UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

EDUARDO VERRUCK ACKER

Validação de Aplicações para Ambientes Móveis Utilizando Injeção de Falhas

Trabalho de Diplomação.

Profa. Dr. Taisy Silva Weber
Orientador

Prof. Dr. Sérgio Luis Cechin
Co-orientador

Porto Alegre, dezembro de 2009.

AGRADECIMENTOS

Agradeço aos meus orientadores Taisy Weber e Sérgio Cechin pelo apoio, dedicação e simpatia durante a elaboração deste trabalho.

Acima de tudo, gostaria de agradecer aos meus pais e irmãos pelo seu apoio durante esses anos de graduação. Também agradeço o apoio da minha namorada, que a cada dia que passa é mais importante para mim.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

OHA	Open Handset Alliance
PFI	Protocol Fault Injection
LWFI	Injetor de Falha Leve
RTC	Relógio de Tempo Real
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports
SDK	Kit de Desenvolvimento de Software
SNB	State Notification Broker
QoS	Qualidade de Serviço
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ComFIRM	Communication Fault Injection through Operating System Resourse Modification
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
ICMP	Internet Control Message Protocol

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O uso de aparelhos celulares, smartphones e outros semelhantes só tende a crescer nos próximos anos. Em parte esse crescimento se deve a grande variedade de aplicações que são desenvolvidas, na maioria das vezes por usuários comuns. Isso é algo bom, pois disponibiliza uma grande variedade de aplicações, mas na grande maioria dos casos, esses desenvolvedores não se preocupam com a possibilidade de seus aplicativos falharem.

Tendo isso em vista, este trabalho visa o porte de uma ferramenta para a avaliação da cobertura de falhas de comunicação em aplicações desenvolvidas para o sistema operacional móvel Android. Para isso, se fará uso de uma ferramenta de injeção de falhas, a qual fará a injeção das falhas, a monitoração do sistema sob teste e o controle do injetor. Após é feita uma análise dos dados coletados.

Palavras-Chave: tolerância a falhas, injeção de falhas, ambientes móveis, Android.

Validating Applications for Mobile Environments Using Fault Injection

ABSTRACT

The use of cellular devices, smartphones, and other similar ones only tend to grow in the next years. In part this growth happens because of the great variety of applications that are developed, in most cases by common users. This is something good, therefore a great variety of applications is available, but in the great majority of cases, these developers are not worried about the possibility of its applications to fail.

Having this in sight, this work aims the portage of a tool to evaluate the fault tolerance to communication faults of the applications developed for the mobile operational system Android. For this, will be used a tool for fault injection, which will make the injection of faults, monitoring of the system under test and the control of the injector. After, an analysis of the collected data is made.

Keywords: Fault tolerance, fault injection, mobile systems, Android.

1 INTRODUÇÃO

Atualmente com o aumento no uso das redes de comunicação, sejam elas cabeadas ou sem-fio, a quantidade de falhas tende a crescer na mesma proporção. A robustez que antes era exigida somente por sistemas críticos, como aviões, mainframes de alta disponibilidade, e sistemas industriais de controle em tempo-real, agora também é requerida por grande parte da população que adquire serviços e querem eles disponíveis quando requisitados. Tendo isso em vista, é necessário que se desenvolvam técnicas mais avançadas para aumentar a dependabilidade dos sistemas computacionais. Dependabilidade é definida por Avizienis (2004) como sendo a união de diversos atributos: confiabilidade (continuidade do serviço correto), disponibilidade (prontidão para o serviço correto), segurança (ausência de consequências catastróficas para usuário e ambiente), integridade (ausência de alterações impróprias no sistema) e facilidade de manutenção (facilidade de executar reparos e modificações).

Para se alcançar a dependabilidade, tem-se alguns meios que podem ser utilizados (AVIZIENIS, 2004):

- Prevenção de falhas: impede a ocorrência ou introdução a falhas;
- Tolerância a falhas: fornecimento do serviço mesmo na presença de falhas;
- Validação: remoção de falhas, verificação da presença de falhas;
- Previsão de falhas: estimativas sobre a presença e consequências das falhas.

Para se avaliar a eficiência e o desempenho das técnicas utilizadas para detecção e correção de falhas, é necessário que falhas ocorram, contudo elas normalmente ocorrem de forma aleatória e incontrolável e com taxas relativamente baixas. Como é inviável ficar esperando que uma rede específica falhe, e também é inviável economicamente ter-se diversas redes operando em paralelo durante um longo tempo para ver as falhas manifestarem-se voluntariamente, devemos utilizar de técnicas que induzam as falhas. A solução nesses casos é a injeção de falhas.

A injeção de falhas tem por objetivo o teste das soluções de tolerância a falhas empregadas. São injetadas falhas controladas em determinados momentos da execução a fim de se averiguar o quão tolerante a falhas é o sistema e quanto de desempenho essa tolerância vai custar ao sistema. Essa abordagem está bem consolidada na teoria, como pode ser visto em (ARLAT, 1990), (HSUEH, 1997). Existem diversas ferramentas que implementam esta funcionalidade, mas muitas têm restrições quanto ao uso e ao sistema operacional.

Sistemas computacionais móveis estão se expandindo rapidamente. O ambiente móvel traz novos desafios devido a mobilidade dos usuários. As técnicas tradicionais usadas para tolerância a falhas não podem ser empregadas devido a limitação de

recursos do ambiente e do usuário, como a banda limitada na rede wireless, e restrições de energia do usuário. Além disso, ambientes móveis são mais sujeitos a influências climáticas que podem causar falhas na comunicação. Por último, como sistemas móveis tem grande apelo com o consumidor, as falhas devem ser corrigidas com o mínimo de percepção do usuário (KRISHNA, 1993).

O trabalho versará sobre a validação de ambientes móveis através do uso de injeção de falhas visando a tolerância a falhas. O objetivo principal consiste em portar uma ferramenta de injeção de falhas para a avaliação da cobertura de falhas de comunicação em aplicações desenvolvidas para o ambiente Android. Este ambiente foi desenvolvido pela Google, e posteriormente pela Open Handset Alliance. Por ser um sistema operacional para celulares aberto e baseado no kernel Linux versão 2.6, e que aceita o desenvolvimento de aplicações por qualquer desenvolvedor, permitirá uma maior exploração de seus recursos. Em se tratando de um sistema que será incorporado em um grande número de aparelhos, devido às suas funcionalidades, necessita de mais estudos sobre capacidades e/ou limitações das aplicações desenvolvidas para o ambiente.

Este trabalho está organizado em seções. Na seção 2 são apresentados os trabalhos relacionados contemplando conceitos básicos de tolerância a falhas, injeção de falhas, os tipos de falhas que podem ser inseridos pelas ferramentas, e uma breve revisão sobre as ferramentas de injeção de falhas de comunicação Na seção 4 são apresentadas as ferramentas utilizadas para o desenvolvimento do projeto, como foi feita a sua integração, e no final desta descreve-se as dificuldades encontradas nessa etapa. Na seção 5 são apresentados os testes realizados para comprovar a validade da solução proposta e por fim, na seção 6 encontram-se as conclusões finais do estudo.

