsibilidade temporal é tão importante quanto a precisão lógica e aritmética dos valores da computação, e esta previsibilidade pode ser perturbada pela execução dos mecanismos de injeção. Deve-se garantir que nenhum erro não intencional (ou seja, não representado no modelo de falhas) será introduzido no sistema pela utilização da injeção de falhas, incluindo-se os erros de temporização.

1.1.4 Falhas de comunicação

Quando sistemas computacionais são conectados através de redes de comunicação, a sua característica distribuída bem como sua dispersão geográfica trazem preocupações relacionadas à dependabilidade normalmente mais sérias do que as existentes em sistemas auto-contidos. Sendo a comunicação feita através de sinais eletromagnéticos no ambiente físico, a natural interferência nesses sinais pode levar à perda de informação, mesmo sem a falha de nenhum componente de hardware do sistema. A ocorrência de falhas nesses sistemas é uma certeza e os efeitos do mundo real tornam-se desafios para o alcance de uma alta disponibilidade.

O uso de técnicas de tolerância a falhas, portanto, deixa de ser simplesmente um valor agregado de um sistema tornando-se algo imprescindível para que o mesmo trabalhe de forma aceitável. As técnicas de tolerância a falhas costumam apresentar um compromisso de desempenho ou custo, mas apesar disso são fundamentais no funcionamento de sistemas distribuídos. Em sistemas de comunicação, é comum o emprego de mecanismos de tolerância a falhas, como códigos de verificação e correção, ou retransmissão.

A caracterização dos tipos de falhas que um sistema está sujeito é dada por um *modelo de falhas*. Para sistemas distribuídos, um modelo de falhas consagrado é o apresentado por Cristian (1991). Nele, um nó entra em colapso quando pára de enviar ou receber mensagens. Um nó omite respostas quando não envia ou recebe algumas das mensagens. Diz-se que o nó tem falhas de temporização quando atrasa ou avança mensagens. O comportamento bizantino do nó, por sua vez, equivale à alteração dos valores (conteúdos) das mensagens, ou se um nó envia mensagens contraditórias a nós diferentes. Este modelo pode ser implementado através de técnicas de injeção de falhas.

Apesar de existirem diversas ferramentas para o teste de protocolos (PARKER; SCHMECHEL, 1998), estas em geral dão ênfase a modelos de falhas para comunicação de dados, não sistemas distribuídos. Apesar de permitirem emular situações de perda de pacotes, variar o atraso e limitar a banda de transmissão, estas ferramentas em geral não permitem o disparo de falhas relacionadas a eventos de mais alto nível, como a interpretação do

conteúdo das mensagens para este disparo.

1.1.5 Injeção de falhas de comunicação

Ao prototipar protocolos e aplicações de redes, estes se tornam os sistemas alvos. Sua validação, entretanto, encontra limitações inerentes de tempo e espaço. O injetor de falhas por software representa uma carga extra no alvo e isto pode alterar o tempo de execução das tarefas, causando um distúrbio nas exigências temporais ou no desempenho. Portanto, ao desenvolver um injetor, cuidados especiais devem ser tomados para limitar a intrusividade do injetor no sistema alvo, minimizando o efeito de sondagem da medida de dependabilidade.

Para qualquer experimento, um *cenário de falhas* deve ser descrito a partir do modelo de falhas. Considerando um injetor de falhas de comunicação, o cenário depende da forma com que a ferramenta lida com a comunicação, ou seja, como as mensagens são selecionadas e manipuladas (DAWSON et al., 1996).

Um cenário deve descrever quais mensagens o injetor deve selecionar para manipular, seja por meios determinísticos ou probabilísticos. Não basta operar aleatoriamente sobre as mensagens, a menos que a distribuição de seleção siga um modelo conhecido. O mesmo pode ser dito da manipulação: as ações a serem executadas sobre as mensagens devem ser conhecidas.

Além das ferramentas de testes de protocolos, existem diversas ferramentas que objetivam a injeção de falhas de comunicação efetivamente. Considerando falhas de comunicação, a maioria das ferramentas é específica para alguma tecnologia de domínio restrito. Por exemplo, EFA (ECHTLE; LEU, 1992) executa somente no sistema distribuído A/ROSE. SFI (ROSENBERG; SHIN, 1993) e DOCTOR (HAN; SHIN; ROSENBERG, 1995) estão restritos ao sistema distribuído HARTS. ORCHESTRA (DAWSON et al., 1996), originalmente desenvolvido para o sistema operacional Mach, foi mais tarde portado para Solaris. Quando as ferramentas não estão restritas a uma plataforma específica, elas podem ter sido projetadas para funcionar com uma biblioteca de aplicação ou de sistema específicas, como no caso de INFIMO (BARCELOS et al., 2004), que utiliza uma aplicação de rede bastante simples desenvolvida para comparar abordagens de injeção de falhas, ou FIONA (JACQUES-SILVA et al., 2004), que destina-se somente a aplicações escritas em Java.

1.2 Objetivos

Considerando-se as limitações acima descritas, o objetivo deste trabalho é o projeto e desenvolvimento de uma nova ferramenta de injeção de falhas de comunicação utilizando o conhecimento adquirido no desenvolvimento deste tipo de ferramenta pelo Grupo de Tolerância a Falhas/UFRGS nos últimos anos (descrito no Apêndice C). Duas características desejadas desta ferramenta podem ser destacadas:

- Alto poder de expressão de cenários de falhas, permitindo ao engenheiro de testes especificar de forma precisa e completa condições de falhas.

- Baixa intrusividade temporal, de forma que a instrumentação causada pelo injetor não afete os requerimentos temporais intrínsicos dos protocolos e sistemas alvo.

Considerando-se a implementação desta ferramenta, busca-se ainda que ela possa facilmente ser instalada em sistemas em execução, sem a exigência de os colocar fora de operação, nem de recompilar o núcleo (*kernel*) do seu sistema operacional, tendo assim uma baixa intrusividade espacial. Dessa forma, ganha-se também a capacidade de modificar o cenário de falhas durante a execução dos alvos. Adicionalmente, a ferramenta deve ser indepentende de arquitetura, podendo ser utilizada nos diversos *ports* do núcleo Linux para diferentes tipos de hardware, bem como oferecer suporte a aplicações e protocolos que utilizem as versões 4 e 6 do protocolo IP (IPv4 e IPv6) para comunicação, independente da biblioteca de comunicação utilizada.

Esta ferramenta, chamada FIRMAMENT (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*) se destina à validação experimental e é construída utilizando a extensibilidade do núcleo Linux através de módulos relocáveis, o que minimiza o efeito de sondagem no núcleo. A ferramenta objetiva injetar falhas de comunicação para a validação dos aspectos de tolerância a falhas de protocolos de comunicação e sistemas distribuídos.

Como em ORCHESTRA, a ferramenta provê diversas opções para a injeção de falhas de comunicação e permite a especificação de testes através de *scripts*. Entretanto, diferentemente desta ferramenta, FIRMAMENT é concebido para utilizar características específicas do núcleo de código fonte aberto Linux. Além disso, uma diferença importante existe entre as duas ferramentas: a primeira utiliza uma linguagem de alto nível interpretada para a descrição dos cenários de teste, o que impõe sobrecarga considerável sobre o processamento de pacotes, enquanto FIRMAMENT utiliza uma máquina virtual para a interpretação de instruções baseadas em *bytecode*, o que dá à ferramenta maior poder para a especificação dos cenários com baixa intrusividade.

1.3 Organização do texto

Este trabalho apresenta a ferramenta FIRMAMENT: seu histórico, questões de projeto e desenvolvimento, especificação e exemplos de uso. No capítulo seguinte são apresentadas diversas ferramentas de injeção de falhas de comunicação. Em seguida, é descrita a ferramenta ComFIRM, que serviu de inspiração e motivou este trabalho. As limitações de ComFIRM são introduzidas ainda no Capítulo 3, partindo para o estudo das soluções encontradas e como estas foram utilizadas no projeto da ferramenta FIRMAMENT no Capítulo 4. O Capítulo 5 descreve a especificação da ferramenta: seus componentes, instruções, mecanismos e comandos. No capítulo seguinte são apresentados testes mostrando a viabilidade e funcionamento correto da ferramenta. O capítulo final faz um apanhado dos resultados alcançados com este trabalho, bem como considerações finais e perspectivas futuras. O Apêndice A apresenta detalhes sobre a extensão da ferramenta. O Apêndice B apresenta ainda exemplos de especificações de testes e ações comuns de interesse no tratamento de pacotes para o injetor FIRMAMENT. Finalmente, o Apêndice C relaciona o desenvolvimento dos diversos injetores desenvolvidos no Grupo de Tolerância a Falhas/UFRGS, através de um histórico cronológico.

2 INJETORES DE FALHAS DE COMUNICAÇÃO

Tanto a emulação de redes de computadores quanto a injeção de falhas de comunicação podem ser vistos como diferentes aspectos de um mesmo processo. A manipulação do estado dos canais de comunicação ou dos nós de um sistema distribuído pode ser empregada tanto para a avaliação dos seus mecanismos de tolerância a falhas quanto de seu desempenho e escalabilidade em ambientes heterogêneos.

Nestes ambientes, o teste de protocolos de comunicação e aplicações distribuídas torna-se complexo, uma vez que as redes não se tornam apenas mais rápidas, mas também mais diversas, tanto relativo aos equipamentos e tecnologias utilizados quanto ao tráfego carregado. A emulação de redes tenta reduzir esta complexidade através do emprego de um conjunto restrito de equipamentos para simular o comportamento de uma vasta gama de tecnologias de comunicação, através da variação de seu desempenho, como atraso, vazão e taxa de perda de pacotes. Nestas condições "semi-sintéticas", encontram-se as facilidades de ambientes reais (como o teste efetivo dos alvos, e não de seus modelos simulados) não incorrendo na complexidade e custos do teste exaustivo em todas as tecnologias e condições sobre as quais o alvo será empregado.

Considerando-se o teste relativo aos mecanismos de tolerância a falhas, a experimentação deve ser feita de forma a forçar o sistema a certos estados naturalmente difíceis de serem alcançados, ativando determinados caminhos de execução. Além disso, é interessante que a injeção de falhas no sistema possa emular participantes "mal-comportados" em uma computação distribuída. Da mesma forma que na emulação de redes, o desempenho pode ser alterado, ou ainda o próprio conteúdo das mensagens.

Apesar das ferramentas utilizadas para ambos os fins serem semelhantes, já que trabalham sobre o fluxo de comunicação interceptando mensagens e modificando seu processamento normal, as ferramentas que visam a emulação de falhas tendem a alterar o comportamento da comunicação estatisticamente, atuando sobre as mensagens a serem descartadas, atrasadas ou alteradas, como fluxos únicos.

Já as ferramentas de teste dos mecanismos de tolerância a falhas de protocolos e sistemas distribuídos costumam permitir a atuação em mensagens específicas, o que torna possível o teste de eventos relacionados ao estado do protocolo. Tipos específicos de mensagens ativam os mecanismos de tolerância a falhas, e é sobre estas mensagens que o injetor deve atuar. Por essa razão, estas ferramentas exigem maior controle sobre a seleção das mensagens a serem manipuladas. Sendo a seleção feita apenas de forma estatística, a possibilidade de falha de uma mensagem específica de controle torna-se baixa,

aproximando-se da situação de funcionamento real do sistema, sem a injeção de falhas, que perde seu sentido a menos que se utilizem taxas irreais, e portanto não representativas, para a obtenção do mesmo efeito.

Este capítulo apresenta uma lista não exaustiva, mas significante, de ferramentas utilizadas para a emulação de redes ou teste experimental de protocolos e aplicações distribuídas através da manipulação de mensagens e injeção de falhas. Apesar da existência de outras ferramentas, estas foram escolhidas por apresentarem características marcantes que as diferenciam das demais. As limitações e aspectos positivos de cada ferramenta aqui apresentadas foram considerados durante o desenvolvimento de FIRMAMENT.

2.1 Dummynet

Dummynet (RIZZO, 1997) é uma das primeiras ferramentas de emulação de redes desenvolvidas para sistemas de código aberto da família UNIX, especificamente FreeBSD. A utilização de Dummynet em um roteador permite a emulação dos efeitos de filas finitas, limitações de banda e atrasos de comunicação.

A facilidade de uso da ferramenta é resultado da sua carga através de um módulo a um núcleo FreeBSD em execução. A partir de então, ela intercepta pacotes de comunicação, inserindo-os em filas por períodos determinados, emulando o cenário configurado. A configuração de atrasos é dada por fluxo e de forma constante, não sendo possível especificar intervalos de ocorrência. A perda de pacotes pode se dar tanto pelo estouro das filas configuradas como de forma estatística através de uma taxa fixa (porcentagem).

2.2 NIST Net

Para a emulação de redes no sistema operacional Linux, a ferramenta mais completa é NIST Net (CARSON; SANTAY, 2003). O conjunto de efeitos que a ferramenta pode impor sobre os pacotes de comunicação incluem atrasos (fixos e variáveis), reordenamento (resultado de atrasos variáveis), perda (aleatória e dependente de congestionamento), duplicação e limitações de banda. NIST Net também é um módulo de núcleo, podendo ser carregado em sistemas Linux em execução.

A peculiaridade de NIST Net está na forma precisa com que emula as métricas de rede, podendo utilizar modelos estatísticos próprios, bem como criados em nível de usuário e passados para o módulo através de tabelas. As rotinas utilizadas para emulação não necessitam ser modelos realísticos dos mecanismos internos das redes e roteadores, mas ser computacionalmente simples e adaptáveis para imitar os variados comportamentos de rede.

A resolução temporal da ferramenta também é bastante precisa: o módulo utiliza o controlador do relógio de tempo real (RTC, *Real-Time Clock*) disponível nos PCs, geralmente usado apenas de forma trivial pelo núcleo. Isto fornece à ferramenta uma granulosidade temporal de aproximadamente 120 μs. Entretanto, o uso do RTC impede a ferramenta de ser usada em núcleos compilados com o controlador de dispositivo do RTC, a menos que este controlador seja compilado como um módulo de sistema, e neste

caso ambos os módulos não podem estar carregados simultaneamente.

2.3 Honeyd

A peculiaridade da ferramenta Honeyd (PROVOS, 2004) está no fato dela não objetivar a emulação de redes para o teste de protocolos e aplicações, mas sim para a monitoração de ataques de segurança. Honeyd permite a criação de topologias virtuais para enganar adversários e ferramentas de mapeamento de rede. Sua intenção, portanto, é diferente da de outras ferramentas, que tentam reproduzir fielmente o comportamento das redes: a ferramenta apenas simula o suficiente para confundir ataques.

Uma capacidade interessante da ferramenta é, ao emular pilhas TCP, poder adotar diversas *personalidades TCP*, fazendo assim com que os nós emulados possam se passar por diferentes sistemas operacionais. Considerando-se o comportamento associado à emulação de redes, é possível integrar sistemas reais à topologia virtual de Honeyd, impondo limitações de banda e atraso sobre aplicações reais.

2.4 NFTAPE

NFTAPE (STOTT et al., 2000) é um *framework* desenvolvido com o objetivo de avaliar a dependabilidade de sistemas distribuídos, sendo que: (i) suporta modelos de falhas múltiplos, (ii) aceita diversos mecanismos de disparo de falhas, (iii) é apropriado para múltiplos alvos, e (iv) fornece métodos versáteis de coleta de dados e relato de erros. Para isto, NFTAPE emprega uma abordagem modular, com componentes independentes formando a ferramenta de injeção de falhas.

O papel comumente desempenhado pelo injetor de falhas é assumido pelos módulos LWFI (*LightWeigth Fault Injector*, injetor de falhas leve). O termo se refere à característica dos LWFIs de serem pequenos programas responsáveis por injetar falhas, que não se preocupam com o disparo das mesmas ou com a detecção de erros. A indicação de quando e qual falha deve ser injetada é fornecida por um processo de disparo, semelhante a um LWFI. As demais funções, como aquisição de dados, configuração do experimento e comunicação são desempenhados pelo próprio NFTAPE.

LWFIs podem ser empregados para a injeção de falhas por diversos métodos, como através de mecanismos de depuração, baseados em controladores de dispositivo, com mecanismos específicos para um alvo, baseados em simulação ou ainda a utilização de injeção de falhas físicas (por *hardware*). Estes módulos são bastante simples, já que apenas injetam as falhas como indicado pelos processos de disparo. Quanto ao processo de disparo das falhas, a ferramenta pode utilizar modos comuns como os baseado em tempo, em caminhos (quando o alvo alcança algum evento específico) ou ainda relacionados à fadiga de componentes do alvo.

NFTAPE foi concebido para o teste de experimentos científicos a serem realizados no espaço. Por este motivo, seu interesse principal é na emulação de falhas geradas pela irradiação de componentes, como alteração de registradores e posições de memória. Apesar disto, também há suporte para falhas de comunicação, através da injeção de falhas

diretamente nos controladores de comunicação dos sistemas distribuídos.

A separação entre mecanismos de disparo e de injeção de falhas (através dos injetores leves) utilizada em NFTAPE torna a ferramenta portável e apropriada para o teste de alvos como os descritos acima, mas apresenta alguns inconvenientes para o teste de protocolos de comunicação. Nestes, o acoplamento entre o disparo e a injeção de falhas é apropriado, pois as falhas tendem a ser injetadas sobre pacotes de acordo com seu conteúdo. Já que tanto o mecanismo de disparo, quanto o módulo responsável pela injeção necessitam acesso ao mesmo fluxo de comunicação, sua integração como feita nas ferramentas de injeção de falhas tradicionais mostra-se mais pertinente.

2.5 ORCHESTRA

ORCHESTRA (DAWSON et al., 1996) é uma ferramenta para avaliar e validar características temporais e de tolerância a falhas de protocolos de comunicação, tentando identificar violações da especificação de protocolos e detectar erros no seu projeto e implementação. A ferramenta introduz o conceito de sondagem e injeção de falhas orientada a *scripts*. Um dos objetivos da ferramenta é tentar minimizar a intrusividade no protocolo alvo.

Como ORCHESTRA pretende ser uma ferramenta para o teste de aplicações distribuídas e protocolos de comunicação, a portabilidade para outras plataformas e a capacidade de se inserir o mecanismo de injeção de falhas diretamente na pilha de protocolos são objetivos chave. Para isso, a ferramenta permite a adição de novas camadas à pilha de protocolos, através de uma interface padrão chamada PFI (*Protocol Fault Injection*). Uma PFI é adicionada à pilha de protocolos abaixo do protocolo alvo a ser testado. Inserindo-se a camada de injeção de falhas entre duas camadas da pilha de protocolos, o código da camada alvo não necessita modificação.

Nestes pontos, ORCHESTRA interpreta *scripts* de injeção de falhas escritos na linguagem de alto nível (interpretada) TCL. Com ela, o engenheiro de testes pode especificar *scripts* suficientemente poderosos para a manipulação de mensagens. O mecanismo de *scripts* pode ser visto como uma máquina de estados que opera sobre a mensagem. A máquina de estado tem acesso ao conteúdo da mensagem e a capacidade de acompanhar o histórico de mensagens e manter contadores e outras informações de estado. Esta abordagem faz desnecessária a recompilação da camada PFI para novos testes sobre um protocolo por ela já suportado. A utilização da linguagem TCL se dá pela sua popularidade entre desenvolvedores, por se tratar de uma linguagem já existente e empregada em outras ferramentas, como o simulador de rede ns-2 (FALL; VARADHAN, 2005). Combinando-a com uma coleção de bibliotecas predefinidas, o engenheiro de testes possui uma ferramenta efetiva para escrever a maioria dos *scripts*.