2 TRABALHOS RELACIONADOS

Falhas são normalmente definidas como o desvio da operação normal do sistema (AVIZIENIS, 2004). A fim de se avaliar a dependabilidade de um sistema, é necessária a utilização de alguma técnica que provoque as falhas que podem acontecer, tendo em vista a falta de controle sobre sua ocorrência e a alta latência para sua manifestação. O método mais comum para se fazer isso, é a utilização de injetores de falhas.

2.1 Injeção de Falhas

O primeiro passo para se avaliar a tolerância a falhas de um sistema é montar um ambiente no qual será feito o experimento. Um ambiente normalmente utilizado é proposto por Hsueh (1997). O ambiente é composto pelo sistema alvo, junto com o injetor de falhas, uma biblioteca de falhas e outra de carga de trabalho, um gerador de carga de trabalho, um monitor, um controlador, coletor e analisador de dados. Na figura 2.1 pode-se ver com esses componentes se interconectam.

O gerador de carga de trabalho alimenta o sistema, enquanto o injetor insere as falhas em determinados momentos. O monitor controla a execução das instruções, iniciando a coleta de dados quando necessário. A coleta é feita online, mas a análise dos dados pode ser feita após o término do programa. O controle do experimento é feito pelo controlador.

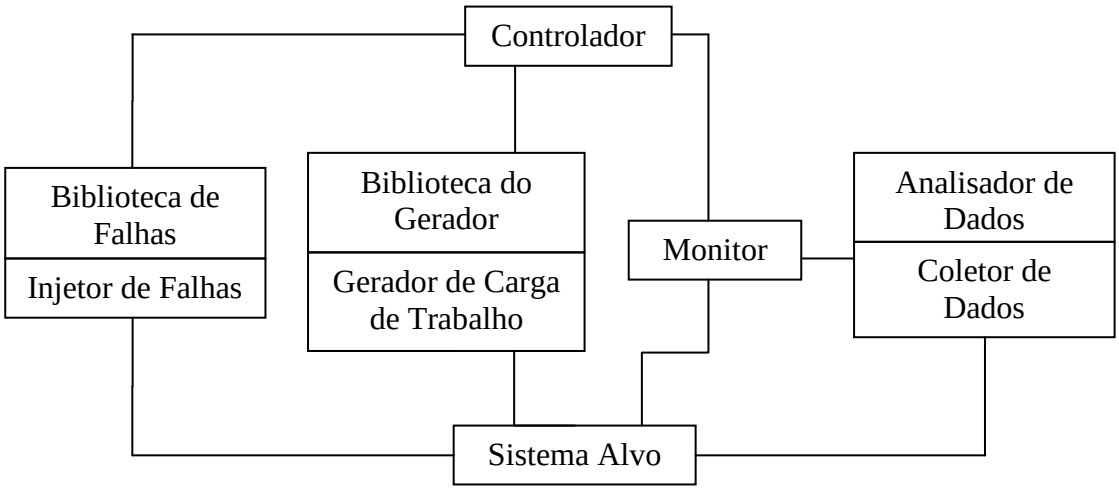

Figura 2.1: Ambiente para injeção de falhas (HSUEH, 1997)

Falhas podem ocorrer tanto no hardware, quanto no software do sistema, para cada caso existem técnicas específicas, com as suas vantagens e desvantagens. Os injetores de falhas de hardware adicionam componentes físicos ao sistema que servem para monitorar e provocar as falhas. Esse método de inserção tem vantagens, pois o injetor e o monitor não utilizam o sistema, assim, não afetando o seu desempenho. Contudo, há um custo alto na inserção de componentes e também se limita a quantidade de pontos de inserção e monitoração de falhas no sistema.

Os injetores por software são na verdade injetores de erros, devido ao nível em que atuam, mas na literatura não é feita essa distinção de nomenclatura para injetores de hardware ou software. A injeção via software é mais fácil de implementar e é mais flexível, mas ela apresenta desvantagens como uma possível interferência na carga de trabalho do sistema e limitação da injeção a locais acessíveis ao software (HSUEH, 1997).

Essa interferência na carga do sistema é conhecida comumente como intrusividade temporal. Ela refere-se ao processamento extra que é adicionado ao sistema devido ao injetor. O processamento pode ser por instruções que são executadas, e pelas trocas de contexto entre o injetor e o sistema que são necessárias.

Ainda segundo Hsueh (1997), existem dois métodos para se inserir a falha. Quando se insere durante a compilação do programa é necessário que se tenha o código fonte para que ele seja alterado e recompilado. Isso é vantajoso, pois não altera o desempenho durante a execução, mas limita o teste a poucas falhas a cada compilação. Por esses detalhes o método citado não é muito utilizado. É mais comum encontrar injetores que fazem a inserção em tempo de execução. Nesse caso é preciso que algum evento pré-determinado ative a injeção da falha, esses eventos também são chamados de ganchos do sistema, entre os quais pode-se citar:

- *Time-out*: ao chegar a zero, um contador provoca uma interrupção que irá ativar a injeção. Recomendável para emular falhas transientes e intermitentes
- *Trap*: uma instrução específica irá ativar o injetor. É necessário que o *trap* esteja conectado ao vetor de interrupções.
- Inserção de código: instruções são adicionadas ao código para ativar a injeção. É diferente da modificação na compilação por que insere instruções, e não as modifica. Difere do *trap*, pois executa em modo usuário, como uma parte do programa alvo.

2.1.1 Falhas de Comunicação

Ao se conectar sistemas computacionais através de redes de comunicação, as suas características distribuídas bem como sua dispersão geográfica trarão preocupações mais sérias quanto à dependabilidade do que se o sistema fosse auto-contido. É possível que ocorra perda de informações, mesmo não havendo falha de nenhum componente do sistema. A fim de diminuir os prejuízos advindos dessas falhas, usam-se técnicas como códigos de detecção e correção de erros, e retransmissão de mensagens.

É necessário se definir os tipos de falhas a que o sistema está sujeito, isso é feito usando-se um modelo de falhas. Em Cristian (1991) foi apresentado um modelo que se

tornou comum para a descrição de sistemas distribuídos. As definições propostas por ele são:

- Omissão: nó não responde a uma mensagem
- Temporização: nó responde corretamente, mas fora do tempo especificado. As mensagens podem ser tanto adiantadas quanto atrasadas
- Comportamento bizantino: respostas diferentes para nodos diferentes, ou alteração do conteúdo da mensagem
- Colapso: servidor para de responder e enviar mensagens

Para se analisar o funcionamento do injetor, um cenário de falhas deve ser descrito. Este cenário vai depender do tipo de falhas que serão inseridas. No caso de falhas de comunicação, o cenário será dependente da forma como as mensagens são selecionadas e manipuladas pela ferramenta (DAWSON et al., 1996). As mensagens que serão manipuladas pelo injetor estarão especificadas no cenário, e a sua seleção será feita por meios determinísticos, ou aleatórios, caso a distribuição de mensagens siga algum padrão definido. A ação a ser executada em determinada mensagem também estará descrita no cenário de falhas.

A emulação de falhas de comunicação pode ser definida basicamente como a interceptação de mensagens trocadas pelos nodos do ambiente de comunicação. Essas trocas ocorrem pelas primitivas *send* e *receive*. Elas são encapsuladas em estruturas que oferecem modelos específicos de comunicação.

2.1.2 Injetores de Falhas de Comunicação

Na literatura existem alguns injetores de falhas de comunicação. Cada um deles apresenta qualidades e restrições que o diferem dos demais. Agora será feita, em ordem cronológica, uma descrição breve de alguns deles, destacando os seus principais aspectos.

2.1.2.1 ORCHESTRA

Proposto por Dawson et al. (1996), o ORCHESTRA é um ambiente de injeção de falhas para o teste de sistemas distribuídos. Desenvolvido para os sistemas operacionais Solaris e Real-Time Mach. A ferramenta introduziu o conceito de sondagem e injeção de falhas orientada a scripts. Isso é importante, pois diminui a intrusividade temporal do injetor.

A ferramenta também objetiva avaliação de protocolos de comunicação, para isso ela precisa fazer a injeção de falhas diretamente na pilha de protocolos. Isso é alcançado utilizando-se uma interface chamada Protocol Fault Injection (PFI), que permite a adição de uma nova camada na pilha de protocolos. Desse modo não é necessária a modificação do código da camada a ser testada.

O ORCHESTRA interpreta os scripts que são escritos na linguagem TCL. Sendo uma linguagem de alto nível, permite a especificação de scripts complexos para a manipulação das mensagens que são interceptadas. A ferramenta também foi testada em protocolos de alto nível, e aplicações baseadas em sockets. As trocas de contexto são evitadas, já que o injetor é executado como uma thread do processo que está sendo avaliado.

2.1.2.2 Dummynet

Feito para o sistema operacional FreeDSB, da família UNIX, o Dummynet (RIZZO, 1997), é uma ferramenta que simula o efeito de filas finitas, limitações de banda e atrasos de comunicação. Ele é inserido entre duas camadas da pilha de protocolos. A alteração dos parâmetros do sistema pode ser feita durante a execução, através de uma chamada de sistema. A baixa interferência no sistema se deve em parte a sua carga, que é feita como um módulo do núcleo.

2.1.2.3 NFTAPE

Framework desenvolvido para avaliar a dependabilidade de sistemas distribuídos, o NFTAPE (STOTT et al., 2000), é uma ferramenta de injeção de falhas que é formada por diversos componentes independentes.

A injeção de falhas, propriamente dita, é feita por pequenos programas responsáveis somente por isso, não se envolvendo com a coleta de dados, esses programas são chamados de Injetores de Falhas Leve (LWFI). Os LWFIs utilizam diversos métodos de inserção de falhas, podem-se citar os mecanismos de depuração, a injeção baseada em drivers do kernel, e a baseada em desempenho, que ataca os recursos do sistema, como a memória, e as interrupções.

Esse sistema modular que foi utilizado na implementação da ferramenta facilita a portabilidade da mesma, mas pode causar alguns problemas para o teste de protocolos de comunicação. Tanto o mecanismo de disparo, quanto o módulo responsável pela injeção, necessitam de acesso ao fluxo de comunicação, assim é mais apropriado o módulo único para ativação e inserção da falha.

2.1.2.4 NIST Net

O NIST Net (CARSON; SANTAY, 2003) é um módulo de núcleo, desse modo pode ser carregado dinamicamente em sistemas Linux que estão executando. Ele apresenta uma grande variedade de falhas que podem ser injetadas, entre elas tem-se: atrasos, reordenamento, perda e duplicação de pacotes, bem como limitações na banda.

O módulo utiliza o controlador do Relógio de Tempo Real (RTC) que está disponível nos computadores. Isso leva a ferramenta a ter uma boa resolução temporal, com uma granularidade na casa dos 120µs. Contudo, caso o controlador do RTC seja compilado no núcleo do sistema, isso irá impedir o uso da ferramenta.

2.1.2.5 VirtualWire

Proposto por De (2003), utiliza uma linguagem de script dirigida a eventos, que permite especificar quais as falhas devem acontecer e em que momento. Essencialmente a ferramenta fornece a abstração de uma rede virtual sobre rede física usada durante o teste. Obteve-se sucesso na implementação, no sentido de se testar implementações de protocolos de redes, gerar falhas de comunicação mais realistas, e facilitar a injeção de falhas.

2.1.2.6 Loki

No Loki (CHANDRA et al., 2004), a inserção de falhas é feita com base numa visão parcial do estado global do sistema distribuído. Uma análise é feita ao final do teste para se deduzir um agendamento global a partir das várias visões parciais, e então se

verificar se as falhas foram corretamente inseridas de acordo com o cenário que foi planejado. Por essa sua característica, o impacto no desempenho do sistema fica reduzido.

Em contrapartida, os cenários de falhas são somente baseados no estado global do sistema. Assim sendo, é muito complicado especificar cenários com falhas mais complexas, como falhas em cascata, sendo necessária a modificação do código fonte da aplicação que será testada. Ele também não fornece suporte a randomização de injeção de falhas.

2.1.2.7 FIRMAMENT

FIRMAMENT (DREBES, 2005) foi desenvolvido no Grupo de Tolerância a Falhas da UFRGS. É uma ferramenta para a avaliação de sistemas distribuídos e de protocolos de communicação. Foi desenvolvida para o Linux, e utiliza a interface de programação do kernel Netfilter (RUSSEL; WELTE, 2002).

A ferramenta também introduz o conceito de faultlet. O faultlet é uma aplicação que emula o comportamento das falhas, atuando também sobre registradores de uso geral que estão ligados a cada um dos fluxos da ferramenta. O FIRMAMENT será descrito em mais detalhes no próximo capítulo.

2.1.2.8 FAIL-FCI

O FAIL-FCI (HOARAU et al., 2007) é composto pelo FAIL, que é uma linguagem de injeção de falhas, e pelo FCI que é uma plataforma de injeção de falhas distribuída. Sendo ambos desenvolvidos como parte do Grid Explorer Project que tem por objetivo a emulação de redes de larga escala em clusters, ou grids menores.

A linguagem FAIL descreve, em alto nível, cenários, que são máquinas de estados que modelam a ocorrência das falhas. Além disso, também é modelada a interação entre essas máquinas de estados e o sistema alvo. A plataforma FCI é composta por alguns blocos: compilador, biblioteca, daemon.

Não é necessário alterar-se o código fonte da aplicação a ser testada, e é possível a injeção de falhas arbitrárias, como a modificação de contadores de programa, ou de varáveis locais. Os daemons para injeção operam tanto num modo randômico ou probabilístico, quanto num modo determinístico. E a ativação das falhas pela depuração do sistema, limita a intrusividade da mesma. Enquanto não se alcança um breakponit, não há influência no sistema.

2.2 Dispositivos Móveis

Atualmente há uma grande gama de dispositivos móveis disponíveis no mercado, e a tendência é de que cada vez mais serviços sejam incorporados neles. Há algum tempo atrás toda a mobilidade disponível era o uso de um notebook, mas depois vieram os PDAs, mas ainda não tinham muito poder computacional. Hoje já temos os chamados Smartphones que integram todos os recursos de computadores, como um sistema operacional, uma grande variedade de aplicativos e o acesso a internet, e mais a comodidade de um celular.

Há alguns sistemas operacionais móveis disponíveis para smartphones. Na tabela 2.1 pode se ver uma comparação entre esses sistemas operacionais (ADMOB, 2009). A participação de mercado de cada um deles no primeiro trimestre de 2009 está na tabela 2.1.a, e a participação estimada nos acessos à Internet que são feitos via celular está na tabela 2.1.b.