Por outro lado, o injetor ORCHESTRA foi testado em protocolos de mais alto nível e aplicações baseadas em *sockets*. Assim, o tráfego passante pela ferramenta é restrito ao protocolo alvo, e a intrusividade da ferramenta não se manifesta na inspeção dos demais pacotes de comunicação. Além disso, estes protocolos são implementados em nível de usuário. O injetor é executado como uma *thread* adicional do processo alvo, evitando as trocas de contexto que existiriam se a injeção fosse realizada em um protocolo interno ao núcleo, como IP, TCP ou UDP.

Tabela 2.1: Comparativo de ferramentas que manipulam o fluxo de comunicação

Ferramenta	Plataforma	Foco	Característica Principal
Dummynet	FreeBSD	Emulação para teste de redes.	Carga através de módulo de núcleo.
NIST Net	Linux	Emulação para teste de redes.	Carga através de módulo de núcleo, precisão temporal.
Honeyd	*BSD, Linux, Solaris	Emulação para monitoração de ataques de segurança.	Emulação de *personalidades TCP*.
NFTAPE	Linux, Solaris, Windows, Lynx	Avaliação de dependabilidade de sistemas distribuídos e protocolos.	Conceito de injetor de falhas leve (LWFI).
ORCHESTRA	RT-Mach, Solaris	Avaliação de dependabilidade de sistemas distribuídos e protocolos.	Execução de *scripts* na camada PFI.

A Tabela 2.1 mostra um comparativo das ferramentas apresentadas, destacando seu foco e característica principal. Para a ferramenta ORCHESTRA, o destaque se dá pelo conceito de injeção de falhas orientada a *scripts*, bastante poderosa para o teste de protocolos e sistemas distribuídos. Entretanto, a escolha de uma linguagem de alto nível a torna inapropriada para aplicações de alta vazão e protocolos de níveis inferiores. Uma linguagem de representação mais compacta e de processamento mais eficiente é necessária para injetores implementados internamente ao núcleo do sistema operacional, onde protocolos de rede e transporte costumam ser implementados. Esta abordagem foi utilizada na ferramenta ComFIRM (LEITE, 2000). Apesar de ComFIRM utilizar uma linguagem bastante primitiva e limitada, e ter sua extensão e aplicação dificultadas, FIRMAMENT herda diversas de suas características e decisões de projeto. Por esta razão, o injetor ComFIRM é descrito detalhadamente de forma individual no próximo capítulo.

3 O INJETOR DE FALHAS COMFIRM

O injetor de falhas ComFIRM (*COMmunication Fault Injection through operating system Resource Modification*) (LEITE, 2000), desenvolvido previamente no Grupo de Tolerância a Falhas/UFRGS, endereça as principais necessidades quanto à injeção por software de falhas de comunicação. Como em ORCHESTRA (DAWSON et al., 1996), os tipos de falhas possíveis de emulação pela ferramenta são: omissão de envio e recebimento, colapso de nó e canal, temporização e falhas bizantinas.

Mesmo com algumas limitações, a ferramenta possui uma forma de funcionamento interessante, associando ao processamento dos pacotes de comunicação a execução de *scripts* simples e de processamento eficiente, responsáveis por alterar o fluxo de comunicação, assim emulando o cenário de falhas configurado. Apesar de FIRMAMENT expandir o poder de processamento destes *scripts* e interceptar a execução do subsistema de comunicação de forma menos intrusiva, como mostrado no próximo capítulo, seu princípio de funcionamento é bastante semelhante ao de ComFIRM, e por isso a descrição mais detalhada desta ferramenta neste capítulo.

3.1 Características

ComFIRM é um injetor de falhas de comunicação para sistemas Linux. A escolha deste ambiente como plataforma para experimentação de injeção de falhas de comunicação se deu, inicialmente, pela sua natureza de software livre. Isto torna uma grande quantidade de informações sobre este sistema disponível, sejam manuais de usuário, documentação sobre o funcionamento e implementação de seu núcleo, listas e grupos de discussão, livros, ou ainda, é claro, o próprio código fonte do sistema.

Duas idéias básicas regeram o projeto e desenvolvimento de ComFIRM: primeiro, a ferramenta deveria possibilitar a modificação da configuração e da descrição dos cenários de falhas de um experimento durante sua execução. Além disso, a ativação das falhas deveria ocorrer através de regras simples, que não sobrecarregassem o processamento de mensagens.

3.2 Funcionamento

O injetor ComFIRM trabalha com o processamento, sobre os pacotes de comunicação, de *scripts* simples compostos de operações de seleção e manipulação. As mensagens

podem ser selecionadas baseando-se em três classes de critérios: (i) o conteúdo da mensagem, (ii) o fluxo das mensagens, e (iii) fatores externos às mensagens.

A seleção baseada no conteúdo considera algum padrão existente na mensagem, como mensagens de confirmação ou solicitações de encerramento de conexão. Quando a seleção é baseada no fluxo de mensagens, o conteúdo é ignorado, e todas as mensagens que casam com uma mesma regra que não testa o conteúdo são consideradas como um único grupo. Assim é possível, por exemplo, manipular uma porcentagem das mensagens, ou mesmo n em cada m mensagens, onde n e m são números inteiros. A seleção baseada em elementos externos considera condições baseadas nos recursos internos do injetor, como temporizadores, contadores e sinalizadores. É possível, ainda, que estes três tipos de operação sejam combinados.

Após determinadas quais mensagens serão selecionadas, é necessário especificar como estas serão manipuladas. Três categorias de manipulação podem ser identificadas: ações internas à mensagem (conteúdo), ações sobre a mensagem (entrega) ou ainda as não relacionadas diretamente às mensagens, mas sim ao estado do injetor. A primeira categoria define ações que modificam algum campo da mensagem. A segunda, define as ações relacionadas à entrega, como o descarte (onde o processamento da mensagem selecionada é simplesmente encerrado), o atraso (entrega atrasada) e a duplicação. A terceira categoria inclui ações que manipulam o estado do injetor de falhas, como seus contadores, temporizadores e sinalizadores. Novamente, é possível combinar diversas ações de uma mensagem.

Deve-se notar que, ao modificar o conteúdo de mensagens através da manipulação, todos os códigos de detecção e/ou correção que o protocolo implementa podem tornar-se inválidos. Se o objetivo é testar o mecanismo de detecção dos protocolos, esta abordagem é apropriada. Se o objetivo é testar protocolos de mais alto nível, é necessário observar quais são os protocolos de suporte usados. Se um protocolo de suporte como UDP, que fornece serviço de entrega não confiável de datagramas, for utilizado, a manipulação do conteúdo das mensagens faz sentido e as falhas injetadas alcançarão os protocolos alvo nos níveis superiores. Entretanto, se protocolos inferiores que tratam erros, como TCP, forem usados como base para protocolos alvo de *multicast* e sistemas de comunicação de grupo, será o próprio protocolo de suporte que reagirá e recuperará os erros. O protocolo alvo nos níveis superiores não terá conhecimento das falhas injetadas.

A configuração em tempo de execução é feita através da manipulação de arquivos virtuais existentes no sistema de arquivos `proc`. Quatro arquivos são usados: a escrita em `ComFIRM_Control` permite que comandos gerais sejam passados para a ferramenta, como ativação, desativação e nível de detalhamento da captura de eventos; a leitura de `ComFIRM_Log` fornece a seqüência de eventos (*log*) de injeção de falhas; finalmente, `ComFIRM_RX_Rules` e `ComFIRM_TX_Rules` recebem os conjuntos de regras de injeção de falhas usadas para recepção e envio, respectivamente.

3.2.1 Instruções de seleção e manipulação

A operação de ComFIRM é feita através de *scripts* descritos usando uma linguagem de *bytecode* desenvolvida para representar um experimento através de códigos de operação. Conjuntos de regras são compostos de regras individuais, e podem ser independentemente configurados para os fluxos de recepção e envio, através dos arquivos descritos

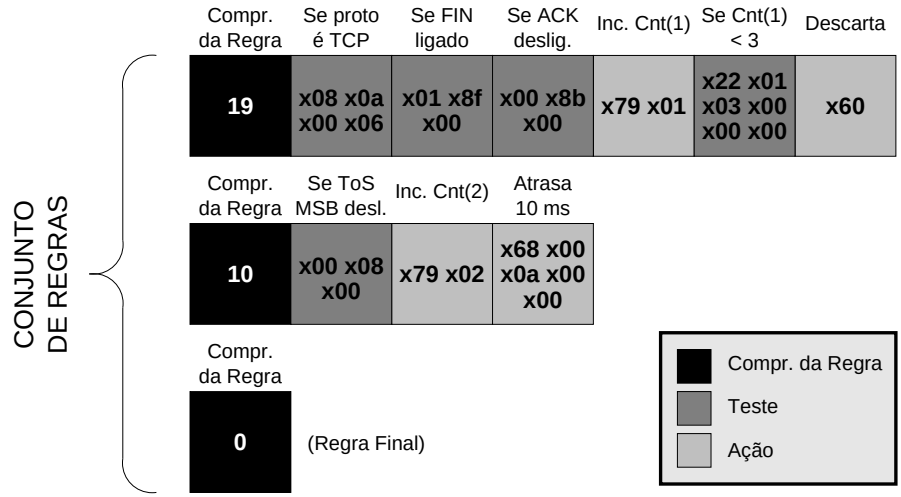

Figura 3.1: Conjunto de regras de ComFIRM

na seção anterior. Uma regra é composta de um conjunto de operações, ou instruções, que determinam uma condição de seleção ou ação de manipulação, arranjadas em uma estrutura encadeada da forma *se-então*.

Isto significa que somente testes que retornem um valor positivo podem ser encadeados, e o fluxo de execução de um *script* prossegue apenas em uma direção. As instruções de uma regra são avaliadas em seqüência até que uma condição de teste retorne 'falso', ou uma ação associada a categoria de entrega (atraso, duplicação ou descarte) seja alcançada. Se um teste é avaliado falso, o processamento é continuado na próxima regra. Se uma mensagem for descartada ou atrasada, nenhuma próxima regra é processada para aquele pacote. A Figura 3.1 apresenta a relação entre instruções, regras e conjuntos de regras. Quando se especificam os conjuntos de regras, cada regra deve ser iniciada pelo seu comprimento, e uma regra nula (de tamanho zero) especifica o fim do conjunto. Os valores em hexadecimal representados na figura (precedidos por x) representam os bytes que compõem as instruções. Como pode ser visto, o número de bytes de cada instrução é variável, dependendo do número e tipo de parâmetros. Uma seqüência de bytes como a mostrada na figura deve ser escrita nos arquivos virtuais de regras.

O injetor pode selecionar uma mensagem pelo seu conteúdo, através do fluxo da mensagem, ou baseando-se em elementos relacionados ao estado do injetor, como temporizadores, contadores e sinalizadores. As instruções de seleção aceitas por ComFIRM são mostradas na Tabela 3.1. É possível testar o conteúdo de bits individuais, bytes, *words* (16 bits) e *double words* (32 bits) presentes nas mensagens. Os modificadores apresentados na Tabela 3.2 devem ser aplicados a estas operações, para determinar a relação entre o parâmetro e o operando. As operações de teste de bit e sinalizadores podem apenas determinar se um bit está ligado ou desligado. As demais operações podem testar se um valor é igual, maior ou menor que o operando.

O teste de bits pode ser usado para verificar sinalizadores de controle específicos dos protocolos. O teste de bytes, *words* e *double words* é interessante quando se deseja testar outros campos do protocolo, como endereços, comandos, números de seqüência e deslocamentos. Para seleção baseada no fluxo de mensagens, o injetor pode testar um valor

Tabela 3.1: Instruções de seleção de ComFIRM

x00	Testar bit	x20	Testar contador
x08	Testar byte	x28	Testar byte aleatório
x10	Testar *word*	x30	Testar temporizador
x18	Testar *double word*	x38	Testar sinalizador

Tabela 3.2: Modificadores de instruções de seleção de ComFIRM

x00	Testar se igual	x00	Testar se desligado
x01	Testar se maior	x01	Testar se ligado
x02	Testar se menor		

de contador ou selecionar aleatoriamente mensagens. ComFIRM fornece 256 contadores de uso geral de 32 bits, que podem ser explicitamente incrementados ou decrementados. Para a seleção aleatória, o injetor pode obter um valor pseudo-aleatório fornecido pelo núcleo. Para seleção de mensagens baseada em elementos externos, temporizadores e sinalizadores podem ser utilizados. Da mesma forma que os contadores, 32 destes sinalizadores (bits individuais) estão disponíveis para a execução de testes. Condições bastante precisas de seleção podem ser especificadas utilizando-se destas operações de seleção.

As instruções de seleção são passivas e por si só podem ser usadas para monitoramento, mas são insuficientes para injeção de falhas ativa. Operações de manipulação, apresentadas na Tabela 3.3, podem ser combinadas às operações anteriores para a definição de cenários de falhas complexos. ComFIRM permite a manipulação de bits, bytes, *words* e *double words* do conteúdo das mensagens, a atuação sobre os elementos externos, como temporizadores, contadores e sinalizadores; bem como atuar sobre a entrega das mensagens, atrasando-as, descartando-as ou duplicando-as. Uma instrução extra imprime o conteúdo de uma mensagem selecionada no registro de eventos (*log*) do injetor de falhas.

De forma semelhante às instruções de seleção, modificadores devem ser aplicados às instruções de manipulação, mostrados na Tabela 3.4. Para operações relacionadas a bits e sinalizadores, os operandos podem ser ligados, desligados ou complementados. Estas ações podem ser usadas para emular falhas do tipo *stuck-at* ou bizantinas. Os valores de bytes, *words*, *double words*, temporizadores e contadores, com os modificadores apropriados, podem receber atribuições, ser incrementados ou decrementados. Operações que descartam, duplicam, atrasam e imprimem mensagens no sistema de registro de eventos não suportam nenhum modificador.

3.2.2 Criando um cenário de falhas

Os conjuntos de regras descritos na Seção 3.2.1 devem mapear uma instância específica do modelo de falhas aceito pela ferramenta. Estas instâncias são o cenário de falhas de uma campanha de injeção de falhas. Por exemplo, o descarte de mensagens pode emular falhas de colapso e omissão. O atraso de mensagens pode emular falhas de temporiza-

Tabela 3.3: Instruções de manipulação de ComFIRM

x40	Modificar bit	x70	Duplicar mensagem
x48	Modificar byte	x78	Modificar contador
x50	Modificar *word*	x80	Modificar temporizador
x58	Modificar *double word*	x88	Modificar sinalizador
x60	Descartar mensagem	x90	Imprimir mensagem no
x68	Atrasar mensagem		registro de eventos

Tabela 3.4: Modificadores de instruções de manipulação de ComFIRM

x00	Atribuir valor	x00	Desligar
x01	Incrementar	x01	Ligar
x02	Decrementar	x02	Complementar

ção, e quando mensagens são duplicadas ou seu conteúdo é modificado, falhas bizantinas podem ser emuladas.

Para ilustrar como regras podem ser criadas para o teste de implementações de protocolos e sistemas distribuídos, aqui é apresentado como ComFIRM pode ser aplicado para testar condições ou eventos específicos de protocolos de comunicação.

Por exemplo, um engenheiro de testes pode estar interessado em observar o comportamento de uma implementação TCP quando pedidos de encerramento de conexão são perdidos. Para isto, uma regra deve ser especificada que, para cada mensagem: (i) verifica se o protocolo do nível de transporte é TCP, valor 6; (ii) verifica se o pacote contém um pedido para o encerramento de conexão (bit FIN ligado e bit ACK desligado) e (iii) descarta a mensagem. Esta regra descartaria todo pedido de fechamento de conexão TCP. O engenheiro de testes pode adicionar um contador à regra, para descartar, por exemplo, somente os 3 primeiros pedidos recebidos. Tal regra faria o incremento, após testar positivamente um pedido de encerramento, de um contador e verificaria se o seu valor é menor que 3, descartando a mensagem nestes casos.

Outra regra pode ser definida para testar como um sistema alvo se comporta quando pacotes IP com bits de tipo de serviço (ToS, *Type of Service*) de valor baixo (no intervalo 0-3) são entregues com um atraso mais alto. Para isto, esta regra deve verificar os bits de prioridade e, se estes possuem um valor de baixa prioridade, introduzir um atraso artificial de, por exemplo, 10 ms. Isto é feito testando-se o bit mais significativo do campo ToS, bit #8, e adicionando um atraso de 10 ms. Adicionalmente, a regra pode incrementar um contador para monitorar quantas vezes esta ação foi realizada. O incremento deve acontecer antes da instrução de atraso, já que ela encerra a avaliação das instruções.

A representação como *bytecode* do conjunto de regras contendo as duas regras descritas acima pode ser vista na Figura 3.1.

3.3 Detalhes de implementação

Como visto na Seção 1.1.3, a intrusividade pode ser considerada sob diferentes aspectos. ComFIRM, apesar de apresentar baixa intrusividade temporal e manipular as mensagens com bastante desempenho, é bastante intrusivo no núcleo Linux relativo à sua alteração. A ferramenta opera através da redefinição de funções chaves do subsistema de rede e no tratador do temporizador periódico. Para o envio, a função interceptada é a `dev_queue_xmit()`, que repassa todos os pacotes do nível de rede para o nível de enlace. Na recepção, a interceptação é feita sobre a função `net_bh()`, o *bottom-half handler*[1] do tratador de interrupção de redes associado ao recebimento dos pacotes de comunicação do nível de enlace (`netif_rx()`). Estas funções são substituídas por *wrappers*, que avaliam as regras configuradas e chamam as funções originais para dar continuidade ao tratamento das mensagens. O tratador periódico dos temporizadores também exige adaptação, já que os temporizadores de ComFIRM devem ser periodicamente atualizados. Estas modificações são feitas nos arquivos fonte originais do núcleo, e portanto a recompilação de todo o núcleo é necessária para instrumentá-lo. Por este motivo, a ferramenta exige atualização constante sempre que uma nova versão do núcleo Linux é disponibilizada.

Outro componente importante de ComFIRM é o processamento dos arquivos virtuais criados no sistema de arquivos `proc`, usados para a entrada de regras, comandos de controle e a leitura dos eventos do injetor de falhas. Para cada um destes arquivos, funções são associadas às primitivas de manipulação `open()`, `close()`, `read()` e `write()`, dando ao operador de testes acesso às funções e estruturas de ComFIRM, internas ao núcleo, usando utilitários comuns de sistema. Essa associação é feita na inicialização do sistema, e portanto os arquivos virtuais de controle são recursos estáticos.

3.4 Limitações de ComFIRM

O fato de ComFIRM não ser modular é a principal limitação da ferramenta. Isto restringe significativamente a sua aplicabilidade, já que para a utilização da ferramenta são necessárias a modificação do código fonte de um núcleo Linux e sua recompilação. Como o núcleo instrumentado é um novo ambiente, a máquina que irá executar os experimentos deve ser reiniciada para sua instrumentação. Conseqüentemente, não há garantias de que este novo núcleo instrumentado se comporte da mesma forma que o núcleo original, já que suas rotinas de comunicação e funções de temporização foram alteradas. Ainda mais complexo é manter a ferramenta atualizada e sempre compatível com a versão mais atual do núcleo Linux, já que isto exige a constante adaptação da ferramenta sempre que uma nova versão é disponibilizada. Pior: modificações na arquitetura interna do Linux ocorridas no desenvolvimento do Linux desde a disponibilização de ComFIRM tornaram a ferramenta incompatível com os núcleos atuais.