Tabela 2.1: Participação no mercado e no uso da web, por sistema operacional

(a)

Sistema Operacional	Mercado (%)
Symbian	52
RIM	17
Windows	12
iPhone	8
Palm	2
Android	1
Outros	9

(b)

Sistema Operacional	Web (%)
iPhone	43
Symbian	36
RIM	9
Windows	5
Android	3
Palm	2
Outros	2

Fonte: ADMOB, abril 2009

2.2.1 O Sistema Android

Desenvolvido por um grupo de diversas empresas, o Android é uma plataforma para dispositivos móveis. Todos os softwares necessários para o funcionamento do aparelho estão integrados nessa plataforma, e outras aplicações desenvolvidas por terceiros podem ser adicionadas.

O código fonte do sistema está disponível no site ANDOIRD-SOURCE. Para facilitar o desenvolvimento das novas aplicações, também se disponibilizou um kit de desenvolvimento de software (SDK). No próximo capítulo serão apresentados mais detalhes sobre seu funcionamento e sua arquitetura.

2.2.2 Injeção de falhas em ambientes móveis

Apesar de ser um mercado que tende a crescer, não foram encontradas muitas ferramentas de injeção de falhas que sejam utilizadas exclusivamente em ambientes móveis. A ferramenta encontrada é o mCrash.

2.2.2.1 mCrash

Desenvolvido para o sistema operacional Windows Mobile 5, o mCrash (RIBEIRO, 2008) é uma ferramenta que utiliza o State Notifications Broker (SNB) para aumentar dependabilidade de aplicativos móveis. O SNB é uma interface de programação que será usada para a monitoração da propagação de erros.

A ferramenta permite testes automáticos para classes, métodos, parâmetros e objetos do framework .NET. Dinamicamente gera scripts de teste, compila-os para o .NET, e invoca o processo de teste.

Utiliza quatro módulos: base de dados para falhas, geração de entradas e módulo de injeção de falhas, analisador de estados, e controle de execução.

3 ESPECIFICAÇÃO DO PROJETO

Nesta seção encontra-se uma breve revisão dos objetivos do estudo contemplando relevância e pertinência dos mesmos e apresentam-se detalhes do projeto e a maneira pela foi desenvolvido os testes.

3.1 Revisão dos Objetivos

A utilização de celulares e smartphones tende a crescer nos próximos anos, assim como as funcionalidades providas por esses aparelhos. Entre as funcionalidades está o acesso a internet, que é usada por muitas das aplicações disponíveis para esses dispositivos.

Apesar desses dispositivos móveis não serem construídos com componentes que forneçam alta disponibilidade de serviço, os usuários desejam que eles estejam disponíveis quando necessário. Um método eficiente para se elevar a qualidade de serviço (QoS) do software e do hardware do aparelho é a utilização de técnicas de tolerância a falhas. Diante disso, para se validar essas técnicas utiliza-se a injeção de falhas.

O objetivo do trabalho proposto consiste no porte de uma ferramenta de injeção de falhas para a avaliação da cobertura de falhas de comunicação de aplicações desenvolvidas para dispositivos móveis. A ferramenta que será portada foi proposta por Drebes (2005), ela disponibiliza diversos métodos para a injeção de falhas, controle do experimento e análise dos dados obtidos. As suas características estão descritas no decorrer do capítulo, bem como o ambiente que será alvo do experimento.

3.2 Netfilter

O Netfilter (RUSSEL; WELTE, 2002) é um *framework* que fornece uma série de ganchos dentro do kernel do Linux para a interceptação e manipulação de pacotes de internet. O Netfilter está presente a partir da versão 2.4 do kernel Linux.

Todas as versões do Linux posteriores a essa possuem o Netfilter compilado no kernel. E sendo independente do hardware, essa interface pode estar disponível até mesmo em sistemas embarcados que utilizem o kernel Linux.

Os protocolos suportados pelo Netfilter são o IPv4, IPv6, e DECnet. Os ganchos disponibilizados podem ser vistos na Tabela 3.1, e na Figura 3.1 tem-se uma representação dos mesmos. Na tabela e na figura os ganchos são referentes ao protocolo IPv4. Para os outros protocolos existem pequenas modificações nos nomes dos ganchos.

Tabela 3.1: Ganchos disponibilizados pelo Netfilter

Gancho	Chamado
NF_IP_PRE_ROUTING	Antes do roteamento dos pacotes
NF_IP_LOCAL_IN	Após roteamento, caso pacote seja para este host
NF_IP_FORWARD	Caso pacote seja destinado a outra interface
NF_IP_LOCAL_OUT	Pacotes de processos locais que estejam saindo
NF_IP_POST_ROUTING	Após serem processados e estarem prontos a serem enviados

Fonte: RUSSEL; WELTE, 2002

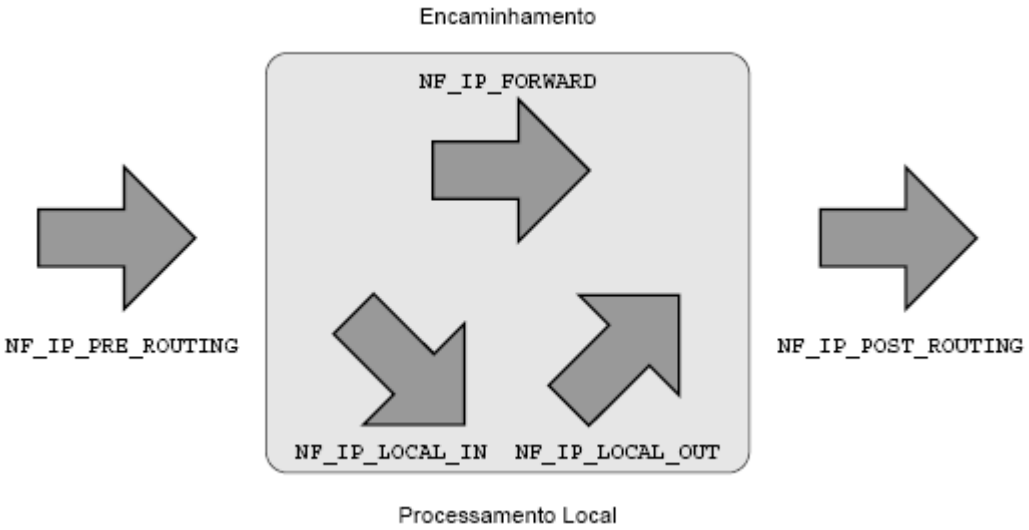

Figura 3.1: Estrutura de ganchos do Netfilter para o IPv4 (DREBES, 2005)

Após as funções dos ganchos tenham feito o processamento necessário no pacote, elas devem retornar um dos códigos definidos. Estes códigos podem ser vistos na Tabela 3.2.

Tabela 3.2: Códigos de retorno do Netfilter

Código	Significado
NF_DROP	Descarte do pacote, liberando recursos
NF_ACCEPT	Aceita o pacote
NF_STOLEN	Captura o pacote, que é desconsiderado pelas outras funções. Recursos são mantidos
NF_QUEUE	Enfileira pacote para ser tratado por função específica
NF_REPEAT	Reexecuta função com o pacote

Fonte: RUSSEL; WELTE, 2002

3.3 FIRMAMENT

Proposto por Roberto Jung Drebes na sua dissertação de mestrado, o FIRMAMENT (DREBES, 2005) foi desenvolvido no Grupo de Tolerância a Falhas da UFRGS. Ele foi baseado em outra ferramenta do mesmo grupo, o ComFIRM (LEITE, 2000).