Essa limitação é resultado da abordagem de interceptação do subsistema de comunicação utilizada. De fato, entre o desenvolvimento de ComFIRM e FIRMAMENT, o primeiro passo foi atualizar a ferramenta para utilizar métodos de interceptação de código mais eficientes (DREBES et al., 2005), o que serviu de prova de conceito da nova abordagem

[1] ComFIRM foi desenvolvido para a versão 2.2 do núcleo Linux. A arquitetura atual do sistema não utiliza mais o conceito de *bottom-half handlers*.

utilizada para o desenvolvimento posterior de FIRMAMENT.

Outra limitação de ComFIRM está relacionada ao poder de expressão das instruções de operação e manipulação para descrever cenários de falhas. A utilização de operandos fixos dificulta o teste e alteração de campos específicos dos protocolos quando sua localização é variável. Além disso, informações contidas nas mensagens não podem ser diretamente empregadas como parâmetro para instruções subseqüentes. Este problema é descrito mais detalhadamente adiante, na Seção 5.1, bem como a solução aplicada em FIRMAMENT. Foi a existência destas limitações que motivou o projeto e o desenvolvimento de uma nova ferramenta, apresentados nos próximos capítulos.

4 QUESTÕES DE PROJETO DE FIRMAMENT

O injetor de falhas FIRMAMENT (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*) objetiva permitir ao engenheiro de testes a especificação de cenários de falhas de comunicação complexos, com baixa intrusividade, para o teste de protocolos de comunicação e sistemas distribuídos. Para isto, o injetor busca solucionar as limitações associadas à ferramenta ComFIRM. Além de permitir condições elaboradas de especificação de cenários de falhas, a ferramenta permite a manipulação genérica de pacotes de comunicação para outros fins, embora este não seja o foco deste trabalho.

Entre as possíveis aplicações de FIRMAMENT encontram-se a depuração de protocolos e aplicações de rede, onde o uso de implementações reais pode evidenciar deficiências da implementação que possivelmente escapariam à simulação; e o estudo de novos protocolos, já que a possibilidade dada pela ferramenta de colocar o sistema em estados incomuns ou de difícil reprodução facilita o seu entendimento.

FIRMAMENT explora a arquitetura da interface de programação Netfilter (RUSSELL; WELTE, 2002) disponível a partir da versão 2.4 do núcleo Linux. Portanto, FIRMAMENT pode ser carregado em qualquer dispositivo executando versões recentes desse sistema, sem a necessidade de recompilação do núcleo. Utilizando apenas interfaces de programação de alto nível do núcleo, a ferramenta é independente da arquitetura da máquina, podendo ser utilizado em servidores, estações de trabalho ou mesmo dispositivos embarcados que utilizem Linux.

Neste capítulo são apresentadas questões associadas à implementação de injetores de falhas de comunicação. Em seguida, parte-se para considerações relativas à instrumentação de código interno ao núcleo do sistema operacional, e abordagens para a interceptação do fluxo de execução do subsistema de comunicação. O estudo destes fatores está diretamente associado ao projeto e desenvolvimento da ferramenta FIRMAMENT. A apresentação da especificação do injetor desenvolvido a partir destes fatores e sua forma de operação é deixada para o próximo capítulo.

4.1 Interceptação do código de comunicação

FIRMAMENT emprega injeção de falhas por *software* (SWIFI) para a instrumentação do alvo. Esta técnica é mais apropriada para trabalhar com estruturas de dados de mais alto nível do sistema, como pacotes de comunicação. A localização da instrumentação

está associada ao trajeto percorrido pelas mensagens, da aplicação de origem até a de destino. Para a determinação da melhor localização, é necessário um estudo dos protocolos envolvidos em todas as camadas de comunicação e da forma com que são implementados.

Aplicações que utilizam a Internet para troca de dados baseiam-se no modelo de camadas TCP/IP, mais simples que o modelo MR-OSI (TANENBAUM, 2003). A camada inferior desse modelo é o nível físico, onde são especificadas as regras que definem a transmissão de bits brutos através do canal de comunicação e características como níveis de tensão, sinalização e temporização. Sobre ela encontra-se o nível de enlace responsável pela divisão dos dados em quadros. Aqui começam a ser implementadas técnicas de detecção e correção de erros e se efetua o controle de acesso ao meio. A combinação das camadas física e de enlace é chamada de camada de acesso à rede, pois provê serviços à camada superior, o nível de rede. De fato, o modelo TCP/IP não se preocupa em definir as camadas de acesso à rede, sendo estas de livre escolha para os usuários. O nível de rede, como primeiro nível fim-a-fim, é responsável pela conexão de diversas tecnologias de acesso à rede possivelmente distintas, bem como pelo roteamento dos dados sobre as mesmas. Em seguida, encontra-se o nível de transporte, responsável por determinar o tipo de serviço disponível à aplicação (orientado à conexão ou serviço de datagrama), e a multiplexação do nível de rede entre os diversos processos locais de um nó. Finalmente, a camada de aplicação define os protocolos e formatos de dados utilizados pelas aplicações. No modelo TCP/IP, não são encontradas as camadas de sessão e apresentação presentes no modelo MR-OSI. Suas funções costumam ser implementadas diretamente pelas aplicações no modelo TCP/IP.

A localização de um injetor de falhas de comunicação para Linux, portanto, será determinada pelo local de implementação de cada um dos protocolos associados. Aplicações são implementadas como processos de usuário. Logo, para injetar falhas associadas ao nível de aplicação, o método mais apropriado é através da modificação das próprias aplicações. Em alguns casos, existem protocolos intermediários entre a aplicação e o nível de transporte, implementando funções dos níveis de sessão e apresentação, ou *middlewares* com novas funções, como os protocolos RTP (*Real-Time Protocol*), XDR (*eXternal Data Representation*), SIP (*Session Initiation Protocol*), *middlewares* de comunicação de grupo, *grid*, etc. Estes protocolos são implementados através do uso de bibliotecas de ligação dinâmica. Os protocolos de nível de transporte (UDP e TCP), ao contrário dos protocolos acima, são implementados internamente ao núcleo e são utilizados através de chamadas de sistema. Eles, por sua vez, utilizam o protocolo de rede IP, também interno ao núcleo, que utiliza os controladores específicos de dispositivo para ter acesso à camada de enlace. Por fim, o nível físico costuma ser implementado através do *firmware* gravado diretamente nos controladores de rede.

A relação entre as camadas e a implementação dos protocolos associados pode ser observada na Figura 4.1. Neste "modelo ampulheta", o protocolo IP possui posição central de destaque, como único protocolo utilizado pela camada de transporte e por permitir a utilização de diversas tecnologias de acesso distintas. Por essas razões, a injeção de falhas torna-se mais apropriada nesta camada, permitindo a validação de diversos protocolos de nível superior em ambientes de comunicação heterogêneos. O protocolo IP, devido a sua característica (serviço de datagrama, sem retransmissão) simplesmente reflete as limitações dos níveis inferiores. A perda ou atraso de um quadro no nível de enlace implica no mesmo comportamento de um pacote IP, com a vantagem da independência de tecnologia

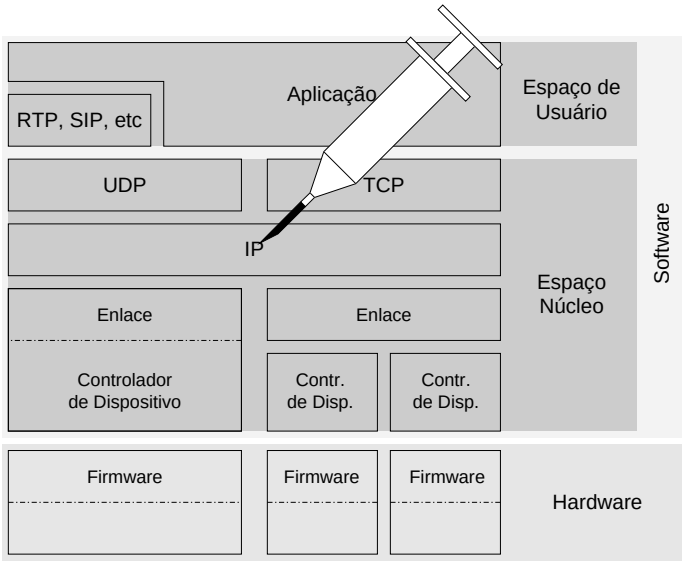

Figura 4.1: Localização da injeção de falhas na pilha de protocolos

de acesso. Ao atuar sobre pacotes IP é possível emular características de falha dos protocolos inferiores, aproximando-se do comportamento de falhas reais nos níveis de acesso à rede.

Sendo a implementação do protocolo IP no sistema Linux interna ao núcleo, qualquer modificação que vise alterar o processamento de pacotes deve ter acesso as suas estruturas internas. Entretanto, a arquitetura de sistemas operacionais monolíticos isola processos de usuário das estruturas de núcleo: todo o acesso às funções do sistema deve ser feito através de chamadas de sistema, e a memória associada ao núcleo é protegida de acessos de processos pela unidade de gerência de memória. Para atuar sobre os pacotes dentro do núcleo, portanto, é necessário executar código interno a ele, o que apresenta alguns desafios.

4.2 Mecanismos para injetores de falhas internos ao núcleo

Injetores de falhas baseados em software podem perturbar o sistema alvo e mascarar suas características temporais. Isso ocorre pois o injetor de falhas altera o código executado inserindo procedimentos que interrompem seu processamento normal. Ao lidar com subsistemas de comunicação de um sistema operacional, estes procedimentos podem ser executados tanto como programas de usuários quanto como componentes do núcleo. Quanto mais próximos estes procedimentos estiverem localizados do hardware, mais eficientes eles tendem a ser. Uma alternativa para a construção de injetores de falhas de baixa intrusividade sobre o alvo é a sua localização no nível do sistema operacional. Esta é a abordagem utilizada pelos injetores ComFIRM e FIRMAMENT.

A localização exata dentro do sistema operacional depende de sua estrutura. O injetor de falhas pode, por exemplo, atuar através das bibliotecas de sistema, chamadas de sistema ou outros recursos executáveis (como interrupções ou rotinas de exceção, controladores de dispositivo e módulos). Sempre que o protocolo alvo requisita serviços

destes recursos do sistema operacional, o injetor de falhas pode agir e modificar o fluxo de execução para injetar falhas.

Se um injetor de falhas está localizado dentro do sistema operacional, a intrusividade, ou efeito de sondagem, no código do protocolo alvo é minimizada. Mesmo que o protocolo alvo não seja perturbado ou alterado diretamente, entretanto, o injetor de falhas pode perturbar a integridade do sistema operacional: a intrusividade passa do protocolo alvo ao sistema operacional. Um injetor de falhas eficiente baseado em núcleo, portanto, deve limitar a forma com que altera o núcleo original, tanto em termos temporais quanto relativo à alteração de código.

A modificação direta do núcleo através de sua reescrita e recompilação é, portanto, uma alternativa possível, mas bastante limitada. Injetores como ComFIRM (LEITE, 2000) que utilizam esta abordagem, apesar de obterem excelentes resultados quanto à intrusividade temporal, tendem a alterar significativamente o núcleo. O novo núcleo pode ser completamente alterado para as atividades de injeção de falhas, o que traz como inconveniente a necessidade de adaptação das modificações a cada nova versão. Além disso, perde-se qualquer garantia de que o núcleo alterado alcança as mesmas especificações do original, e portanto a validade dos resultados de uma validação feita nesse sistema pode ser questionada.

A chamada de sistema para depuração `ptrace()` também pode ser utilizada para a modificação do fluxo de execução normal do núcleo. Apesar de não permitir a adição de código modificado ao núcleo, ela pára a execução de um processo alvo sempre que este executa a entrada e saída de outras chamadas de sistema, permitindo a modificação dos parâmetros de chamada ou valores de retorno. Ao interceptar chamadas associadas à comunicação (recebimento e envio de dados) é possível omití-las ou atrasá-las. Entretanto, essa técnica tem efeito considerável na carga do sistema, pois gera trocas de contexto, entre o processo alvo e um processo concorrente responsável pela injeção de falhas, sempre que uma chamada de sistema do processo alvo é feita. Apesar disto, esta técnica, utilizada dentre outras na ferramenta INFIMO (BARCELOS et al., 2004), não requer a modificação de código de sistema, nem privilégios de administrador para sua operação, o que é necessário em ferramentas que operam diretamente com o núcleo.

Para a adição de novo código ao núcleo Linux, a técnica mais apropriada é a utilização de módulos relocáveis, que podem ser carregados após a inicialização do sistema, como realizado pela ferramenta NIST Net (CARSON; SANTAY, 2003). Entretanto, a simples adição de código ao núcleo sendo executado não implica na sua automática invocação. É necessário que o módulo registre seus recursos no núcleo original para que os mesmos sejam empregados. Para isto, são utilizadas as interfaces de programação do núcleo, utilizadas por controladores de dispositivos e demais subsistemas. A utilização correta destas interfaces garante uma intrusividade mínima quanto à alteração do núcleo. Em seguida são apresentadas duas interfaces que permitem o registro de funções de tratamento em locais específicos do núcleo, consideradas para o desenvolvimento de FIRMAMENT.

4.2.1 Sondagem dinâmica

O mecanismo de sondagem dinâmica (DProbes, *dynamic probes*) (MOORE, 2001) está disponível para o núcleo Linux a partir da sua versão 2.5, através da aplicação de *patches* ao código fonte original. Algumas distribuições, como SUSE Enterprise Server 9, já

vêm com as modificações do mecanismo integradas, mas até o momento a versão oficial estável do núcleo (2.6.12) não incorpora estas modificações por padrão.

Sondas dinâmicas são análogas em software das ponteiras de sondagem, dispositivos de hardware utilizados para a observação e injeção de falhas em sistemas em funcionamento. Sua utilização não é recente. O mecanismo DProbes do Linux descende diretamente do mecanismo DTrace existente no núcleo do sistema operacional OS/2 da IBM. Outros sistemas operacionais da mesma empresa, como z/OS, OS400, AIX e z/VM, também possuem mecanismos de interceptação semelhantes, mas não tão extensíveis e generalizados. O fabricante Sun Microsystems também inclui um mecanismo de sondagem dinâmica em seu sistema operacional Solaris (CANTRILL; SHAPIRO; LEVENTHAL, 2004).

O mecanismo DProbes permite a depuração do software tanto em nível de usuário quanto de sistema, com baixa intrusividade, o que é interessante ao se operar com protocolos de comunicação, que costumam possuir restrições temporais significativas. Além disso, sua arquitetura implica em uma baixa dependência do sistema operacional, podendo ser portado para outras variantes de sistemas operacionais UNIX sem maiores desafios.

DProbes não é uma ferramenta de depuração, mas um habilitador para outras tecnologias de depuração. Seu principal componente é um mecanismo para a interceptação do fluxo de execução, em qualquer localização, seja em nível de usuário ou de sistema, através do registro de pontos de sondagem (*probe points*). A cada um desses pontos, é associado um tratador de sondagem (*probe handler*), que corresponde a uma função com acesso à memória de núcleo, de usuário e aos registradores do processador que é executada quando estes pontos são alcançados.

O registro de pontos de sondagem é semelhante à instalação de um ponto de inspeção em uma ferramenta de depuração. Entretanto, como não é possível parar a execução, já que o código executado pode ser de sistema, somente é possível executar uma sonda prédeterminada, daí o emprego do termo ponto de sondagem. Por não exigir interatividade e restringir as ações a serem realizadas dentro do tratador de sondagem, a sua implementação é feita através de tratadores de interrupção. Isto permite operar dentro de uma tarefa, interrupção ou mesmo troca de contexto, já que a interrupção altera o processador para o modo privilegiado. É esta possibilidade de observação e modificação dos dados em qualquer ponto do código sem intrusão significativa que leva à associação com as ponteiras de sondagem por hardware. Como pode ser visto na Figura 4.2, em cada local onde um ponto de sondagem deve ser inserido, DProbes substitui a instrução de máquina original (a) por uma instrução de interrupção (b). Como tratador desta interrupção, ele executa isoladamente a instrução original. Se a instrução executar sem a ocorrência de exceção, o controle é passado ao tratador de sondagem, retornando ao fluxo original após a sua execução.

A inserção de pontos de sondagem nas funções de envio de pacotes e no tratamento da interrupção do controlador de rede permitiria a construção de um injetor de falhas de comunicação como uma aplicação do mecanismo DProbes, beneficiando-se das vantagens que esse mecanismo apresenta, como baixa intrusividade temporal e uma forma restrita de modificação das estruturas originais do núcleo. Isto motivou um estudo da tecnologia dentro do Grupo de Tolerância a Falhas/UFRGS para o desenvolvimento de uma nova ferramenta para experimentação de falhas de comunicação, NEEDLE (*Network Experimentation Environment using DProbes for Link Errors*) (DREBES; WEBER, 2004). Este

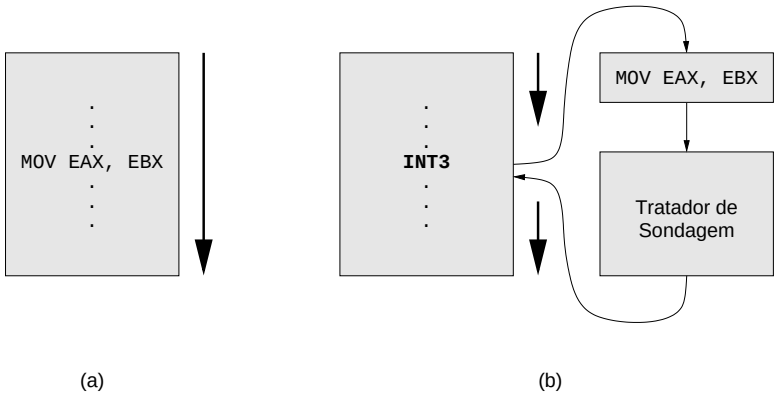

Figura 4.2: Técnica de instrumentação do mecanismo DProbes (IA32)

estudo levou a algumas conclusões importantes, que limitam a viabilidade prática da implementação de tal injetor.

Inicialmente, o registro de um ponto de sondagem exige a localização exata de uma instrução de máquina executada pelo núcleo. Localizar a primeira instrução das funções de comunicação é simples, pois seus endereços estão registrados na tabela de símbolos do núcleo. A localização de pontos internos a estas funções, entretanto, é um desafio. É necessário conhecer como o código fonte da função foi montado em linguagem de máquina, além do tamanho das instruções utilizadas. Somando-se a isso, a localização de um ponto pode ser modificada a cada nova versão do núcleo, pela adição e alteração de outras partes do código, e o conjunto de instruções sobre o qual baseiam-se estes cálculos depende da arquitetura do processador utilizado no sistema.

Considerando-se a superação destas dificuldades, surgem novos problemas, ao se tentar acessar o conteúdo dos pacotes a serem tratados. Ao tratador de sondagem não é fornecido um ponteiro para a estrutura associada ao pacote de comunicação: sua localização deve ser determinada a partir dos parâmetros recebidos pela função interceptada, localizados nos registradores do processador ou na pilha de execução. Finalmente, após o tratamento do pacote não existe forma trivial de recolocar ou alterar o caminho do pacote na pilha de protocolos.

Por estas razões, apesar do mecanismo de sondagem dinâmica possibilitar grande liberdade na localização dos pontos de sondagem, ele limita o desenvolvedor quanto à função executada nestes pontos. O tratador de sondagem, por ser executado dentro de um tratador de interrupção, deve restringir sua atividade para que outras interrupções não sejam perdidas durante o seu processamento.