O objetivo de ambas as ferramentas é descrever os cenários de falhas de comunicação para a avaliação de sistemas distribuídos e de protocolos de comunicação. Contudo, o ComFIRM pecava na flexibilidade, pois era necessária a alteração e recompilação do código fonte do kernel para a sua execução, entre outros problemas. Em ambos os casos deseja-se descrever os cenários com o mínimo de intrusividade possível. O texto que segue é baseado na dissertação de Drebes (2005).

3.3.1 Recursos do Netfilter utilizados pelo FIRMAMENT

As comunicações na internet atualmente utilizam o modelo TCP/IP para troca de mensagens. As alterações feitas em pacotes IP irão se refletir em todos os outros protocolos encapsulados nele. O modelo TCP/IP é tão comum que o protocolo IP está implementado no kernel do Linux. Por esses motivos o FIRMAMENT utiliza o kernel como ponto de injeção de falhas.

O FIRMAMENT tenta se posicionar o mais próximo possível do kernel do sistema operacional, para assim diminuir a sua influência no desempenho do mesmo. Para tanto, ele utiliza uma interface de programação do kernel, a qual é usada por controladores de dispositivos e outros subsistemas. Essa interface é o Netfilter.

Somente alguns dos ganchos disponíveis são usados pelo FIRMAMENT: NF_IP_PRE_ROUTING, NF_IP_POST_ROUTING, NF_IP6_PRE_ROUTING, NF_IP6_POST_ROUTING. Estes dois primeiros são para o protocolo IPv4, e os últimos para o IPv6. As funções disponibilizadas pelo Netfilter também não são todas utilizadas, restringindo-se ao: NF_ACCEPT, NF_DROP e NF_QUEUE.

3.3.2 Faultlets

A fim de especificar os cenários de falhas, o FIRMAMENT usa o conceito do faultlet. Um faultlet é uma aplicação que emula o comportamento de falhas. Ele pode atrasar, duplicar, ou descartar um pacote, entre outras ações disponíveis. Além disso, ele também atua sobre as variáveis de estado de fluxo, que são registrados de uso geral.

O faultlet pode executar operações lógicas e aritméticas sobre o pacote, assim podendo alterar a sua execução com base nesses dados lidos. Essa característica permite a criação de estruturas complexas para a inserção de falhas. Os faultlets podem ser configurados de forma independente para os fluxos de entrada e saída do protocolo IP.

3.3.3 FIRMVM

A máquina virtual FIRMVM tem a função de executar o faultlet, o qual é associado ao pacote e às variáveis de estado. Ela trabalha sobre as entradas e saídas dos protocolos IPv4 e IPv6, tendo assim os pontos de injeção de falhas. As variáveis de estado são um conjunto de 16 registradores de 32 bits, para cada um dos fluxos. Para o programador, esses registradores utilizam a representação de números inteiros com sinal no formato de rede (MSB/BIG_ENDIAN), e a conversão entre esse formato e o utilizado pelo hardware é feita de forma automática.

Os faultlets são especificados através de 31 instruções divididas em sete classes: instruções de entrada e saída, instruções lógicas e aritméticas, instruções de ação sobre o pacote, instruções de desvio de fluxo, instruções de manipulação de auto-incremento, instruções de manipulação de sequências de caracteres e outras instruções.

A máquina de registradores FIRMVM é Turing-completa, logo, é possível que entre em um laço infinito. Para evitar o colapso do nó de injeção de falhas, foi implementado um cão-de-guarda. Este detecta faultlets que estejam demorando muito tempo para terminar e interrompe a sua execução. O pacote então é aceito no estado em que estiver. O tempo necessário para a ativação do cão-de-guarda pode ser alterado pelo programador.

3.3.4 FIRMASM

O FIRMAMENT não interpreta diretamente os faultlets, antes é necessário que eles passem por um montador. O FIRMASM faz a verificação da tipagem dos parâmetros das instruções, gerando uma saída binária. Essa tradução gera um arquivo que é mais apropriado para o nível de núcleo, e ainda evita que faultlets mal escritos sejam carregados.

3.3.5 Controle e Monitoração

A utilização do FIRMAMENT é feita através de arquivos virtuais que estão localizados no diretório **/proc/net/firmament**. No subdiretório **rules** encontram-se as regras para a escrita e leitura dos faultlets em formato binário, separados pelo fluxo a que pertencem. Pode ser feita a passagem de comandos de controle diretamente para o injetor de falhas através do arquivo **control**.

Para se verificar se a ferramenta está funcionando de forma correta, é necessário que se tenha registros das instruções que são executadas pelo injetor. O FIRMAMENT utiliza o mecanismo de captura de mensagens do sistema **klogd(8)**. Os buffers de mensagens do núcleo são lidos do arquivo **/proc/kmsg** pelo utilitário **klogd**, o qual repassa as mensagens ao utilitário **syslogd(8)**.

Na figura 3.2 está descrito um esquema da relação conceitual entre os arquivos virtuais, comandos e faultlets, bem como os locais de atuação do injetor. Os números relacionam os arquivos virtuais de regras e aos pontos de interceptação associados.

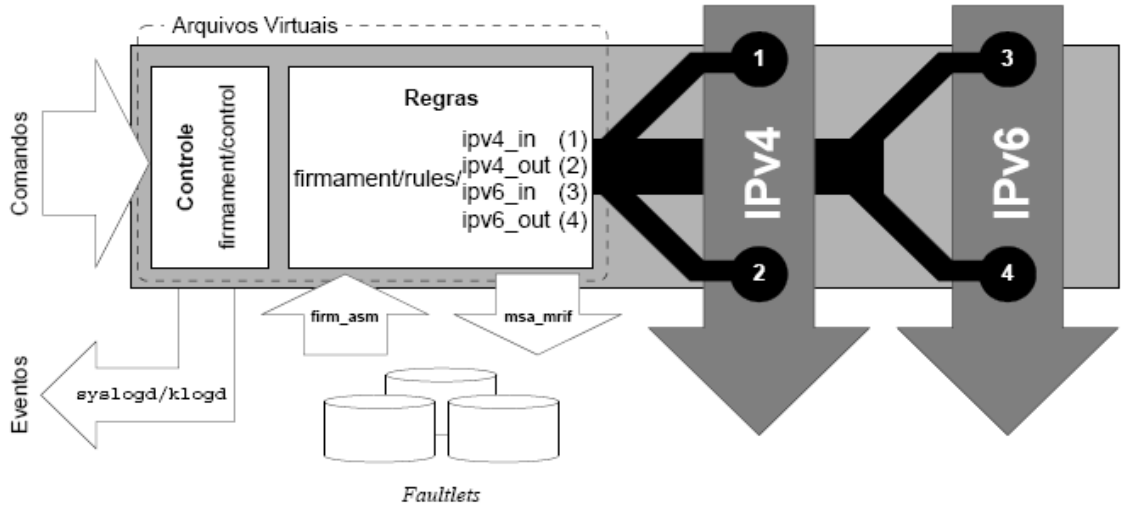

Figura 3.2: Relação conceitual dos elementos do FIRMAMENT (DREBES, 2005)

3.4 Android

A Open Handset Allience (OHA) é um grupo de mais de 40 empresas de áreas como operadores de telefonia, empresas de software, de semicondutores, entre outros. Liderada pela Google, a OHA visa a criação de padrões para a indústria da telefonia móvel. O primeiro passo nessa direção é a criação do Android.

Lançado oficialmente pela OHA em outubro de 2008, o Android é uma plataforma para dispositivos móveis baseado no sistema operacional Linux. Ele disponibiliza todos os softwares necessários para um dispositivo móvel: sistema operacional, *middleware*, e as aplicações essenciais. Mais aplicações podem ser desenvolvidas por terceiros, estas serão integradas ao sistema, e terão os mesmos recursos disponíveis.

A fim de facilitar a criação dessas aplicações, foi lançado um kit de desenvolvimento de software (SDK). A linguagem Java é utilizada, e todas as ferramentas necessárias para o desenvolvimento de aplicativos para o Android estão nesse kit. Entre os diversos componentes de desenvolvimento e de correção do SDK, pode-se destacar o emulador do Android, e o conjunto de ferramentas de desenvolvimento para Eclipse.

3.4.1 Arquitetura

A arquitetura do sistema operacional pode ser vista na figura 3.3. As suas camadas são:

- Aplicações: escritas em Java, essas serão as aplicações básicas que virão com o sistema. Tem-se cliente de e-mail, programa de SMS, calendário, mapas, contatos, entre outros.
- Framework de aplicação: para facilitar o reuso de código, desenvolvedores terão acesso as mesmas interfaces de aplicação. Entre os serviços que estarão disponíveis, tem-se os provedores de conteúdo, que farão o compartilhamento de dados entre aplicativos, gerente de notificação, que disponibilizarão diferentes tipos de alertas para as aplicações, e gerente de atividades, o qual fará a gerência do ciclo de vida das aplicações.
- Bibliotecas: oferece algumas bibliotecas básicas, que são acessíveis através do framework de aplicação. Tem-se: libc (versão compacta para sistemas embarcados), bibliotecas de mídia (baseado na PacketVideo's OpenCore, tem suporte aos formatos mais comuns de áudio e vídeo), gerente de navegação (controla acesso ao display), SQLite (base de dados relacional disponível para as aplicações).
- Android runtime: cada aplicação do Android roda o seu próprio processo, com uma instancia própria na máquina virtual Dalvik. A Dalvik foi desenvolvida para requerer pouca memória, e permitir que múltiplas instâncias de sua máquina virtual executem ao mesmo tempo. As aplicações são compiladas em Java e traduzidas para o formato “.dex” por uma ferramenta que está incluída no SDK.
- Linux Kernel: Tem como base o kernel Linux versão 2.6. Utiliza os serviços de gerência de memória e processos, pilha de rede, e controladores de dispositivos. Além disso, o kernel atua como uma camada de abstração entre o hardware e o resto do sistema.

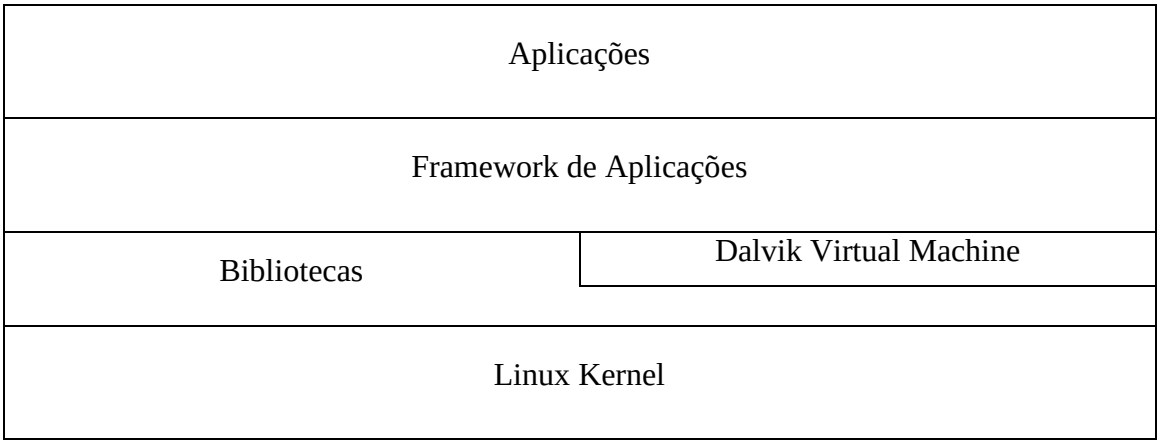

Figura 3.3: Arquitetura do Sistema Operacional Android (ANDROID)

3.5 Descrição do Projeto

O ambiente Android é baseado no kernel Linux 2.6, assim sendo, tem disponível o *framework* Netfilter. Com isso, iremos utilizar, ou adaptar se necessário, a ferramenta FIRMAMENT para realizar a campanha de injeção de falhas.

Um detalhe a ser ressaltado é que o ambiente Android não tem disponível a biblioteca *libc* completa, estando disponível somente uma versão compacta da mesma. Há relatos na Internet indicando esse problema (ARSTECH) como o maior empecilho para o reaproveitamento de códigos desenvolvidos para outros sistemas Linux. Por causa desses relatos foi considerada a opção de ser necessária a adaptação do FIRMAMENT para o ambiente Android, de modo a utilizar somente as bibliotecas disponíveis. Nessa hipótese a ferramenta poderia simular a perda de pacotes, e se possível, outros métodos de injeção de falhas, tendo em vista o tempo disponível para a realização da segunda etapa do Trabalho de Conclusão.

4 DESENVOLVIMENTO

Nesta seção são apresentadas as ferramentas utilizadas para o desenvolvimento do projeto: a Plataforma de Desenvolvimento do Android, o kernel Linux e as ferramentas necessárias para a execução. Também está descrita como foi realizada a integração destas ferramentas para a injeção de falhas, e por fim finaliza-se com as principais dificuldades encontradas.

Visando atender a proposta deste estudo, que consiste em portar uma ferramenta de injeção de falhas para avaliar a cobertura de falhas de comunicação em aplicações desenvolvidas para o ambiente Android, tornou-se indispensável averiguar na Internet recursos que pudessem fornecer subsídios para auxiliar na execução da tarefa

Inicialmente pesquisou-se o site oficial de desenvolvimento do Android, (DEV-ANDROID). Entre as diversas informações foram encontradas as instruções para o download da plataforma de desenvolvimento, incluindo: as bibliotecas específicas para o sistema, o compilador adequado e os headers do sistema operacional. Também foi encontrado link para o SDK, o qual traz ferramentas necessárias para o desenvolvimento de aplicações do Android, entre elas: o emulador e um debugger.

No mesmo site localizam-se diversas versões do kernel Linux para download, entre elas há uma para os aparelhos físicos e outra para ser executada no emulador. Em ambos os casos pode-se ter acesso a versões anteriores, caso seja necessário. Esta diferença entre as versões do kernel deve-se ao fato do Emulador apresentar limitações, principalmente na parte de conexão com um computador e entre os aparelhos. Por esse motivo é complicado testar-se aplicativos que utilizam o Bluetooth ou o USB no Emulador.