Injetores de falhas de comunicação não necessitam toda esta liberdade na localização da sonda: é apenas necessário atuar nos pontos de recepção e envio de pacotes. Por outro lado, a função de sondagem pode descrever cenários de falhas com condições complexas, tornando-se extensa, já que sua atividade depende destes cenários especificados pelo usuário. Sendo assim, o mecanismo de sondagem dinâmica, apesar de apropriado para a construção de injetores de falhas genéricos que emulam alterações aleatórias simples no conteúdo de memórias e registradores, é inapropriado para a construção de ferramentas avançadas de injeção de falhas de comunicação. O desenvolvimento da ferramenta

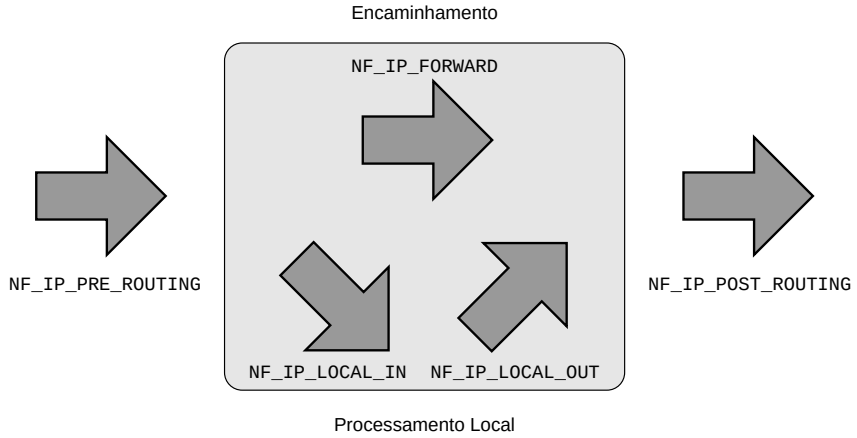

Figura 4.3: Estrutura de ganchos do Netfilter para o protocolo IPv4

NEEDLE foi descontinuado em favor de uma nova ferramenta de injeção de falhas que aproveita uma interface de programação mais facilmente associada às abstrações de comunicação. Esta interface, chamada Netfilter, é apresentada na seqüência.

4.2.2 Netfilter

Netfilter (RUSSELL; WELTE, 2002) é um *framework* de núcleo para a manipulação de pacotes que define pontos específicos, ou ganchos, na pilha de protocolos onde funções de *callback* podem ser registradas. Quando os pacotes alcançam esses pontos, são passados para estas funções, que possuem acesso completo ao seu conteúdo. Elas podem, então, processar os pacotes e decidir se os mesmos devem prosseguir cruzando a pilha de protocolos ou ser descartados. Netfilter permite o registro de funções de *callback* para as famílias de protocolo IPv4, IPv6, DECnet e ARP. Os ganchos estão disponíveis em diversos pontos de interesse: no recebimento e envio de pacotes, no encaminhamento de pacotes roteados, e na entrega e recebimento de pacotes para os níveis superiores. A Figura 4.3 mostra a disponibilidade de ganchos para o protocolo IPv4.

FIRMAMENT é um módulo de núcleo que utiliza os ganchos das famílias de protocolo IPv4 e IPv6. Para pacotes de entrada, o injetor associa funções de *callback* aos ganchos `NF_IP_PRE_ROUTING` e `NF_IP6_PRE_ROUTING`, processando pacotes identificados como IPv4 e IPv6 ainda antes que estes sejam roteados. Para pacotes enviados, funções são registradas nos ganhos `NF_IP_POST_ROUTING` e `NF_IP6_POST_ROUTING`, após os pacotes IP terem sido processados e estarem prontos para ser enviados ao nível de enlace. As funções de processamento interpretam, repetidamente para cada pacote enviado e/ou recebido, micro-programas que expressam o cenário de falhas. Estes programas, chamados *faultlets* e apresentados na Seção 5.1, além de terem capacidade de alterar o conteúdo dos pacotes, retornam à pilha de protocolos qual a ação final a ser realizada sobre o pacote, isto é, descarte, atraso ou processamento normal. A Tabela 4.1 apresenta quais os valores de retorno possíveis para as funções de *callback* do Netfilter, e o tratamento associado ao pacote em cada caso.

As ações do Netfilter utilizadas por FIRMAMENT são `NF_ACCEPT`, `NF_DROP` e `NF_QUEUE`. `NF_ACCEPT` é utilizado quando o processamento de um *faultlet* não se-

Tabela 4.1: Valores de retorno para funções do Netfilter

Valor	**Ação**
NF_ACCEPT	Continua cruzando a pilha de protocolos.
NF_DROP	Descarta pacote; não prossegue na pilha de protocolos. Os recursos utilizados pelo pacote são liberados.
NF_STOLEN	A função captura o pacote, que deve ser desconsiderado pelas demais funções. Os recursos são mantidos.
NF_QUEUE	Enfileira pacote (para tratamento em função a ser registrada pelo desenvolvedor).
NF_REPEAT	Reexecuta esta função (gancho) com o pacote.

leciona o pacote, isto é, ele deve ser processado normalmente, ou quando um pacote é selecionado, mas as instruções a serem executadas sobre ele apenas alteram seu conteúdo ou o estado do injetor de falhas. NF_DROP, como o nome sugere, é empregado quando uma ação de descarte deve ser aplicada ao pacote. A ação NF_QUEUE é um caso especial de descarte, onde o pacote também interrompe seu cruzamento da pilha de protocolos, mas seus recursos alocados não são liberados, e sim passados para uma função previamente registrada para o seu processamento. FIRMAMENT utiliza este mecanismo para o atraso de mensagens. As mensagens são enfileiradas para processamento nesta função, que registra temporizadores a serem chamados após o atraso configurado, reinserindo os pacotes na fila de protocolos para dar continuidade ao seu processamento. Para a duplicação de mensagens, a mensagem é copiada e esta cópia é enviada ou entregue antecipadamente ainda durante o processamento do *faultlet*, sendo utilizado um NF_ACCEPT ao seu término para que o envio ou entrega da mensagem original seja realizado novamente.

O próximo capítulo apresenta a especificação da ferramenta FIRMAMENT e seus blocos básicos. Uma descrição mais completa do conceito de *faultlet* é introduzida, bem como da máquina virtual da ferramenta, FIRMVM, executada no interior das funções de *callback* do Netfilter e responsável pelo processamento dos *faultlets*.

5 A ESPECIFICAÇÃO DE FIRMAMENT

Este capítulo apresenta a especificação de FIRMAMENT, detalhando sua forma de operação: o conceito de *faultlet*, as intruções reconhecidas pela máquina virtual responsável pelo seu processamento, e os comandos de controle da ferramenta. A leitura deste capítulo é o principal ponto de partida para pessoas interessados em realizar experimentos com protocolos e aplicações de rede utilizando FIRMAMENT.

5.1 *Faultlets*

Uma limitação importante das ferramentas de injeção de falhas de comunicação usuais, como ComFIRM, FIONA e INFIMO, é a forma restrita de especificação dos cenários de falhas. As operações a serem efetuadas sobre as mensagens são codificadas de acordo com a forma com que a ferramenta especifica as possíveis ações, de forma bastante rígida e específica para protocolos pré-determinados.

Mesmo quando é possível especificar ações a partir do conteúdo das mensagens, esta especificação costuma ser feita de forma limitada, através do teste de valores a partir de deslocamentos fixos da mensagem. Os dados de um pacote não podem ser lidos e utilizados como parâmetros para as ações do injetor, nem processados de forma combinada para a determinação destas ações.

Um exemplo simples do inconveniente causado por esta limitação é o processamento de cabeçalhos do protocolo IPv4. Este protocolo possui um cabeçalho de tamanho variável. Evidentemente, se o objetivo é examinar sua área de dados não é suficiente simplesmente desconsiderar um número fixo de bytes iniciais, associados ao cabeçalho. É necessário ler, do próprio cabeçalho, seu tamanho, e utilizar esta informação lida como índice para o acesso à área de dados do pacote. Complicando ainda mais o processamento, a indicação do tamanho do cabeçalho IPv4 é dada em número de palavras de 32 bits, portanto para se obter o deslocamento em bytes é necessário multiplicar esta informação por 4.

Este tipo de inconveniente não é exclusivo ao protocolo IPv4. O protocolo IPv6, diferentemente do IPv4, possui um cabeçalho básico de tamanho fixo, mas o cabeçalho do nível de transporte não se encontra sempre imediatamente após este cabeçalho. Por utilizar uma estrutura de cabeçalhos de opções encadeados, o nível de transporte só estará localizado após estes cabeçalhos, caso existam. Seu processamento, portanto, deve ser feito de forma iterativa, verificando-se o tipo do próximo cabeçalho até que este seja um

protocolo de transporte, como UDPv6 ou TCPv6.

Evidentemente, injetores de falhas de comunicação tradicionais podem ser desenvolvidos para considerar estas questões e suportar estes protocolos, através do conhecimento prévio de seu formato. Entretanto, essa abordagem não é extensível a novos protocolos, e limita o engenheiro de testes aos cenários e protocolos pré-suportados pela ferramenta. Mesmo que a ferramenta suporte o processamento dos cabeçalhos dos protocolos IP, o alvo dos testes pode ser um protocolo de aplicação. Apesar de muitos protocolos de aplicação terem suas especificações disponíveis através de RFCs (*Request for Comments*), isto nem sempre ocorre, ou seja, é impraticável desenvolver uma ferramenta de teste de protocolos genérica utilizando as abordagens usuais de especificação de cenários.

FIRMAMENT alcança esta extensibilidade através de uma nova abordagem para a especificação de cenários de falhas, de forma simples porém flexível. A ferramenta utiliza o conceito de *faultlet*, definido neste trabalho. Um *faultlet* é uma aplicação que emula o comportamento de falhas e o diferencial dos *faultlets* em relação a outras abordagens é o poder de expressão desse comportamento.

Um *faultlet* é executado sobre cada pacote que cruza um dos fluxos, podendo inspecionar e modificar o conteúdo do pacote, ou ainda descartá-lo, duplicá-lo ou atrasá-lo. Além do pacote, um *faultlet* trabalha sobre variáveis de estado do fluxo: um conjunto de registradores de uso geral. Ele pode, portanto, mover dados entre o pacote e suas variáveis de estado, nos dois sentidos, e executar operações lógicas, aritméticas e de controle sobre estes dados. Conseqüentemente, o fluxo de execução de um *faultlet* pode ser alterado por dados lidos a partir do pacote, sendo esse o ponto chave da ferramenta: a possibilidade de criação de estruturas avançadas de controle, como laços e desvios, tornando FIRMAMENT apropriado ao teste de qualquer protocolo.

Faultlets são configurados independentemente para os fluxos de entrada e saída[1] do protocolo IP, e sua execução é feita pela máquina virtual FIRMVM. É ela quem efetivamente associa o pacote ao *faultlet* e as variáveis de estado. A próxima seção apresenta a descrição de FIRMVM, listando as instruções e variáveis de estado disponíveis para programação dos *faultlets*. Exemplos de *faultlets* podem ser encontrados no Apêndice B deste volume.

5.2 A máquina virtual FIRMVM

A máquina virtual FIRMVM é o módulo responsável pela interpretação e processamento das instruções que especificam os cenários de falhas. FIRMVM trabalha sobre quatro fluxos de pacotes, associados aos processamentos de entradas e saídas dos protocolos IPv4 e IPv6. A especificação de cenários de falhas para estes fluxos é feita de forma independente, através do fornecimento de *faultlets* para a ferramenta pela interface de arquivos virtuais descrita na Seção 5.5. Esta seção apresenta as variáveis de estado disponíveis para o processamento dos *faultlets* e as instruções disponíveis para a construção dos *faultlets* pelo usuário responsável pela modelagem dos cenários de falhas.

[1] O processamento de *faultlets* diferentes na recepção e no envio dos pacotes de comunicação permite a injeção de falhas associadas a uma topologia de rede assimétrica.

5.2.1 Variáveis de estado

Além do conteúdo dos pacotes, cada fluxo pode operar sobre um conjunto de registradores de uso geral. Os registradores são zerados ao se iniciar um fluxo e seus estados são mantidos durante o processamento seqüencial dos diversos pacotes a eles pertencentes, sendo portanto utilizados como variáveis de estado. A alteração de um registrador não tem efeito sobre a estrutura de mesmo nome de um fluxo distinto.

Um conjunto de 16 registradores de uso geral de 32 bits está disponível para cada fluxo. Sob o ponto de vista do programador, os registradores trabalham com representação de números inteiros[2] com sinal no formato de rede (MSB/BIG_ENDIAN), ou seja, é neste formato que os dados lidos ou escritos dos registradores devem ser considerados. Entretanto, FIRMAMENT faz a conversão automática deste formato para o formato nativo do hardware onde está sendo executado. A especificação para o programador dos registradores na linguagem FIRMASM (descrita mais adiante) é sempre na forma R_x, onde x é um inteiro de 0 a 15.

Cada registrador pode ser configurado para auto-incremento periódico. A cada auto-incremento é associado um período de despertar, em milisegundos, e o registrador é incrementado sempre que este período se esgota. Instruções permitem a ativação e desativação do auto-incremento de forma individual para cada registrador. A sua precisão depende da resolução temporal da interrupção de temporizador da plataforma, sendo tipicamente entre 1 e 10 ms.

5.2.2 Instruções

Faultlets são especificados através de um conjunto de 31 instruções divididas em 7 classes. Estas instruções possuem uma representação textual, chamada FIRMASM, utilizada pelo montador `firm_asm` apresentado na Seção 5.4. O conjunto de instruções e sua representação textual são apresentados abaixo.

5.2.2.1 Instruções de entrada e saída

- **`READB Ry Rx` (Leitura de byte do pacote)**

  Lê em R_x o conteúdo do byte do pacote apontado (em bytes) pelo registrador R_y. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes, o valor de R_x não é alterado.

  $R_x \leftarrow$ (8 bits) pacote[R_y]

- **`READS Ry Rx` (Leitura de *short* do pacote)**

  Lê em R_x o conteúdo do *short* (16 bits) do pacote apontado (em bytes) pelo registrador R_y. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes (menos 1, para dar espaço ao byte restante), o valor de R_x não é alterado.

  $R_x \leftarrow$ (16 bits) pacote[R_y]

[2] FIRMAMENT executa em modo núcleo e o sistema Linux não suporta o uso de aritmética de ponto flutuante neste modo.

- **`READW Ry Rx` (Leitura de *word* do pacote)**

  Lê em R_x o conteúdo do *word* (32 bits) do pacote apontado (em bytes) pelo registrador R_y. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes (menos 3, para dar espaço aos bytes restantes), o valor de R_x não é alterado.

  $R_x \leftarrow$ (32 bits) pacote[R_y]

- **`WRTEB Ry Rx` (Escrita de byte no pacote)**

  Escreve no byte do pacote apontado (em bytes) pelo registrador R_y o valor contido no byte menos significativo do registrador R_x. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes a instrução não tem efeito.

  pacote[R_y] $\leftarrow$ (8 bits) R_x

- **`WRTES Ry Rx` (Escrita de *short* no pacote)**

  Escreve no *short* (16 bits) do pacote apontado (em bytes) pelo registrador R_y o valor contido no short menos significativo do registrador R_x. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes (menos 1, para dar espaço ao byte restante) a instrução não tem efeito.

  pacote[R_y] $\leftarrow$ (16 bits) R_x

- **`WRTEW Ry Rx` (Escrita de *word* no pacote)**

  Escreve no *word* (32 bits) do pacote apontado (em bytes) pelo registrador R_y o valor contido no registrador R_x. Se o conteúdo de R_y é maior que o comprimento do pacote em bytes (menos 3, para dar espaço aos bytes restantes) a instrução não tem efeito.

  pacote[R_y] $\leftarrow$ (32 bits) R_x

- **`SET y Rx` (Escrita de valor em registrador)**

  O registrador R_x recebe o valor absoluto y.

  $R_x \leftarrow$ (32 bits) y

5.2.2.2 Instruções lógicas e aritméticas

- **`ADD Ry Rx` (Soma de registradores)**

  O registrador R_x recebe o valor da soma aritmética dos registradores R_y e R_x.

  $R_x \leftarrow R_x + R_y$

- **`SUB Ry Rx` (Subtração de registradores)**

  O registrador R_x recebe o valor da subtração aritmética dos registradores R_y e R_x.

  $R_x \leftarrow R_x - R_y$

- **`MUL Ry Rx` (Multiplicação de registradores)**

  O registrador R_x recebe o valor da multiplicação aritmética dos registradores R_y e R_x.

  $R_x \leftarrow R_x \times R_y$

- **`DIV Ry Rx` (Divisão de registradores)**

  O registrador R_x recebe o valor da divisão aritmética (inteira) dos registradores R_y e R_x.

  $R_x \leftarrow R_x \div R_y$

- **`AND Ry Rx` (Operação "E" bit a bit de registradores)**

  O registrador R_x recebe o valor da operação lógica bit a bit "E" dos registradores R_y e R_x.

  $R_x \leftarrow R_x \ \& \ R_y$

- **`OR Ry Rx` (Operação "OU" bit a bit de registradores)**

  O registrador R_x recebe o valor da operação lógica bit a bit "OU" dos registradores R_y e R_x.

  $R_x \leftarrow R_x \mid R_y$

- **`NOT Rx` (Inversão de registrador)**

  O registrador R_x recebe o valor da operação lógica de inversão bit-a-bit do registrador R_x.

  $R_x \leftarrow ! \ R_x$

5.2.2.3 Instruções de ação sobre o pacote

- **`ACP` (Aceite de pacote)**

  O pacote é aceito. O processamento do *faultlet* é concluído.

- **`DRP` (Descarte de pacote)**

  O pacote é descartado. O processamento do *faultlet* é concluído.

- **`DUP` (Duplicação de pacote)**

  O pacote é duplicado. O processamento do *faultlet* é concluído.

- **`DLY Rx` (Atraso de pacote)**

  O pacote é atrasado pelo número de milisegundos contido em R_x. O processamento do *faultlet* é concluído.

5.2.2.4 Instruções de desvio de fluxo

- **`JMP x` (Desvio incondicional)**

  O fluxo de execução do *faultlet* é desviado para a instrução localizada na posição x do *faultlet*.

- **`JMPZ Ry x` (Desvio se zero)**

  Se o conteúdo do registrador R_y for zero, o fluxo de execução do *faultlet* é desviado para a instrução localizada na posição x do *faultlet*.

- **`JMPN Ry x` (Desvio se negativo)**

  Se o conteúdo do registrador R_y for negativo, o fluxo de execução do *faultlet* é desviado para a instrução localizada na posição x do *faultlet*.

5.2.2.5 *Instruções de manipulação de auto-incremento*

- **`AION Rx Ry` (Ativação de auto-incremento)**

  Ativa o auto-incremento do registrador R_y, utilizando o conteúdo de R_x como valor periódico de despertar (em milisegundos).

- **`AIOFF Ry` (Desativação de auto-incremento)**

  Desativa o auto-incremento do registrador R_y.

5.2.2.6 *Instruções de manipulação de seqüências de caracteres*

- **`CSTR Ry Rx "string"` (Comparação de seqüência)**

  Compara o conteúdo do pacote a partir da posição contida em R_y com a seqüência de caracteres *string*[3]. O valor '1' é escrito em R_x caso as seqüências de caracteres sejam iguais, '0' caso contrário.

- **`SSTR Ry "string"` (Escrita de seqüência)**

  Escreve a seqüência de caracteres *string* no pacote, a partir da posição contida em R_y, até o fim da seqüência ou do pacote, o que ocorrer primeiro.

5.2.2.7 *Outras instruções*

- **`MOV Ry Rx` (Cópia de registrador)**

  O registrador R_x recebe o valor contido em R_y.