Diante disso, constata-se que este site é mais indicado para os que desenvolvem aplicativos, contudo pode ser considerado um bom ponto de partida para se adquirir um conhecimento basal sobre o Android.

4.1 Plataforma de Desenvolvimento

A Plataforma de Desenvolvimento do Android pode ser encontrada no repositório oficial (ANDROID-GIT). Este site usa um sistema de controle de versões distribuído, o Git, isso facilita o compartilhamento das informações sobre as atualizações que são desenvolvidas pela comunidade.

Constatou-se inicialmente que a plataforma disponibilizada é bastante completa e se caracteriza por possuir muitos dos elementos necessários para a compilação e execução do Android. Entre estes, destacam-se os códigos fonte para a máquina virtual e o

sistema de dados, a biblioteca libc específica que foi desenvolvida, chamada Bionic, os headers do kernel versão 2.6, e a máquina virtual Dalvik.

A biblioteca Bionic foi desenvolvida por três razões principais:

- Foi usada a licença BSD, se fosse usada a libc normal, a licença GPL contaminaria o espaço de usuário.
- A biblioteca deve ser carregada em cada processo, então é de interesse que ela não ocupe muito espaço. Para tanto a Bionic usa somente em torno de 200KB.
- Ela foi adiciona funções específicas para a aceleração do Android.

4.2 Kernel Goldfish

O Emulador do Android executa uma máquina virtual chamada Goldfish, que funciona com instruções ARM926T e disponibiliza ganchos para entrada e saída de dados. Nela há arquivos específicos para a exibição na tela, e a entrada de dados se dá através do teclado, e da tela touchscreen. Essas interfaces são usadas somente pelo Emulador, não sendo compiladas em um aparelho físico.

Pode-se fazer o download dos códigos fontes do kernel – para o Emulador e para os aparelhos – no mesmo link da plataforma de desenvolvimento. Quando a página foi acessada, a versão mais recente do kernel era a 2.6.29, esta foi a utilizada nesse trabalho.

4.3 Plataforma de Execução

Para realizar a avaliação da cobertura de falhas de comunicação em aplicações desenvolvidas para o ambiente Android foi necessário emular-se o ambiente no qual o Android estaria sendo executado, uma vez que não havia um aparelho disponível para a realização dos testes. Prevendo essa dificuldade, foi lançado, juntamente com o Android, um Kit de Desenvolvimento de Software (SDK). Este Kit possui várias ferramentas necessárias para o funcionamento do sistema, sendo então utilizada a versão 1.6 do Kit, composta por: Android Virtual Devices (AVDs), Emulador, o Android Debug Bridge (ADB), entre outras. A seguir descritas suas principais características e resultados.

4.3.1 AVDs

Android Virtual Devices (AVDs) são configurações do emulador que fazem a modelagem do emulador. Cada AVD é composto de:

- Um profile de hardware: faz as configurações de hardware do Emulador. Por exemplo, a quantidade de memória disponível e o tipo de teclado disponível, isto é, se é QWERT ou numérico, entre outras opções.
- Um mapa para a imagem do sistema: aqui é definida a versão da plataforma que será executada pelo emulador.
- Outras opções: dimensões da tela, aparência, entre outros.

Para se executar um Emulador é imprescindível que seja criada antes uma AVD, ou se utilize aquela que já vem pré-configurada junto com o SDK. Criando-se várias AVDs

estas, serão independentes de qualquer outra que esteja na mesma máquina de desenvolvimento.

4.3.2 Emulador

O principal componente do SDK é o Emulador do Android. Ele é um emulador de um dispositivo móvel que permite o desenvolvimento e teste das aplicações que são desenvolvidas. Ele consegue simular os componentes típicos de um aparelho celular, como botões de navegação, um teclado, e até mesmo uma tela que pode ser clicada com o mouse.

No entanto, também há algumas limitações, como não ter suporte a conexões USB, a câmera, a fones de ouvido, ao estado de carga da bateria, a Bluetooth, e é claro, a realizações de chamadas. As chamadas, tanto realizadas quanto recebidas, podem ser simuladas através da geração de um evento específico no console do emulador.

O Emulador executa através de um roteador virtual, o qual não tem acesso a rede diretamente. O Emulador funciona como um processo normal da máquina de desenvolvimento, assim sendo, ele está sujeito as mesmas limitações que sua rede impõe aos outros processos. O roteador suporta todos os tráfegos TCP e UDP do Emulador, contudo outros protocolos, como o ICMP, por exemplo, ainda não são suportados. Atualmente, a falta de suporte do Emulador se estende para mensagens multicast e IGMP.

4.3.3 ADB

O Android Debug Bridge permite a manipulação de um Emulador ou um aparelho físico que estejam conectados na máquina de desenvolvimento. Ele é um programa client/servidor baseado em três componentes:

- Cliente: executado na máquina de desenvolvimento. É invocado a partir de um shell ao se executar um comando adb.
- Servidor: também é executado na máquina de desenvolvimento, é ele o responsável pela intermediação dos comandos originados do cliente e enviados ao Daemon.
- Daemon: executado no emulador ou aparelho como um processo em background.

Ao se iniciar um cliente adb, ele busca um servidor que esteja rodando. Não encontrando, inicializa um. Quando o servidor está rodando, ele utiliza a porta TCP 5037 da máquina de desenvolvimento para receber comandos dos clientes. Todos os clientes vão enviar para essa mesma porta. Ao inicializar-se um servidor, ele varre em todas as portas pares entre 5554 e 5584 buscando um emulador ou aparelho que esteja conectado. Na porta onde ele encontra um Daemon ADB ele se conecta. Observe que quando se executa um emulador, ou se conecta um aparelho à máquina de desenvolvimento, este vai ocupar duas portas sequenciais, uma porta par para si, e uma ímpar para o seu adb.

Há diversos comandos suportados pelo adb, sendo que todos eles são executados como o Administrador do sistema. A lista completa pode ser encontrada no site (ANDROID-ADB). Nesse trabalho somente os comandos shell, push e pull foram utilizados.

O Shell pode ser considerado um comando que facilita a utilização dos outros comandos. Pois ao utilizá-lo abre-se uma interface que permite a navegação pelo sistema de arquivos do emulador, bem como a execução de aplicativos e outros comandos utilizados neste estudo.

Com a utilização do push foi possível realizar a cópia dos arquivos que estavam na máquina de desenvolvimento para o emulador, e com o pull foi realizada a cópia dos arquivos do emulador para a máquina de desenvolvimento dos testes da avaliação.

4.4 Integração das Ferramentas

A fim de se realizar o projeto proposto foi necessária a utilização de todas as ferramentas citadas acima. Para facilitar para os próximos que irão trabalhar com esse assunto, abaixo segue uma descrição dos comandos utilizados, bem como a sequência correta. O ambiente de desenvolvimento é um computador com processador Intel E7400, 2Gb de RAM, sistema operacional Windows como hospedeiro de uma máquina virtual VirtualBox V2.2.2 com Ubuntu versão 8.04 kernel 2.6.24.