- **`RND Ry Rx` (Sorteio de número pseudo-aleatório)**

  O registrador R_x recebe um valor pseudo-aleatório cujo módulo está entre 0 (zero) e o valor contido em R_y.

  $R_x \leftarrow\ z$, tal que $-R_y < z < +R_y$

- **`SEED Rx Ry Rz` (Semente de número pseudo-aleatório)**

  FIRMAMENT utiliza um algorítmo baseado nos geradores de Tausworthe (1965) para a geração de números pseudo-aleatórios. A semente por padrão é escolhida de forma automática, podendo ser explicitamente definida através desta instrução que a indica através dos três registradores passados como parâmetros (R_x, R_y e R_z).

- **`DBG Rx "string"` (Auxílio de depuração)**

  Exibe, através do sistema de registro de eventos, a seqüência de caracteres *string*, com o conteúdo de R_x substituíndo a primeira ocorrência da seqüência de escape no formato `printf()` (KERNIGHAN; RITCHIE, 1988) (e.g., `%d` para representação decimal, `%x` para representação hexadecimal, etc) da seqüência *string*.

- **`DMP` (Exibição de pacote)**

  Exibe o conteúdo completo do pacote através do sistema de registro de eventos.

[3] O comprimento das seqüências de caracteres é limitado a 255 caracteres.

- **`VER Rx` (Versão da máquina virtual)**

  O registrador R_x é preenchido com o número da versão da máquina virtual FIRMVM em execução. A versão textual, tipicamente expressa na forma $A.B$ (onde A representa a versão principal e B a versão secundária), é armazenada no registrador na forma $(A << 16) + B$.

  $R_x \leftarrow$ versão(FIRMVM)

5.3 Cão-de-guarda da função de processamento

A máquina de registradores FIRMVM é *Turing-completa* e, portanto, *faultlets* nela executada são capazes de colocar o sistema em laço infinito (*loop*), causando um colapso do nó de injeção de falhas. Sendo impossível evitar que isto ocorra, FIRMAMENT implementa mecanismos para evitar o congelamento do processamento de pacotes nestas situações.

Um mecanismo de cão-de-guarda detecta *faultlets* que tomam um tempo elevado para completar, interrompendo sua execução. Neste caso, ao alcançar este limite temporal o pacote é aceito e passado adiante para dar seqüência ao seu processamento pela pilha de protocolos, na forma em que se encontra (isto é, se o *faultlet* altera o pacote, estas alterações são mantidas). Como o conteúdo do pacote pode ser inconsistente, já que depende das ações do *faultlet* antes do instante em que ele é interrompido, testes que disparam o mecanismo de cão-de-guarda não devem ser considerados como válidos e não são de interesse para a experimentação. O objetivo do mecanismo é permitir que o engenheiro de testes possa detectar estas situações e remontar seu experimento de forma a evitá-las.

O intervalo padrão para o mecanismo de cão-de-guarda é de 20 milisegundos, podendo ser alterado (ou mesmo desabilitado) através do comando de controle `settimeout` apresentado na Seção 5.5. A disponibilidade deste recurso, além de oferecer uma proteção extra ao usuário quanto à utilização de *faultlets* mal formados, permite que a ferramenta seja usada em ambientes de tempo real brando (*soft real-time*), já que é possível limitar a intrusividade temporal máxima da ferramenta através de limites superiores para o tempo de injeção de falhas/processamento de pacotes. Nestes casos, é necessário certificar-se de que os *faultlets* empregados não deixem os pacotes em estados inconsistentes, ao serem interrompidos.

5.4 Montador para FIRMASM

O injetor FIRMAMENT não interpreta diretamente a forma textual das instruções apresentadas na Seção 5.2, mas sim *faultlets* representados em um formato binário mais apropriado para processamento interno ao núcleo. Isto exige a tradução, ou montagem, das instruções de sua forma textual para o formato binário.

A ferramenta responsável pela montagem chama-se `firm_asm`. Sua invocação recebe como parâmetro o arquivo contendo o *faultlet* descrito na linguagem FIRMASM a ser montado e, opcionalmente, um parâmetro indicando o destino de salvamento do

arquivo montado, sendo assumido o valor `fa.out` por padrão. Este parâmetro pode indicar diretamente um dos arquivos virtuais de regras existente no diretório `/proc/net/firmament/rules` (descritos na próxima seção), evitando a criação de um arquivo intermediário.

O utilitário de montagem verifica a tipagem de parâmetros das instruções, evitando que *faultlets* mal formados sejam carregados no injetor. Isto permite algumas simplificações na implementação interna ao núcleo do interpretador, diminuindo a intrusividade em tempo de execução dos experimentos.

Outra facilidade da ferramenta é o suporte a rótulos: indicações de posição que podem ser utilizadas como parâmetros nas instruções de desvio. A definição de um rótulo pode conter até 10 caracteres, sendo localizada sempre antes de uma instrução e terminada por ':'. A referência ao rótulo deve ser feita omitindo-se este marcador. Ainda, tanto os rótulos como as intruções e operandos são insensíveis à caixa alta/baixa (*case-insensitive*).

O tratamento de seqüências de caracteres permite a indicação de caracteres não imprimíveis (ou qualquer caractere ASCII[4], pela sua representação nas formas octal ou hexadecimal) através da utilização de seqüências de escape. A listagem dos caracteres especiais suportados pode ser encontrada na Tabela 5.1.

Tabela 5.1: Códigos de escape para seqüência de caracteres

`\a`	alerta	`\t`	tabulação
`\b`	retrocesso	`\v`	tabulação vertical
`\f`	avanço de página	`\\`	contra barra
`\n`	nova linha	`\`*ooo*	número octal
`\r`	retorno de carro	`\x`*hh*	número hexadecimal
`\"`	aspas duplas		

Para a instrução de carga de valor absoluto em registrador (`SET`), o montador aceita números inteiros nas formas hexadecimal ou decimal. Para a representação hexadecimal, o número deve ser precedido por "`0x`" e considerado sem sinal. Para a decimal, tanto números positivos quanto negativos podem ser indicados.

Comentários podem ser adicionados através da utilização do caractere ';' em qualquer ponto da linha de entrada (exceto dentro de variáveis do tipo seqüencia de caracteres, sendo nesse caso considerados parte da seqüencia). Todo texto contido após a ocorrência deste caractere na linha é ignorado.

Uma ferramenta complementar, chamada `msa_mrif`, permite a conversão no sentido contrário, fazendo a "desmontagem" dos *faultlets* a partir de sua representação binária. Esta ferramenta permite a inspeção dos *faultlets* carregados nos arquivos virtuais de regras localizados em `/proc/net/firmament/rules` ou a simples depuração de arquivos montados pelo montador `firm_asm`.

[4]Com exceção do caractere nulo (código 0). Esta é uma limitação do montador e não do interpretador.

5.5 Arquivos virtuais

A operação de FIRMAMENT é feita através da manipulação de arquivos virtuais existentes no diretório `net/firmament` do sistema de arquivos `proc`. Neste diretório encontram-se dois tipos de arquivos. Os arquivos de regras (localizados no subdiretório `rules`) permitem a escrita e leitura dos *faultlets* (em formato binário gerado pelo montador `firm_asm`) para cada um dos fluxos de pacotes. O arquivo `control` permite a passagem de comandos de controle diretamente para o injetor de falhas.

- `/proc/net/firmament/rules/ipv4_in`

  Recebe e armazena o *faultlet* executado a cada vez que um pacote IPv4 é recebido pela máquina.

- `/proc/net/firmament/rules/ipv4_out`

  Recebe e armazena o *faultlet* executado a cada vez que um pacote IPv4 é enviado pela máquina.

- `/proc/net/firmament/rules/ipv6_in`

  Recebe e armazena o *faultlet* executado a cada vez que um pacote IPv6 é recebido pela máquina.

- `/proc/net/firmament/rules/ipv6_out`

  Recebe e armazena o *faultlet* executado a cada vez que um pacote IPv6 é enviado pela máquina.

- `/proc/net/firmament/control`

  Recebe comandos para o controle da ferramenta de injeção de falhas. Comandos podem ser passados à ferramenta através do utilitário `echo`. Por exemplo, para a execução do comando `command`, usa-se:

  ```
  echo "command" > /proc/net/firmament/control
  ```

  Os comandos atualmente suportados pela ferramenta são:

  - `startflow {`*`flow`*`|all}`

    Inicia o processamento de *faultlets* do fluxo *flow* (`ipv4_in`, `ipv4_out`, `ipv6_in` ou `ipv6_out`) ou de todos os fluxos (`all`).

  - `stopflow {`*`flow`*`|all}`

    Interrompe o processamento de *faultlets* do fluxo *flow* ou de todos os fluxos (`all`).

  - `showregister` *`flow register`*

    Exibe, através do sistema de registro de eventos, o conteúdo do registrador *register* do fluxo *flow*.

– `settimeout` *`value`*

  Altera o valor do cão-de-guarda da função de processamento de pacotes para o valor *value*, em milisegundos. Um valor de 0 (zero) desabilita o cão-de-guarda[5].

– `reset`

  Reinicia a ferramenta, parando todos os fluxos, zerando registradores e interrompendo seu auto-incremento, eliminando *faultlets* configurados e configurando o cão-de-guarda para o valor padrão (20 ms).

– `version`

  Exibe, através do sistema de registro de eventos, o número da versão da máquina virtual FIRMVM em execução, na forma $A.B$ (onde A representa a versão principal e B a versão secundária).

– `wdverbose {yes|no}`

  Configura se a exibição dos avisos do mecanismo de cão-de-guarda feita através do sistema de registro de eventos deve ser detalhada (argumento `yes`) ou simplificada (argumento `no`). O detalhamento inclui a exibição do conteúdo dos registradores de um fluxo quando o mecanismo é disparado.

5.6 Sistema de registro de eventos

Uma campanha de injeção de falhas é de utilidade limitada se não existe a possibilidade de se registrar os eventos associados à injeção de falhas e à operação do injetor. Isto é necessário para a posterior análise e relacionamento dos eventos gerados com o reflexo por eles causados no comportamento das aplicações e mecanismos de tolerância a falhas, para que conclusões sobre sua dependabilidade possam ser levantadas.

Como descrito no capítulo anterior, ComFIRM utiliza um arquivo virtual no sistema de arquivos `proc`, para o registro de eventos associados à operação da ferramenta e às atividades de injeção de falhas. Este mecanismo próprio captura os eventos e os formata para exibição quando este arquivo está aberto para leitura por algum utilitário, que torna-se o responsável pelo monitoramento do experimento. Entretanto, a forma com que este mecanismo foi implementada é limitada, já que o número e tamanho dos eventos registrados é fixo, e eventos gerados quando o arquivo de registro está fechado não podem ser posteriormente registrados.

FIRMAMENT utiliza uma abordagem mais simples, empregando o mecanismo padrão de captura de mensagens do sistema `klogd(8)`[6]. Os *buffers* de mensagens do núcleo são lidos do arquivo `/proc/kmsg` (bastante semelhante ao utilizado por ComFIRM) pelo utilitário `klogd`, que as repassa ao utilitário `syslogd(8)`. Ele, por sua vez, é capaz de tanto armazenar estas mensagens localmente em arquivo quanto fazer o registro em máquinas remotas, o que auxilia na condução de experimentos distribuídos. Além disso, estes utilitários suportam priorização de eventos. Mensagens informativas

[5] Esta configuração deve ser usada com cuidado: o usuário assume a reponsabilidade pelo colapso do nó de injeção de falhas causado por um *faultlet* que coloque a máquina virtual FIRMVM em laço infinito.

[6] Seguindo a notação comumente utilizada em sistemas UNIX, o número entre parênteses indica a seção do manual *online* em que a ferramenta é descrita. Para visualizar tal descrição, deve-se, portanto, utilizar o comando `man 8 klogd`.

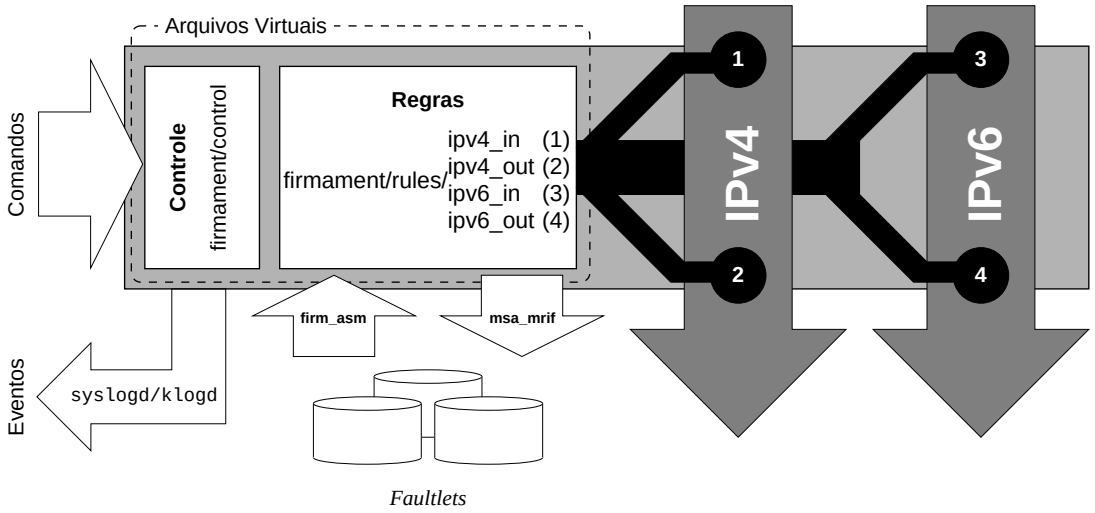

Figura 5.1: Relação conceitual dos elementos do injetor FIRMAMENT

podem ser separadas de mensagens críticas, ou ainda das de depuração, como as criadas pela instrução `DBG` do injetor. É possível capturar apenas mensagens acima de um nível configurado, evitando o registro de mensagens excessivas e menos importantes.

A Figura 5.1 apresenta um esquema da relação conceitual entre arquivos virtuais, comandos e *faultlets*, bem como os locais de atuação do injetor. Os números relacionam os arquivos virtuais de regras aos pontos de interceptação associados. Para a validação desta especificação, uma implementação do injetor de falhas foi construída. O próximo capítulo apresenta testes desta implementação, aplicando-a a protocolos reais e mostrando o funcionamento dos mecanismos aqui descritos.

6 TESTES DE FIRMAMENT

Este capítulo apresenta testes realizados com a ferramenta FIRMAMENT sobre protocolos e aplicações de rede reais, mostrando seu funcionamento de acordo com a especificação apresentada no capítulo anterior e sua viabilidade para experimentação com tais protocolos e aplicações. Estes testes, realizados com exemplos simples, práticos e funcionais, apresentam a capacidade da ferramenta de descartar e atrasar pacotes, alterar seu conteúdo de forma elaborada, bem como o funcionamento do mecanismo de cão-de-guarda da função de processamento dos *faultlets*. Cada teste é acompanhado da listagem comentada do *faultlet* nele utilizado, auxiliando o leitor a ter uma noção mais prática de seu funcionamento do que aquela resultante da simples leitura da listagem de instruções apresentadas no capítulo anterior.

O ambiente de execução dos testes foi composto por máquinas idênticas, com processadores AMD Athlon XP 2000+, 512 MB de memória principal e controladores de rede modelo VIA Rhine II padrão IEEE 802.3u (Ethernet, 100 Mbit/s). Todas as máquinas foram carregadas com o núcleo Linux versão 2.6.12 e interligadas através de um *switch* do mesmo padrão que os controladores de rede, sendo FIRMAMENT carregado apenas em uma das máquinas, onde os *faultlets* foram carregados.

6.1 Intrusividade temporal e atraso de pacotes

Possuir baixa intrusividade é um dos requisitos principais buscados no desenvolvimento de FIRMAMENT. Portanto, o objetivo deste primeiro teste da ferramenta é testar sua intrusividade temporal, já que a intrusividade espacial é resultado de seu projeto e sua forma de implementação, sendo assim mais difícil de ser quantificada.

Este teste mostra a sobrecarga incidente sobre os pacotes processados resultante da instrumentação do fluxo de comunicação e do processamento dos *faultlets*. Nele, foi usado o utilitário `ping(8)`, que calcula o tempo de ida-e-volta de pacotes entre duas máquinas usando o protocolo ICMP (*Internet Control Message Protocol*).

A ferramenta `ping` é bastante simples, e por esta razão muitas vezes não recebe o devido respeito como instrumento de testes, sendo apenas utilizada para experimentação inicial ou superficial de conectividade. Sua grande vantagem para a medição de atrasos de comunicação está na utilização do próprio protocolo ICMP para o cálculo dos tempos de ida-e-volta. Como este protocolo é implementado diretamente dentro do núcleo do sistema operacional, as medidas de tempo não possuem a latência associada à comunicação

entre processos do nível de usuário, dependente do escalonamento.

O mecanismo de medição funciona através da adição de uma opção de marca de tempo (*timestamp*) (BRADEN, 1989) ao pacote ICMP de requisição de eco no momento do envio deste pacote. A resposta à requisição possui uma cópia da requisição, incluindo a sua marca de tempo. Ao receber a resposta, o núcleo da máquina que originou a requisição também adiciona uma opção de marca de tempo a ela, antes de passá-la ao nível de usuário. Chegando no nível de usuário (utilitário `ping`) a resposta possui a marca de tempo efetiva (criada dentro do núcleo) tanto do envio da requisição quanto da recepção da resposta, podendo o cálculo do tempo de ida-e-volta da mensagem ICMP ser feito de forma bastante precisa.

Para a primeira tomada de tempos, o utilitário foi empregado para a medição do atraso de comunicação sem o injetor de falhas. Estes valores servem, portanto, de referência para as tomadas de tempo seguintes, onde o injetor foi utilizado. Em todos os casos, `ping` foi utilizado com a opção "-f" (*flood ping*), que elimina a espera padrão de um segundo entre o envio de requisições e recebimento de respostas. Com ela, além dos tempos de teste serem reduzidos, é possível executar um teste de fadiga e carga de FIRMAMENT, que necessita processar uma taxa elevada de pacotes, na ordem de 9 000 requisições por segundo.

A próxima tomada de tempos buscou medir a intrusividade intrínseca do injetor, causado pela interceptação do fluxo de comunicação. Para isto, utilizou-se um *faultlet* sem ação sobre as mensagens do protocolo ICMP observado. Portanto, a medida é independente da execução de ações do *faultlet*. Este teste indica o efeito do injetor sobre o sistema de comunicação como um todo, isto é, sobre as mensagens que não pertencem ao protocolo ou aplicação alvo de injeção de falhas, ou seja, que não são selecionadas. O *faultlet* configurado testa o protocolo de nível de transporte utilizado e, sendo este UDP, atrasa as mensagens em 20 milisegundos. As demais mensagens, como as do protocolo ICMP utilizadas no `ping`, são processadas normalmente. A intrusividade temporal adicionada está associada apenas à interceptação do pacote por FIRMAMENT e o teste de protocolo executado pelas 5 instruções no início do *faultlet*, apresentadas nas linhas 6 a 11 da Figura 6.1.

A terceira tomada de tempos buscou obter a precisão temporal do atraso criado pelo injetor. Para isto, um *faultlet* semelhante ao utilizado na medição anterior foi usado, mas este selecionando as mensagens do protocolo de interesse, ICMP, atrasando-as em um valor constante de 12 milisegundos, como mostrado na Figura 6.2.

Finalmente, a quarta medida apresenta o resultado da geração de números pseudo-aleatórios de FIRMAMENT, onde o atraso aplicado aos pacotes é variável a partir de um valor médio dentro de limites inferiores e superiores de ocorrência. Neste caso, a partir do valor médio de 12 milisegundos, são calculadas variações entre $+5$ e -5, adicionadas a este valor, como mostrado na Figura 6.3.

Os resultados obtidos nas 4 rodadas de teste são mostrados na Tabela 6.1. Para cada rodada, são apresentados os valores mínimos, médios e máximos de tempo de ida-e-volta, em milisegundos, bem como o desvio padrão. Além disso, são exibidos os tempos totais de execução de cada rodada (em segundos) e a taxa média de requisições respondidas por segundo. Os valores apresentados são médias de 5 medições, sendo em cada uma enviadas 1 000 000 (10^6) requisições. Durante a realização dos experimentos, as máquinas

Atrasa 20 milisegundos cada mensagem UDP

```
1  ; A identificação do protocolo de transporte utilizado
2  ; é contida no 10º byte (índice 9) do cabeçalho IP.
3  ; O valor é lido, subtraído do valor de teste, e a ação
4  ; apropriada (atraso de 20 ms) é tomada caso o valor seja
5  ; igual (o resultado da subtração seja zero).
6          SET    9        R0      ; Deslocamento do pacote a ser lido
7          READB  R0       R1      ; R1 = pacote[R0] (pacote[9])
8          SET    17       R0      ; Protocolo a ser selecionado
9                                  ; (6 para TCP, 17 para UDP...)
10         SUB    R0       R1
11         JMPZ   R1       UDP     ; Se igual (UDP), vai para UDP.
12         ACP                     ; Se diferente, o pacote é aceito.
13 UDP:    SET    20       R0      ; Igual: Atrasa o pacote 20 ms.
14         DLY    R0
```

Figura 6.1: *Faultlet* sem ação sobre as mensagens ICMP

Atrasa 12 milisegundos cada mensagem ICMP

```
1          SET    9        R0      ; Deslocamento do pacote a ser lido
2          READB  R0       R1      ; R1 = pacote[R0] (pacote[9])
3          SET    1        R0      ; Protocolo a ser selecionado: ICMP
4          SUB    R0       R1
5          JMPZ   R1       ICMP    ; Se igual (ICMP), vai para ICMP.
6          ACP                     ; Se diferente, o pacote é aceito.
7  ICMP:
8          SET    12       R0      ; Igual: Atrasa o pacote 12 ms.
9          DLY    R0
```

Figura 6.2: *Faultlet* de atraso fixo de mensagens ICMP

Atrasa 12 $\pm$ 5 milisegundos cada mensagem ICMP

```
1          SET    9        R0      ; Deslocamento do pacote a ser lido
2          READB  R0       R1      ; R1 = pacote[R0] (pacote[9])
3          SET    1        R0      ; Protocolo a ser selecionado: ICMP
4          SUB    R0       R1
5          JMPZ   R1       ICMP    ; Se igual (ICMP), vai para ICMP.
6          ACP                     ; Se diferente, o pacote é aceito.
7  ICMP:
8          SET    12      R0       ; Atraso médio.
9          SET    5       R1       ; Módulo do número aleatório.
10         RND    R1      R2       ; Sorteio: R2 = random(-5, +5);
11         ADD    R2      R0       ; Atrasa pacote entre 7 e 17 ms.
12         DLY    R0
```

Figura 6.3: *Faultlet* de atraso variável de mensagens ICMP

foram configuradas em modo mono usuário, a fim de diminuir a carga de processamento de fundo. A carga de rede correspondente era composta apenas de um tráfego leve de *broadcast* gerado por outras máquinas na mesma subrede, associado aos protocolos ARP e de sistemas de arquivos de rede, sendo seu valor inferior a 1% da banda disponível (100 Mbit/s).

Tabela 6.1: Medidas de tempo de ida-e-volta (em ms, exceto onde indicado)

Rodada	**Mín.**	**Méd.**	**Máx.**	**Desv.**	**Tempo (s)**	**Taxa (r/s)**
Referência	0,073	0,083	0,303	0,009	101,8	9822,4
Injetor sem ação	0,074	0,084	0,302	0,008	102,5	9758,7
Atraso fixo (12 ms)	11,095	11,975	12,213	0,109	11 958,1	83,6
Atraso var. (12 ± 5 ms)	7,102	11,974	16,304	2,452	11 936,9	83,7

Os valores obtidos na primeira e segunda rodada, sem o injetor e com o injetor sem ação sobre os pacotes ICMP, respectivamente, são bastante próximos, sendo as diferenças absolutas nos valores mínimos, médios e máximos mais resultado da imprecisão da medição de intervalos tão pequenos (na ordem de microsegundos) do que da intrusividade temporal efetiva do injetor. A taxa média de requisições por segundo, por outro lado, é medida mais confiável, já que acumula todas as diferenças temporais das requisições processadas. Neste caso, nota-se que a diferença de desempenho no processamento de pacotes registra uma queda de aproximadamente 0,6% na taxa de requisições atendidas com o uso do injetor, valor aceitável dentro da baixa intrusividade esperada.

Quando a ferramenta é configurada para o atraso dos pacotes, o atraso observado fica próximo do valor antecipado (12 ms). Na terceira rodada, de atrasos constantes, os valores mínimos e máximos apresentam-se dentro da margem de erro esperada, mesmo considerando-se os tempos de referência que representam o atraso natural. A margem de 1 milisegundo é conseqüência da precisão temporal do núcleo Linux, que tem este valor. Considerando-se atrasos variáveis, os valores máximos e mínimos também encontram-se na margem de erro aceitável, em relação aos valores de referência de 7 e 17 milisegundos, respectivamente.

Além disso, para a obtenção destes resultados, 15 milhões de requisições, ou cerca de 1,2 GB de dados, foram processados por FIRMAMENT, num período superior a 33 horas, sem que uma única requisição fosse perdida, ou qualquer outra anomalia ou falha não intencional fosse percebida no sistema de suporte, indicando a robustez da ferramenta e a baixa intrusividade espacial esperada.

6.2 Descarte de pacotes

Em seguida, FIRMAMENT foi aplicado a um alvo para a observação de como as falhas de comunicação por ele criadas se manifestam no alvo nos níveis superiores. Aqui, a aplicação utilizada é mais elaborada que o utilitário `ping` empregado na seção anterior, e o objetivo é utilizar o sistema de monitoramento da própria aplicação para detectar a perda de pacotes gerada pelo injetor.

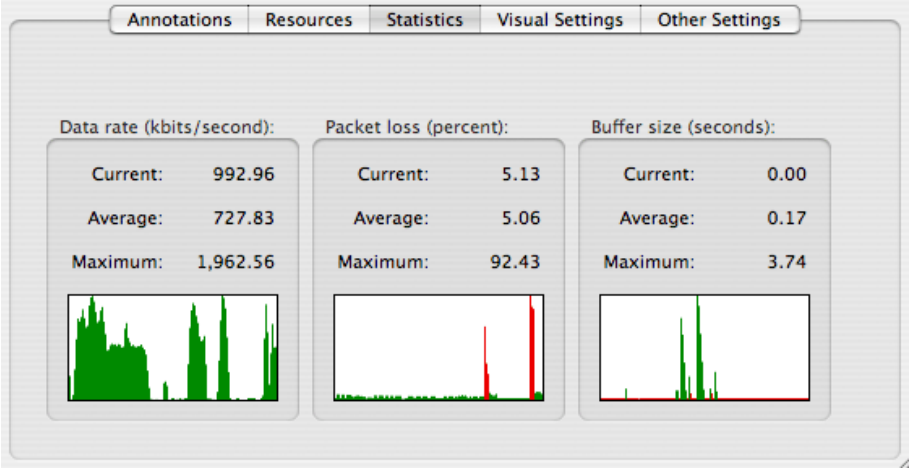

Figura 6.4: Estatísticas do fluxo sob a influência de FIRMAMENT

Figura 6.5: Fluxo de vídeo com erros devido ao descarte

O alvo escolhido é composto por um cliente e um servidor de fluxos (*streams*) de vídeo padrão MPEG-4, que se comunicam utilizando os protocolos RTP (nível de aplicação) e UDP (nível de transporte). O software específico empregado no servidor foi a versão 5.5 do Darwin Streaming Server, e no cliente foi escolhida a versão 7.0.1 do QuickTime Player. O cliente foi executado em uma máquina distinta das descritas no início do capítulo, já que o software QuickTime Player não está disponível para a plataforma Linux.

FIRMAMENT foi configurado com um *faultlet* que, para cada mensagem enviada, testa se o protocolo do nível de transporte é UDP. Caso positivo, *faultlet* verifica se o protocolo de aplicação é RTP, através da porta de origem número 6970 por ele usada. Obtendo sucesso em ambos os testes anteriores, a mensagem é descartada estatisticamente com uma probabilidade de 5%. A listagem do *faultlet* utilizado neste teste encontra-se no Apêndice B deste volume.

A captura de tela da Figura 6.4 apresenta dados estatísticos sobre a sessão de vídeo conduzida no teste. A captura foi feita cerca de 40 minutos após o início da sessão, para que os efeitos transientes fossem desconsiderados, como a carga inicial do *buffer* que o cliente realiza. O dado interessante para este teste é a informação de 'Packet loss' (Perda de pacotes). Os gráficos de barra representam o estado da transmissão durante os últimos 180 segundos. Apesar da perda de pacotes não ser homogênea, apresentando alguns pontos de pico, sua média é muito próxima da taxa configurada de 5%. Isto é o esperado, já que a taxa de perda é estatística, não sendo o descarte determinístico de uma em cada vinte mensagens. Este é um resultado quantitativo, sendo resultados qualitativos demonstrados pela presença de artefatos durante a reprodução do vídeo, como pode ser visto na Figura 6.5.

6.3 Mecanismo de cão-de-guarda

O terceiro teste não tem como objetivo mostrar o funcionamento das instruções de FIRMAMENT, mas sim apresentar o comportamento da ferramenta quando o engenheiro de testes inadvertidamente nela carrega um *faultlet* que coloca a execução da máquina virtual em laço infinito. Nestes casos, um mecanismo de segurança, ou cão-de-guarda, detecta o tempo elevado associado ao processamento do *faultlet*, interrompendo seu processamento e permitindo que o pacote seja entregue.

Um exemplo canônico de uma configuração do injetor cuja execução nunca termina é mostrado na Figura 6.6. O *faultlet* é composto de uma única instrução, que desvia incondicionalmente a execução para sua própria localização.

Laço infinito

```
1 ; Exemplo de faultet mal formado, que nunca termina.
2 ; Em situações normais, uma entrada como esta não seria
3 ; utilizada no injetor, sendo o exemplo apresentado
4 ; apenas para evidenciar o funcionamento do seu mecanismo
5 ; de cão-de-guarda.
6
7 DESVIO: JMP DESVIO           ; Desvia para si próprio.
```

Figura 6.6: *Faultlet* de teste do cão-de-guarda

Para testar o cão-de-guarda, o mesmo utilitário `ping` empregado na Seção 6.1 foi usado. A Figura 6.7 apresenta a saída do utilitário. Nela, é possível observar que os pacotes ICMP levaram aproximadamente 20 milisegundos para ser respondidos, intervalo este que corresponde ao *timeout* padrão de FIRMAMENT. Novamente, há uma imprecisão de cerca de 1 milisegundo associada à precisão temporal do núcleo.

O atraso adicional imposto aos pacotes não deve ser encarado como uma fuga de especificação, já que os casos em que o cão-de-guarda é invocado correspondem a situações de erro e quaisquer experimentos feitos nestas condições devem ser desconsiderados. Esta situação deve ser vista, sim, como uma oportunidade do engenheiro de testes para detectar o problema e reconfigurar o ambiente com *faultlets* corretos, sem a ocorrência do colapso do nó de injeção de falhas e a conseqüente perda de dados.

Não se espera que o condutor dos experimentos detecte o disparo do cão-de-guarda através do aumento do atraso imposto aos pacotes. Isto pode ser feito pela observação do sistema de registro de eventos (*log*) usado por FIRMAMENT, isto é, pelos utilitários `klogd` e `syslogd`. A Figura 6.8 exibe as mensagens registradas a cada invocação do mecanismo, e a partir delas é possível determinar a situação de erro dos *faultlets*. Adicionalmente, através do comando de controle `wdverbose`, é possível exibir o conteúdo dos registradores associados ao fluxo nestas situações para determinar a causa da execução em laço (*loop*) do *faultlet*.

Saída da execução do utilitário ping

```
PING mercedes.inf.ufrgs.br (143.54.12.9) 56(84) bytes of data.
64 bytes from mercedes.inf.ufrgs.br (143.54.12.9): icmp_seq=1 ttl=64 time=19.7 ms
64 bytes from mercedes.inf.ufrgs.br (143.54.12.9): icmp_seq=2 ttl=64 time=20.0 ms
.
.
.
64 bytes from mercedes.inf.ufrgs.br (143.54.12.9): icmp_seq=99 ttl=64 time=19.7 ms
64 bytes from mercedes.inf.ufrgs.br (143.54.12.9): icmp_seq=100 ttl=64 time=19.7 ms
--- mercedes.inf.ufrgs.br ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99084ms
rtt min/avg/max/mdev = 19.115/19.704/21.770/0.374 ms
```

Figura 6.7: Atraso nos pacotes resultante do cão-de-guarda

Saída do *buffer* de mensagens do sistema

```
firm_vm: watchdog called for flow ipv4_in.
firm_vm: watchdog called for flow ipv4_in.
  .
  .
  .
firm_vm: watchdog called for flow ipv4_in.
firm_vm: watchdog called for flow ipv4_in.
```

Figura 6.8: Registro de eventos de invocação do cão-de-guarda

6.4 Falhas bizantinas

O quarto e último teste apresenta a utilização de FIRMAMENT para a emulação de falhas bizantinas, onde o conteúdo de uma mensagem é alterado, resultando em um estado inconsistente entre os nós de um sistema distribuído. Injetores de falhas de comunicação tradicionais, ao modificarem o conteúdo das mensagens, não permitem que as mesmas tenham seu cálculo de verificação (CRC, *cyclic redundancy check*) facilmente recalculado, resultando no descarte das mensagens modificadas. A máquina virtual FIRMVM, por outro lado, permite que o *faultlet* executado atue sobre a mensagem atualizando o resultado do cálculo de verificação após a modificação do pacote. Ao chegar no destino, portanto, o código correto faz a mensagem alterada ser entregue à aplicação alvo, o que não ocorreria se o seu valor anterior fosse mantido. A falha injetada, nestes casos, emula a alteração de um *buffer* de memória contendo os dados a serem enviados e/ou recebidos, já que esta é a única forma de alterações no conteúdo das mensagens chegarem às aplicações sem serem detectadas e descartadas.

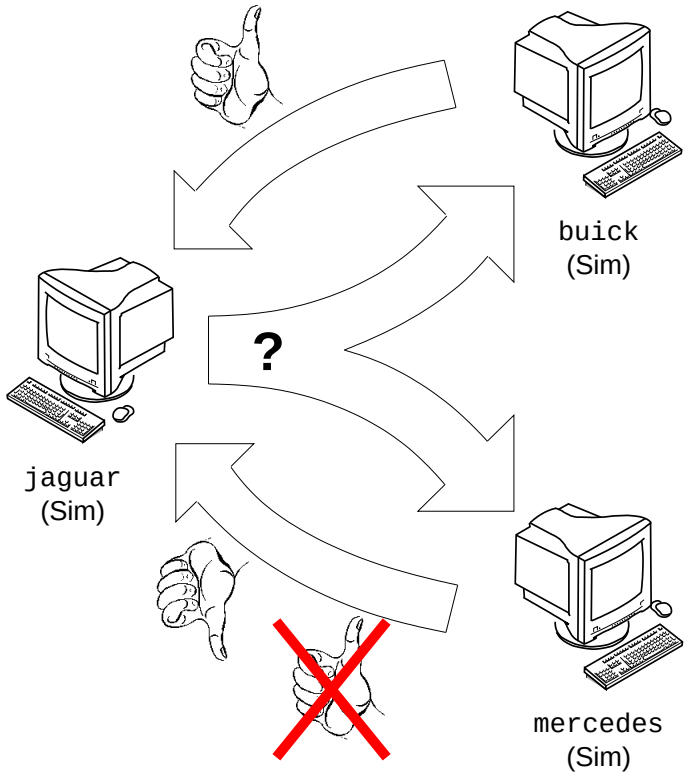

Figura 6.9: Falha bizantina com alteração do conteúdo de uma mensagem

Para a realização do experimento, uma aplicação simples foi desenvolvida, que envia mensagens usando o protocolo UDP. Seu funcionamento está baseado num mecanismo de consultas e respostas, sendo as consultas feitas através de *multicast*, e as respostas em *unicast*. A cada instância, ao ser disparada, é indicado se sua resposta deve ser positiva ("sim") ou negativa ("não") e ela pode, a qualquer momento, enviar uma consulta para as demais instâncias através de um comando do usuário.

A aplicação foi executada em três nós, com configuração idêntica à apresentada no início deste capítulo: `jaguar`, `mercedes`, e `buick`. Todos os nós foram configurados para dar uma resposta sempre positiva ("sim"). Na máquina `mercedes` o injetor foi carregado, configurado com um *faultlet* que capturava as mensagens UDP enviadas e alterava seu conteúdo de "sim" para "não" (e vice-versa) com uma probabilidade de 50%, recalculando o CRC da mensagem nesses casos. A listagem do *faultlet* encontra-se no Apêndice B. Um esquema de funcionamento do experimento é mostrado na Figura 6.9. A Figura 6.10 mostra a saída da aplicação nas três máquinas. Em negrito podem ser vistas as incoerências entre as mensagens enviadas por `mercedes` (antes da injeção de falhas) e as recebidas por `jaguar` (após a injeção), demonstrando que o injetor efetivamente consegue alterar o conteúdo das mensagens e recalcular seu CRC, de forma que a falha injetada não seja percebida pelas verificações de mais baixo nível. Desta maneira, cenários com falhas bizantinas podem ser mais facilmente emulados.

```
jaguar
sou jaguar.inf.ufrgs.br, e respondo "sim"
digite '?' para fazer uma nova consulta
enviando consulta.
mercedes.inf.ufrgs.br respondeu: "sim"
buick.inf.ufrgs.br respondeu: "sim"
digite '?' para fazer uma nova consulta
enviando consulta.
buick.inf.ufrgs.br respondeu: "sim"
mercedes.inf.ufrgs.br respondeu: "não"
digite '?' para fazer uma nova consulta
enviando consulta.
buick.inf.ufrgs.br respondeu: "sim"
mercedes.inf.ufrgs.br respondeu: "não"
digite '?' para fazer uma nova consulta
enviando consulta.
mercedes.inf.ufrgs.br respondeu: "não"
buick.inf.ufrgs.br respondeu: "sim"
digite '?' para fazer uma nova consulta
enviando consulta.
buick.inf.ufrgs.br respondeu: "sim"
mercedes.inf.ufrgs.br respondeu: "sim"
digite '?' para fazer uma nova consulta
```

```
mercedes
sou mercedes.inf.ufrgs.br, e respondo "sim"
digite '?' para fazer uma nova consulta
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
```

```
buick
sou buick.inf.ufrgs.br, e respondo "sim"
digite '?' para fazer uma nova consulta
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
recebi consulta de jaguar.inf.ufrgs.br, respondendo "sim"
```

Figura 6.10: Saída da execução do teste de falhas bizantinas nas 3 máquinas

7 CONSIDERAÇÕES FINAIS

7.1 Conclusões

A execução de testes é um passo essencial na adoção de novos protocolos de comunicação e sistemas distribuídos. A forma com que estes se comportam na presença de falhas, tão comuns em ambientes de comunicação geograficamente distribuídos, dadas sua dispersão geográfica e heterogeneidade, deve ser conhecida e considerada. Condições de falha devem ser criadas e as implementações dos protocolos e sistemas distribuídos devem trabalhar de acordo com sua especificação quando nestas condições, garantindo explicitamente o funcionamento dos seus mecanismos de detecção e recuperação de erros.

Para a realização de tais testes, uma técnica bastante poderosa é a injeção de falhas. Ferramentas de injeção de falhas permitem ao projetista ou engenheiro de testes medir a eficiência dos mecanismos de um sistema antes que o mesmo seja colocado em operação efetiva.

Diversas abordagens para a implementação de injetores de falhas de comunicação existem. Este trabalho apresenta o projeto e desenvolvimento da ferramenta FIRMAMENT, que executa, dentro do núcleo do sistema operacional, micro-programas sobre cada mensagem processada para a emulação de situações de falha de comunicação, utilizando uma abordagem de *scripts*.

O projeto e a implementação da ferramenta alcançam todos os objetivos apresentados na Seção 1.2. A ferramenta é implementada como um módulo de núcleo do sistema operacional Linux, registrando-se na pilha de protocolos, o que lhe dá acesso total aos fluxos de entrada e saída de pacotes, de forma limpa e não intrusiva. Pela sua localização próxima às funções de processamento dos protocolos de transporte, a ferramenta permite o teste de sistemas baseados nos protocolos IPv4 e IPv6.

A utilização da interface Netfilter minimiza a intrusividade da ferramenta relativa ao efeito no desempenho das aplicações ou protocolos a serem validados, bem como à interferência no código do núcleo da plataforma de testes. O desempenho da ferramenta, reflexo de sua baixa intrusividade temporal, é significativo, como mostrado no Capítulo 6. FIRMAMENT evita que os mecanismos da ferramenta sejam invocados nos fluxos que não sejam de interesse aos testes, bem como dispensa a cópia de dados dos pacotes de comunicação a serem inspecionados e manipulados.

A aplicabilidade da ferramenta, dada pela sua facilidade de integração a um ambiente

de produção, é conseqüência da disponibilidade da ferramenta como um módulo de núcleo. Isto permite o carregamento do injetor de falhas como um *plugin* a um núcleo não modificado, o que colabora com a baixa intrusividade espacial da ferramenta.

O alto poder de expressão de cenários de falhas resulta das instruções suportadas pela máquina virtual FIRMVM. As instruções permitem a inspeção e seleção de mensagens de forma determinística ou estatística, além de fornecer diversas ações a serem realizadas sobre os pacotes de comunicação e sobre as variáveis internas do injetor. Isto permite a imitação do comportamento de falhas reais, como descarte e duplicação de mensagens, atraso na sua entrega e modificação de seu conteúdo.

Duas características importantes de FIRMAMENT resultam diretamente de sua forma de implementação. Primeiro, e isto estava entre os objetivos iniciais, a ferramenta é independente de arquitetura. Apesar dos testes apresentados no Capítulo 6 terem sido realizados sobre a arquitetura Intel x86, testes básicos de todas as instruções e mecanismos, em especial aqueles que dependem de detalhes da arquitetura, foram realizados com sucesso na arquitetura PowerPC. Além disso, a ferramenta é extensível, sendo aspectos relacionados à sua extensão apresentados no Apêndice A.

FIRMAMENT é uma evolução sobre ferramentas de injeção de falhas de comunicação construídas anteriormente no Grupo de Tolerância a Falhas/UFRGS. Seu projeto e desenvolvimento aproveita-se de toda experiência resultante da pesquisa anterior, atacando suas principais carências. A ferramenta facilita o trabalho de projetistas de protocolos e aplicações de rede, fornecendo uma plataforma útil, de fácil aprendizado e de instalação simplificada para a execução de experimentos de validação. A ferramenta encontra-se totalmente operacional e pode ser facilmente adicionada a um sistema Linux em execução para a realização de experimentos com protocolos e sistemas distribuídos. Seu código-fonte está disponível na Internet (DREBES, 2005), através de uma licença de software livre.

7.2 Trabalhos futuros

A especificação e implementação de FIRMAMENT já são apropriados para o teste de protocolos de acordo com as características apresentadas na Seção 1.2. Entretanto, questões relacionadas à coordenação distribuída de experimentos de injeção de falhas de comunicação utilizando FIRMAMENT não são consideradas neste trabalho, sendo a ferramenta no momento estritamente de caráter local. Ela é inapropriada, por exemplo, para a emulação de forma direta de falhas de particionamento de rede.

A extensão do injetor para este tipo de experimento envolve a sua integração em um sistema de coordenação de experimentos, como o utilizado por FIONA (JACQUES-SILVA, 2005). Um ponto de especial atenção nesta integração seria o processamento das mensagens de controle do experimento. Em FIONA, a injeção de falhas é feita somente nas mensagens enviadas pela máquina virtual Java. Como a comunicação de controle é feita fora desta, em um nível inferior a máquina virtual, através de interfaces nativas de programação, as mensagens de controle não estão sujeitas às regras de injeção de falhas. FIRMAMENT, ao injetar falhas nas mensagens dentro do núcleo do sistema operacional, atuaria sobre estas mensagens, dificultando a execução dos experimentos coordenados.

Duas soluções para este problema são possíveis, ambas de simples implementação. A primeira é permitir ao engenheiro de testes especificar quais as interfaces de comunicação que estão sujeitas às atividades de injeção de falhas, e utilizar redes distintas para experimentação e coordenação. Isto é trivial utilizando a interface Netfilter, já que cada gancho recebe como parâmetro a interface associada ao pacote capturado. Outra alternativa, que não exige o emprego de redes duplicadas, é a configuração de um *socket* de comunicação de coordenação, através do arquivo virtual `firmament/control`. Um *faultlet* especial, de preâmbulo, seria sempre executado antes dos *faultlets* especificados pelo usuário, e testaria se as mensagens são pertencentes a este *socket* de controle, sempre aceitando-as caso positivo.

Considerando a usabilidade da ferramenta, apesar da linguagem FIRMASM ser poderosa para a especificação de cenários de falhas, ela ainda apresenta alguma dificuldade para novos usuários da ferramenta, em especial os acostumados a especificar seus testes através de descrições ou linguagens de mais alto nível. O desenvolvimento de uma linguagem de programação estruturada, com laços simplificados e auto alocação de registradores, seria uma alternativa para a solução deste inconveniente. *Faultlets* escritos nesta linguagem seriam compilados, resultando nos já suportados pela ferramenta (FIRMASM).

Ainda, para tornar a aplicação da ferramenta mais fácil e transparente, FIRMAMENT poderia ser integrado a ambientes de desenvolvimento beaseado em testes (*test-driven development*) (BECK, 2003). Tal tarefa, entretanto, não é trivial, exigindo um maior estudo de como esta integração seria realizada, adaptando o injetor aos *frameworks* de teste unitário (*unit testing*) disponíveis.

Finalmente, a existência da ferramenta não faz sentido sem a realização de seu intento, o teste de protocolos. Um dos alvos particularmente atraentes para experimentação é o protocolo SCTP (*Stream Control Transmission Protocol*) (STEWART et al., 2000). Este novo protocolo de transporte fornece confiabilidade sobre canais não confiáveis, servindo para aplicações de trocas de mensagens, como controle e sinalização de telefonia. Por ser ainda bastante recente e pouco empregado, suas implementações não possuem a maturidade necessária. A combinação dos recursos de confiabilidade providos com seu ainda curto histórico de uso tornam sua validação de especial interesse.

REFERÊNCIAS

BARCELOS, P. P. de A. et al. A toolkit to test the intrusion of fault injection methods. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 5., 2004, Cartagena, Colombia. **Digest of Papers**. [S.l.: s.n.], 2004. p. 152–157.

BECK, K. **Test-Driven Development: By Example**. Boston: Addison-Wesley, 2003. 240 p. ISBN 0-321-14653-0.

BRADEN, R. (Ed.). **Requirements for Internet Hosts - Communication Layers**: RFC 1122. [S.l.]: Internet Engineering Task Force, Network Working Group, 1989.

CANTRILL, B. M.; SHAPIRO, M. W.; LEVENTHAL, A. H. Dynamic instrumentation of production systems. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2004, Boston, MA. **Proceedings...** [S.l.: s.n.], 2004. p. 15–28.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: Software fault injection and monitoring in processor functional units. In: IFIP WORKING CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, DCCA, 5., 1995, Urbana-Champaign, USA. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1995. p. 135–149.

CARSON, M.; SANTAY, D. NIST Net: a Linux-based network emulation tool. **SIGCOMM Computer Communication Review**, New York, NY, USA, v. 33, n. 3, p. 111–126, 2003.

CLARK, J. A.; PRADHAN, D. K. Fault injection: A method for validating computer-system dependability. **IEEE Computer**, Los Alamitos, CA, v. 28, n. 6, p. 47–56, June 1995.

CRISTIAN, F. Understanding fault-tolerant distributed systems. **Communications of the ACM**, New York, NY, USA, v. 34, n. 2, p. 56–78, Feb. 1991.

DAWSON, S. et al. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING, FTCS, 26., 1996, Sendai, Japan. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1996. p. 404–414.

DEPENDABILITY BENCHMARKING PROJECT. **Preliminary Dependability Benchmark Framework**. Toulouse, France, 2001.

DREBES, R. J. **FIRMAMENT Communication Fault Injector**. 2005. Disponível em: <http://firmament.sourceforge.net/>. Acesso em: out. 2005.

DREBES, R. J. et al. ComFIRM: a communication fault injector for protocol testing and validation. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador, Bahia, Brazil. **Digest of Papers**. [S.l.: s.n.], 2005. p. 115–120.

DREBES, R. J.; WEBER, T. S. Injeção de falhas de comunicação por sondagem dinâmica. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, ERRC, 2., 2004, Canoas. **Anais...** Canoas: SBC, LARC, 2004. p. 29–34.

ECHTLE, K.; LEU, M. The EFA fault injector for fault-tolerant distributed system testing. In: IEEE WORKSHOP ON FAULT-TOLERANT PARALLEL AND DISTRIBUTED SYSTEMS, FTPDS. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1992. p. 28–35.

FALL, K.; VARADHAN, K. (Ed.). **The ns manual**. 2005. Disponível em: <http://www.isi.edu/nsnam/ns/ns-documentation.html>. Acesso em: out. 2005.

HAN, S.; SHIN, K. G.; ROSENBERG, H. DOCTOR: An integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 1995, Erlangen, Germany. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1995. p. 204–213.

JACQUES-SILVA, G. **Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala**. 2005. 79 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

JACQUES-SILVA, G. et al. FIONA: A fault injector for dependability evaluation of Java-based network applications. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS, NCA, 3., 2004, Cambridge, Massachussets, USA. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 2004. p. 303–308. ISBN 0-7695-2242-4.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. 2nd ed. Englewood Cliffs, New Jersey: Prentice Hall PTR, 1988. 272 p. ISBN 0-13-110362-8.

LAPRIE, J.-C. Dependability of computer systems: from concepts to limits. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, DCIA, 1998, Johannesburg, South Africa. **Proceedings...** Johannesburg, South Africa: University of the Witwatersrand, 1998. p. 108–126.

LEITE, F. O. **ComFIRM**: Injeção de falhas de comunicação através da alteração de recursos do sistema operacional. 2000. 117 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

MOORE, R. J. A universal dynamic trace for Linux and other operating systems. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2001, Boston, MA. **Proceedings...** [S.l.: s.n.], 2001.

PARKER, S.; SCHMECHEL, C. **Some Testing Tools for TCP Implementors**: RFC 2398. [S.l.]: Internet Engineering Task Force, Network Working Group, 1998.

PROVOS, N. A virtual honeypot framework. In: USENIX SECURITY SYMPOSIUM, 13., 2004, San Diego, CA USA. **Proceedings...** [S.l.: s.n.], 2004. p. 1–14.

RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. **SIGCOMM Computer Communication Review**, New York, NY, USA, v. 27, n. 1, p. 31–41, 1997.

ROSENBERG, H. A.; SHIN, K. G. Software fault injection and its application in distributed systems. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING, FTCS, 23., 1993, Toulouse, France. **Digest of Papers**. Los Alamitos, CA: IEEE Computer Society Press, 1993. p. 208–217.

RUSSELL, R.; WELTE, H. **Linux netfilter Hacking HOWTO**. 2002. Disponível em: <http://www.netfilter.org/documentation/>. Acesso em: mar. 2005.

STEWART, R. et al. **Stream Control Transmission Protocol**: RFC 2960. [S.l.]: Internet Engineering Task Force, Network Working Group, 2000.

STOTT, D. T. et al. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 4., 2000, Chicago, Illinois, USA. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 2000. p. 91–100.

TANENBAUM, A. S. **Computer Networks**. 4th ed. Upper Saddle River, NJ: Prentice Hall PTR, 2003. 891 p. ISBN 0-13-066102-3.

TAUSWORTHE, R. C. Random numbers generated by linear recurrence modulo two. **Mathematics of Computation**, Providence, RI, v. 19, p. 201–209, 1965.

APÊNDICE A ESTENDENDO FIRMAMENT

FIRMAMENT é extensível. Sua extensão pode ser feita em três domínios: (i) relativa aos protocolos suportados pela ferramenta, (ii) às instruções suportadas pela máquina virtual FIRMVM e (iii) aos comandos de controle e operação da ferramenta.

A extensão a novos protocolos é a mais direta, sendo resultado do emprego da interface Netfilter. Para protocolos suportados pelo Netfilter, sua adição se resume à inclusão de novas entradas em uma estrutura de fluxos e à recompilação do módulo de injeção de falhas. Por exemplo, para adicionar suporte ao protocolo ARP (*Address Resolution Protocol*) basta incluir as seguintes entradas na tabela `f_flowtable[]` do arquivo `firm_vm.h`:

```
1     {"arp_in", NF_ARP_IN, NF_ARP, 0, 0x0806}
2     {"arp_out", NF_ARP_OUT, NF_ARP, 0, 0x0806}
```

Os arquivos `arp_in` e `arp_out` serão criados no diretório `rules`, funcionando da mesma forma que os demais arquivos de regras já existentes.

O suporte a novas instruções exige alterações em apenas 3 pontos do código. Inicialmente, é necessário definir um código ainda não utilizado para a nova instrução. Por exemplo, para adicionar o suporte a uma instrução de "ou exclusivo" (XOR) bit-a-bit entre dois registradores, deve-se adicionar ao arquivo `firm_global.h` o seu código:

```
1     #define F_XOR    0x1D
```

Em seguida, informar seu mnemônico na linguagem FIRMASM, o número de operandos, seu tamanho e o tipo dos seus operandos. A estrutura deve ser adicionada à tabela `instrtable[]` localizada no arquivo `firm_asm.h`:

```
1     {"XOR", F_XOR, 2, 3, 0, {'R', 'R', 0}}
```

Finalmente, a máquina virtual deve receber uma nova entrada para a execução da instrução, na função `f_hook()` do arquivo `firm_vm.c`:

```
1         case F_XOR:
2             idx1 = faultlet[pc + 1];
```

```
3           idx2 = faultlet[pc + 2];
4           /* Invalid register */
5           if (idx1 >= F_MAXREG || idx2 >= F_MAXREG)
6               f_faulterr(flow, "invalid register.");
7           flow->reg[idx2] = flow->reg[idx1] ^ flow->reg[idx2];
8           pc += 3;
9           break;
```

A adição de novas instruções de controle e operação do injetor também é bastante direta. O primeiro passo é a inclusão da entrada relativa ao comando na tabela `firmcmds[]` do arquivo `firm_vm.h`. Por exemplo, para a criação de um comando `clearregister` que limpa o conteúdo de um registrador, a entrada poderia ser especificada como:

```
1     {"clearregister", f_doclreg, 3, "clearregister flow register"}
```

E em seguida, adicionada a função que implementa o comando (2º parâmetro):

```
1  static int f_doclreg(int argc, char *argv[])
2  {
3    struct f_flow *flow;
4    int reg;
5    if (argc > 3) {
6        printk(KERN_WARNING
7               "firm_vm: error, use: clearregister flow register\n");
8        return 0;
9    }
10   flow = f_findflowbyname(argv[1]);
11   if (!flow) {
12       printk(KERN_WARNING  "firm_vm: error, bad flow: %s\n", argv[1]);
13       return 0;
14   }
15   argv[2]++;
16   if (argv[2][-1] == 'r' || argv[2][-1] == 'R') {
17       if (f_isanumber(argv[2], &reg) && (reg > 0)
18                                       && (reg < F_MAXREG)) {
19           flow->reg[reg] = 0;
20           return 1;
21       }
22   }
23   argv[2]--;
24   printk(KERN_WARNING
25               "firm_vm: error, bad register: %s\n", argv[2]);
26   return 0;
27 }
```

Para maiores detalhes sobre a extensão da ferramenta e sobre os campos necessários em cada estrutura, o código fonte deve ser consultado.

APÊNDICE B EXEMPLOS DE *FAULTLET*

Abaixo encontram-se as listagens comentadas dos dois *faultlets* utilizados nos testes das seções 6.2 e 6.4. Apesar de exemplos específicos, algumas de suas estruturas apresentam ações comuns no processamento de protocolos, além de exemplificarem a utilização de certos recursos do montador `firm_asm`, como rótulos, comentários e códigos de escape em seqüências de caracteres.

Descarta 5% das mensagens RTP

```
1
2  ; Este exemplo inicia da mesma forma que os testes ICMP,
3  ; selecionando o protocolo de transporte a partir da informação
4  ; contida no 10º byte (índice 9) do pacote IP.
5
6         SET    9        R0      ; Deslocamento do pacote a ser lido
7         READB  R0       R1      ; R1 = pacote[R0] (pacote[9])
8         SET    17       R0      ; Protocolo a ser selecionado
9                                 ; (6 para TCP, 17 para UDP...)
10        SUB    R0       R1
11        JMPZ   R1       UDP     ; Se igual (UDP), vai para UDP.
12        ACP                     ; Se diferente, o pacote é aceito.
13 UDP:
14
15 ; Neste ponto obtem-se o tamanho do cabeçalho IP (variável)
16 ; para que ele seja desconsiderado e se obtenha a porta de
17 ; origem contida no cabeçalho UDP. O tamanho da cabeçalho IP
18 ; (em palavras de 32 bits) encontra-se nos bits 5 a 8 do pacote,
19 ; por isto a operação de 'E' bit-a-bit com a máscara 0x0f e
20 ; a multiplicação por 4.
21
22        SET    0        R0      ; Inicializa deslocamento
23        READB  R0       R1      ; R1 = pacote[R0] (pacote[0])
24        SET    0x0f     R0      ; R0 recebe máscara (0x0f)
25        AND    R0       R1      ; R1 = R1 & R0 (R1 separa o número
26                                 ; de words do cabeçalho)
27        SET    4        R0      ; R0 recebe bytes por word (4)
28        MUL    R1       R0      ; R0 = R0 * R1 (R0 recebe o número
29                                 ; de bytes do cabeçalho IP)
30
31 ; Aqui, R0 aponta para o primeiro byte do nível
32 ; de transporte (UDP). Para lidar com pacotes UDP todos
```

```
33 ; acessos devem ser feitos a partir desta posição.
34 ; Para determinar a porta de origem do pacote, deve-se
35 ; ler os primeiros 16 bits (short) do pacote UDP. Como
36 ; FIRMAMENT trabalha com o formato de números de rede,
37 ; o número não precisa ser convertido podendo ser diretamente
38 ; comparado.
39
40       READS  R0      R1     ; R1 recebe porta do pacote
41                             ; (udp[R0])
42       SET    6970    R2     ; R2 recebe porta de teste (6970)
43       SUB    R2      R1
44       JMPZ   R1      RTP    ; Se igual (RTP), vai para RTP.
45       ACP                   ; Se diferente, o pacote é aceito.
46 RTP:
47
48 ; O pacote é RTP. Deve ser descartado com probabilidade de 5%.
49 ; O sorteio do número é feito abaixo. Como os números aleatórios
50 ; são calculados em módulo, a idéia é sortear um número entre
51 ; -50 e +50, e verificar se este número é menor que -45. Se
52 ; o número estiver neste intervalo, ao se somar 45 ele
53 ; ainda será negativo, sendo feito o descarte nesse caso. Os
54 ; valores +50, -50 e 45 são multiplicados por 10 pois o
55 ; gerador de números aleatórios é mais justo com intervalos
56 ; maiores, mas o algoritmo é o mesmo.
57
58       SET     500     R0     ; Módulo do número a ser sorteado.
59       RND     R0      R1     ; Sorteio.
60       SET     450     R0     ; Verifica se o número sorteado
61       ADD     R0      R1     ; é menor que -45.
62       JMPN    R1      DROP   ; Caso afirmativo, descarta.
63       ACP                    ; Caso contrário, aceita.
64 DROP:
65     DRP
```

Altera conteúdo de 50% das mensagens UDP (falhas bizantinas)

```
1
2  ; A seleção de mensagens deste exemplo é bastante semelhante a
3  ; do anterior, ficando a diferença na porta UDP selecionada
4  ; (4242). As modificações significativas ficam a partir do
5  ; rótulo PARSE, localizado na linha 44.
6
7         SET    9        R0     ; Deslocamento do pacote a ser lido
8         READB  R0       R1     ; R1 = pacote[R0] (pacote[9])
9         SET    17       R0     ; Protocolo a ser selecionado
10                               ; (6 para TCP, 17 para UDP...)
11        SUB    R0       R1
12        JMPZ   R1       UDP    ; Se igual (UDP), vai para UDP.
13        ACP                     ; Se diferente, o pacote é aceito.
14 UDP:
15        SET    0        R0     ; Inicializa deslocamento
16        READB  R0       R1     ; R1 = pacote[R0] (pacote[0])
17        SET    0x0f     R0     ; R0 recebe máscara (0x0f)
18        AND    R0       R1     ; R1 = R1 & R0 (R1 separa o número
19                               ; de words do cabeçalho)
20        SET    4        R0     ; R0 recebe bytes por word (4)
21        MUL    R1       R0     ; R0 = R0 * R1 (R0 recebe o número
22                               ; de bytes do cabeçalho IP)
23
24        MOV    R0       R15    ; Cópia do início do pacote UDP.
25                               ; Será usado mais tarde.
26
27        READS  R0       R1     ; R1 recebe porta do pacote
28                               ; (udp[R0])
29        SET    4242     R2     ; R2 recebe porta de teste (4242)
30        SUB    R2       R1
31        JMPZ   R1       PARSE  ; Se mesma porta, vai para PARSE.
32        ACP                     ; Se diferente, o pacote é aceito.
33
34 ; No rótulo PARSE o pacote é interpretado e algumas informações
35 ; utilizadas mais tarde no recálculo do checksum são extraídas
36 ; do pacote. Estas informações, e os registradores associados,
37 ; são:
38 ;
39 ; R12: Maior par do tamanho do pacote.
40 ; R13: Padding necessário para checksum do pacote UDP, se
41 ;      seu comprimento tiver tamanho ímpar.
42 ; R14: Tamanho da área de dados do pacote UDP.
43
44 PARSE:
45        SET    2        R0     ; Leitura de tamanho total do
46        READS  R0       R1     ; pacote IP.
47        SET    1        R0     ; Verificação se o tamanho é impar,
48        AND    R1       R0     ; usado para padding no recálculo
49                               ; do checksum UDP. Salva a resposta
50        MOV    R0       R13    ; em R13, para usar mais tarde.
```

```
51
52          SET     0xfffffffe R0  ; Verificação do maior par do
53          AND     R0      R1     ; comprimento do pacote, usada
54          MOV     R1      R12    ; no laço de recálculo de checksum.
55                                 ; Resultado salvo em R12.
56
57          MOV     R15     R0     ; Lê o tamanho do payload UDP,
58          SET     4       R1     ; também, para ser usado no
59          ADD     R1      R0     ; recálculo de checksum. Salva o
60          READS   R0      R1     ; resultado em R14.
61          MOV     R1      R14
62
63
64  ; Sorteia se o pacote deve ser alterado ou não. Como o sorteio
65  ; é feito em módulo, basta selecionar os números negativos
66  ; para termos uma probabilidade de 50%
67
68  SORTEIO:
69          SET     10000   R0     ; Módulo do número a ser sorteado.
70          RND     R0      R1     ; Sorteio.
71          JMPN    R1      ALTERA ; Se negativo, altera pacote.
72          JMP     ACEITA         ; Se positivo, aceita.
73
74  ; O pacote foi selecionado para alteração. Inicialmente é feito
75  ; o salto sobre o cabeçalho UDP, e em seguida verifica-se o
76  ; conteúdo do pacote em busca do string "sim" ou "não",
77  ; substituindo-se o primeiro pelo segundo e vice-versa.
78
79  ALTERA:
80
81          SET     8       R0     ; R1 recebe tamanho de cab. UDP.
82          ADD     R15     R0     ; R0 aponta para o primeiro byte
83                                 ; do nível de aplicação sobre UDP.
84          CSTR    R0      R1  "sim"
85                                 ; Compara conteúdo do pacote com
86                                 ; string "sim".
87          JMPZ    R1      TESTNAO
88                                 ; Se conteúdo era diferente, desvia
89                                 ; para "TESTNAO"
90          SSTR    R0     "n\xe3o"
91                                 ; Conteúdo era igual, substitui por
92                                 ; "não". A representação de "ã" na
93                                 ; codificação ISO8859-1 (caracteres
94                                 ; da Europa ocidental) corresponde
95                                 ; ao valor 0xE3 hexadecimal.
96          JMP     CHECKSUM       ; Pronto, faz recálculo de CRC.
97
98  TESTNAO:
99          CSTR    R0      R1  "n\xe3o"
100                                ; Análogo ao teste anterior, mas
101                                ; substitui "não" por "sim".
```

```
102         JMPZ    R1      ACEITA ; Também era diferente de "não"
103                                ; vai para ACEITA.
104         SSTR    R0      "sim"  ; Substitui "não" por "sim".
105
106 ; Pacote alterado, parte-se para o recálculo do checksum UDP,
107 ; definido como:
108 ;
109 ;    "Checksum is the 16-bit one's complement of the one's
110 ;    complement sum of a pseudo header of information from the
111 ;    IP header, the UDP header, and the data,  padded  with zero
112 ;    octets  at the end (if necessary) to make a multiple of
113 ;    two octets."
114
115 CHECKSUM:
116
117         SET     0       R7     ; Inicialização da soma em R7
118         SET     12      R0     ; Pseudo cabeçalho IP, contendo
119                                ; endereço IP de origem e destino.
120         SET     2       R1     ; Tamanho de incremento do laço.
121
122 LOOPIP:
123         READS   R0      R2     ; Lê short.
124         ADD     R2      R7     ; Acumula na soma.
125         ADD     R1      R0     ; Incrementa índice.
126         SET     20      R3     ; Testa limite.
127         SUB     R0      R3
128         JMPZ    R3      FIMLOOPIP ; Se igual: fim, sai do laço.
129         JMP     LOOPIP
130 FIMLOOPIP:
131         SET     9       R0     ; Acumula na soma o tipo do
132         READB   R0      R2     ; protocolo de transporte.
133         ADD     R2      R7
134         ADD     R14     R7     ; E o seu tamanho.
135         ; Soma do cabeçalho UDP, contendo porta de origem,
136         ; de destino, e comprimento. (bytes 0-5 do cabeçalho).
137         MOV     R15     R0     ; Início do cab. UDP.
138         MOV     R15     R5     ; Configura fim do laço, UDP[0]+6.
139         SET     6       R1
140         ADD     R1      R5
141         SET     2       R1     ; Tamanho do incremento em bytes.
142 LOOPUDP:
143         MOV     R5      R4
144         READS   R0      R2     ; Lê short.
145         ADD     R2      R7     ; Acumula na soma.
146         ADD     R1      R0     ; Incrementa índice.
147         SUB     R0      R4     ; Testa limite.
148         JMPZ    R4      FIMLOOPUDP
149         JMP     LOOPUDP
150 FIMLOOPUDP:
151         ; Parte para o laço dos dados. Mas antes, é necessário
152         ; saltar a posição correspondente ao CRC original, que
```

```
153         ; não está preenchida com zero.
154         SET     2       R1
155         ADD     R1      R0
156 LOOPDADOS:
157         MOV     R12     R9      ; Limite é o maior par do pacote.
158         READS   R0      R2      ; Lê short.
159         ADD     R2      R7      ; Acumula na soma.
160         ADD     R1      R0      ; Incrementa índice.
161         SUB     R0      R9      ; Testa limite.
162         JMPZ    R9      FIMLOOPDADOS
163         JMP     LOOPDADOS
164 FIMLOOPDADOS:
165         JMPZ    R13     CARRY   ; Se o pacote tem tamanho par,
166                                 ; não é necessário padding. Só
167                                 ; adiciona os carries.
168         READB   R0      R2      ; Caso contrário, lê o último
169         SET     256     R1      ; byte e multiplica por 256 (equiv.
170         MUL     R1      R2      ; shift left 8) e acumula o
171         ADD     R2      R7      ; resultado na soma.
172
173 CARRY:
174         SET   0x10000   R2       ; 2^16
175         SET   0xFFFF    R3       ;
176
177 LOOPCAR:
178         MOV     R7      R6      ; aux = soma
179         DIV     R2      R6      ; aux = soma >> 16
180         JMPZ    R6      WRTESUM ;  if (!aux) terminou
181         AND     R3      R7      ; soma = soma & 0xFFFF
182         ADD     R6      R7      ; soma = soma + aux
183         JMP     LOOPCAR
184
185 ; Finalmente, escreve-se o resultado, mas antes tira-se
186 ; o complemento de 1s do valor. O resultado é escrito no
187 ; 4º short (bytes índices 6 e 7) do cabeçalho UDP, apontado
188 ; por R15.
189
190 WRTESUM:
191         MOV     R15     R0      ; Ponteiro para UDP
192         SET     6       R1      ; Posição de escrita.
193         ADD     R1      R0
194
195         SET     0xffff  R1      ; Cálculo do complemento.
196         SUB     R7      R1
197         WRTES   R0      R1      ; Escrita do checksum no pacote.
198
199 ; Aceita o pacote.
200
201 ACEITA:
202         ACP
```

APÊNDICE C HISTÓRICO DE DESENVOLVIMENTO

O interesse do Grupo de Tolerância a Falhas/UFRGS em injetores de falhas de comunicação para avaliação de ambientes distribuídos iniciou na década de 90, associado às Dissertações de Mestrado de Patrícia Pitthan de Araújo Barcelos (*ADC - Ambiente para Experimentação e Avaliação de Protocolos de Difusão Confiável*, 1996) e Irineu Sotoma (*Arquitetura para Injeção de Falhas em Sistemas Distribuídos*, 1997). O envolvimento do autor na área, entretanto, só se deu a partir de 1999, incialmente como bolsista de iniciação científica trabalhando junto à Tese de Doutorado de Patrícia (*INFIMO - Um Toolkit para Experimentos de Intrusão de Injetores de Falhas*, 2001).

INFIMO é um *toolkit* para comparar a intrusividade de métodos de injeção de falhas, e já no seu desenvolvimento ficavam evidentes os desafios a ela associados. Numa das abordagens utilizadas pelo ambiente, a biblioteca de comunicação utilizada pelo alvo era instrumentada, o que resultava numa baixa intrusividade temporal em contrapartida a modificações significativas no ambiente de experimentação e conseqüentemente elevada intrusividade espacial. Além disso, os alvos tinham a restrição de utilizar a biblioteca instrumentada para comunicação. Noutra abordagem, era utilizada a primitiva de depuração `ptrace()` existente nos sistemas Linux, técnica sugerida e empregada no trabalho de Luís Cláudio Gonçalves (*Injetor de Falhas por Depuração*, 2002). Esta abordagem, entretanto, apresentava-se no outro extremo, com uma elevada intrusividade temporal devido às trocas de contexto freqüentes entre o alvo e o injetor de falhas, penalizando o desempenho da aplicação sob teste e tornando-a inapropriada para o teste de aplicações com restrições temporais.

ComFIRM, de Fábio Olivé Leite (*ComFIRM: Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional*, 2000) mostrava-se como a ferramenta com melhor compromisso entre intrusividade espacial e temporal. A localização da ferramenta interna ao núcleo do sistema operacional garantia eficiência na manipulação de pacotes, além de não exigir a modificação da aplicação alvo, nem de suas bibliotecas de suporte, como as usadas para comunicação. Inegavelmente, ComFIRM foi a maior inspiração no desenvolvimento de FIRMAMENT. Entretanto, a ferramenta exigia a sua adaptação para cada nova versão do núcleo do sistema operacional, e a recompilação do mesmo para instrumentação.

Devido a estes desafios, a continuidade do trabalho sobre ComFIRM foi colocada em segundo plano, mas não esquecida. Ao mesmo tempo, o grupo buscou abordagens de mais alto nível para implementação de injetores de falhas de comunicação. Abordagens estas que, apesar de possuirem desvantagens em comparação ao trabalho dentro do núcleo do sistema operacional, permitiram a continuidade dos experimentos de avaliação de

sistemas de comunicação. A ferramenta Jaca, de injeção de falhas genérica para aplicações Java, desenvolvida na UNICAMP, foi estendida para suportar modelos de falhas de comunicação (JACQUES-SILVA; DREBES; WEBER; MARTINS, 2005). Esta extensão foi mais tarde abandonada por apresentar intrusividade espacial significativa sobre a aplicação alvo. Como solução para este problema, partiu-se para o uso de uma interface nativa da máquina virtual Java para o desenvolvimento de ferramentas de monitoramento e depuração, chamada JVMTI, sob a qual é baseado o injetor FIONA (JACQUES-SILVA; DREBES; GERCHMAN; WEBER, 2004)

NEEDLE (DREBES; WEBER, 2004) retornou à tentativa de buscar interceptar o subsistema de comunicação de forma mais limpa e menos intrusiva, mas a abordagem proposta não se mostrou viável. Somente com o emprego da interface Netfilter foi possível atualizar ComFIRM, disponibilizando-o como um módulo de núcleo do sistema operacional Linux (DREBES; LEITE; JACQUES-SILVA; MOBUS; WEBER, 2005).

Ainda assim, a impossibilidade de operar sobre o conteúdo dos pacotes para a avaliação de testes de seleção, a restrição dos testes sempre avaliarem posições fixas do pacote e a obrigatoriedade de conectar os testes em estruturas *se-então* encadeadas, fez necessária a criação de uma nova forma de avaliação e execução de regras, resultando no conceito de *faultlets* e da máquina virtual do injetor, utilizados em FIRMAMENT.

Finalizado o desenvolvimento de FIRMAMENT, a integração das diversas ferramentas de injeção de falhas desenvolvidas em um ambiente comum é um passo natural. FIONA teve sua arquitetura estendida para suportar a condução de experimentos de forma distribuída em ambientes de larga escala, motivada primordialmente pelo interesse do grupo em técnicas de tolerância a falhas e validação em ambientes de computação em *grid*. Suas novas características, como a distribuição de configurações e possibilidade de injeção direta de falhas de particionamento de rede, distingüem FIONA das ferramentas anteriores do grupo, levando a sua integração aos injetores ComFIRM (MOBUS; DREBES; JACQUES-SILVA; WEBER, 2005) e FIRMAMENT. Isto é interessante pois as aplicações alvo de FIONA devem ser desenvolvidas em Java, mas ComFIRM e FIRMAMENT suportam quaisquer aplicações executando sobre Linux, permitindo a experimentação com ambientes heterogêneos. É nesta integração que encontra-se atualmente o trabalho do grupo.

Artigos publicados

BARCELOS, P. P. de A.; DREBES, R. J.; WEBER, T. S. Um toolkit para avaliação da intrusão de métodos de injeção de falhas. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 3., 2002, Búzios. *Anais...* Rio de Janeiro: Núcleo de Computação Eletrônica/UFRJ, 2002. p. 17–24.

BARCELOS, P. P. de A.; DREBES, R. J.; JACQUES-SILVA, G.; WEBER, T. S. A toolkit to test the intrusion of fault injection methods. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 5., 2004, Cartagena, Colombia. *Digest of Papers*. [S.l.], 2004. p. 152–157.

DREBES, R. J. Emulating networks by using dynamic probes for fault injection. In: STUDENT FORUM OF THE INTERNATIONAL CONFERENCE ON DEPENDABLE

SYSTEMS AND NETWORKS, DSN, 2003, San Francisco, California. *Supplemental Volume of DSN 2003*. [S.l.], 2003. p. A37–A39.

DREBES, R. J.; WEBER, T. S. Injeção de falhas de comunicação por sondagem dinâmica. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, ERRC, 2., 2004, Canoas. *Anais...* Canoas: SBC, LARC, 2004. p. 29–34.

DREBES, R. J.; LEITE, F. O.; JACQUES-SILVA, G.; MOBUS, F.; WEBER, T. S. ComFIRM: a communication fault injector for protocol testing and validation. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador, Bahia, Brazil. *Digest of Papers*. [S.l.], 2005. p. 115–120.

GERCHMAN, J.; JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S. Ambiente distribuído de injeção de falhas de comunicação para teste de aplicações Java de rede. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 19., 2005, Uberlândia, MG. *Anais...* Rio de Janeiro: Pontifícia Universidade Católica do Rio de Janeiro, 2005. p. 232–246.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; WEBER, T. S. FIONA: A fault injector for dependability evaluation of Java-based network applications. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS, NCA, 3., 2004, Cambridge, Massachussets, USA. *Proceedings...* Los Alamitos, CA: IEEE Computer Society Press, 2004. p. 303–308.

JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S.; MARTINS, E. Injecting communication faults to experimentally validate Java distributed applications. In: INTERNATIONAL SCHOOL AND SYMPOSIUM ON ADVANCED DISTRIBUTED SYSTEMS, ISSADS, 5., 2005, Guadalajara, Jalisco, México. *Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 2005. v. 3563. p. 235–245.

MOBUS, F.; DREBES, R. J.; JACQUES-SILVA, G.; WEBER, T. S.; JANSCH-PORTO, I. Estendendo FIONA para uma arquitetura híbrida. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 6., 2005, Fortaleza, CE. *Anais...* Fortaleza, CE: II/UFC, 2005. p. 125–128.

Artigo aceito aguardando publicação

DREBES, R. J.; JACQUES-SILVA, G.; TRINDADE, J. M. F.; WEBER, T. S. A kernel-based communication fault injector for dependability testing of distributed systems. In: IBM VERIFICATION CONFERENCE, 2005, Haifa, Israel. *Proceeedings...*

Artigo em avaliação

JACQUES-SILVA, G.; DREBES, R. J.; WEBER, T. S.; MARTINS, E. Integrating network fault scenarios in a dependability evaluation tool to validate Java-based distributed applications. Submetido ao *Journal of Systems and Software*, em dezembro de 2004.