1. Baixar fontes da plataforma de desenvolvimento, do Kernel Goldfish bem como do SDK.
2. A compilação do FIRMAMENT será dividida em 2 etapas. Primeiro será visto como compilar os componentes firm_asm e mas_mrif.
    a. Dentro da plataforma de desenvolvimento há o diretório */external* esse diretório contém os programas que estarão disponíveis no sistema. Para facilitar a organização, criou-se uma pasta *myapps* que irá conter os nossos aplicativos.
    b. Copia-se uma das pastas do */external* (a *ping* por exemplo) para dentro da pasta criada e altera-se o nome para *firm_asm*.
    c. Copiam-se os fontes do firm_asm para dentro dessa pasta e apaga-se os fontes originais da pasta.
    d. Altera-se no arquivo *Android.mk* as variáveis: LOCAL_SRC_FILES e LOCAL_MODULE para o nome do programa e o nome do executável gerado.
    e. Abrir um terminal e mudar o diretório para
        $ANDROID_HOME/external/myapps/firm_asm.
    f. Setar as variáveis de compilação com o comando:
        source $ANDROID_HOME/build/envsetup.sh
    g. A compilação propriamente dita é feita com o comando:
        mm
    h. O executável será criado na pasta:
        $ANDROID_HOME/out/target/product/generic/system/bin/
    i. Para a compilação do *mas_mrif* seguem-se os mesmos passos descritos acima, com as devidas alterações no nome do arquivo a ser compilado.
3. A compilação da máquina virtual *firm_vm* segue um processo um pouco diferente.

a. É necessário ajustar as bibliotecas e o kernel que vai ser usado durante a compilação. No diretório do *modules* do FIRMAMENT insere-se no *Makefile* as linhas:

CROSS_COMPILE = $ANDROID_HOME/prebuilt/linux-x86/toolchain/arm-eabi-4.3.1/bin/arm-eabi-

KERNEL_DIR ?= $GOLDFISH_KERNEL_HOME/

b. Abrir um terminal e mudar para o diretório do FIRMAMENT.

c. Como a arquitetura do Emulador não é a mesma da máquina de desenvolvimento, precisa-se setar a arquitetura alvo, isso é feito com o comando:
export ARCH= arm

d. Compilar usando:
make

e. O arquivo resultante *firm_vm.ko* será criado na pasta atual.

4. O kernel utilizado pelo Android não tem por padrão o suporte ao Netfilter, assim sendo é necessário que se recompile o kernel.

a. Primeiro deve-se habilitar o suporte ao Netfilter e selecionar a opção de carregamento de módulos. Essas opções podem ser selecionadas pelo arquivo de configurações do kernel, mas é mais fácil usar-se o menu de configuração disponível pelo comando:

make menuconfig

b. Como a arquitetura alvo não é a mesma da máquina de desenvolvimento, é necessário setar-se a mesma, bem como as bibliotecas para compilação:

export ARCH= arm

CROSS_COMPILE = $ANDROID_HOME/prebuilt/linux-x86/toolchain/arm-eabi-4.3.1/bin/arm-eabi-

c. Com isso só falta compilar o kernel:

make

d. O arquivo que será carregado no emulador é o:

$GOLDFISH_KERNEL_HOME/arch/arm/boot/zImage

5. Após essas etapas é necessário criar-se um AVD. Dentro do SDK há uma série de ferramentas que serão utilizadas. A etapa pode ser feita através de uma interface gráfica, instalando-se um plugin do Eclipse. Pelo terminal executa-se o comando:

android create avd –n <nome>

6. Com isso estamos prontos para iniciar o emulador. É necessária atenção para que seja carregado kernel correto para execução.

emulator –avd <nome> -kernel $GOLDFISH_KERNEL_HOME/arch/arm/boot/zImage

7. Em outro terminal deve-se iniciar o adb:

   adb shell

8. Agora é possível copiar os arquivos desejados. A pasta */data* do sistema de arquivos do emulador contém os arquivos do usuário, logo, recomenda-se que seja copiado para essa pasta. Para isso usa-se o comando em um terminal paralelo ao do adb:

   adb push <arquivo a ser copiado> <pasta destino no emulador>

4.5 Dificuldades no Desenvolvimento

Durante o processo de desenvolvimento foram encontradas algumas dificuldades. Elas se deviam principalmente à falta de experiência e ao desconhecimento do funcionamento exato do sistema. Todos os problemas foram resolvidos com a leitura de manuais, quando a dúvida era referente a um tema mais geral, ou mesmo com o auxílio de outras pessoas através das listas de discussão como o (ANDROID-KERNEL) e o (ANDROID-BEGINNERS).

Num primeiro momento a compilação das ferramentas se mostrou uma tarefa complicada, pois era necessário que elas fossem cross-compiladas. A primeira tentativa foi feita seguindo o site (MOTZ). Seguindo as orientações do site foi possível recompilar o kernel habilitando o Netfilter. Contudo, esse tutorial foi feito em 2007, por isso ele indica compiladores e bibliotecas que não são os mais indicados para o desenvolvimento do sistema Android. Para se aperfeiçoar esse processo, foram utilizados o kernel, compilador e bibliotecas específicas do Android, os quais foram encontrados no site (ANDROID-GIT).

O outro problema que surgiu refere-se ao primeiro teste feito com o FIRMAMENT. A fim de garantir que as funcionalidades básicas estavam disponíveis, foi feito um faultlet simples com somente uma instrução para o DROP de pacotes, independente do protocolo, e origem/destino. O resultado não foi o esperado, pois ao iniciar-se o descarte, o ADB shell ficava travado. Isso acontecia porque toda a comunicação entre o ADB e o Emulador é feita com mensagens TCP. Além disso, a pilha utilizada pelo sistema operacional do Emulador é única, assim sendo, não há uma distinção entre mensagens de acesso à Internet ou de comandos do ADB. Para solucionar esse problema, os faultlet que foram carregados na máquina virtual precisavam fazer um controle pelo IP do destino ou da origem do pacote, dependendo do fluxo que estava sendo analisado e do objetivo do teste.

5 TESTES DA SOLUÇÃO PROPOSTA

Neste capítulo são apresentados os testes realizados para comprovar a validade da solução proposta. O ambiente de testes é um computador com processador Intel E7400, 2Gb de RAM, sistema operacional Windows como hospedeiro de uma máquina virtual VirtualBox V2.2.2 com um convidado Ubuntu versão 8.04 kernel 2.6.24.

5.1 Mecanismo de cão-de-guarda

Este primeiro teste não visa mostrar o funcionamento das instruções do FIRMAMENT, mas sim apresentar o comportamento da ferramenta quando o engenheiro de testes inadvertidamente carrega nela um *faultlet* que coloca a máquina virtual em um laço infinito. Nestas situações, um mecanismo de segurança, ou cão-de-guarda, detecta o tempo elevado de processamento do *faultlet*, interrompendo a sua execução e permitindo que o pacote seja entregue.

Na figura 5.1 é exibido o *faultlet* utilizado para acionar-se o cão-de-guarda. Primeiro avalia-se o IP pelo qual será feita a seleção para o acionamento do *faultlet*, caso o pacote tenha o IP esperado, será direcionado para o laço, lá permanecendo até a execução do mecanismo de cão-de-guarda.

O acionamento ou não do mecanismo não é percebido pelo usuário, mas sim pelo sistema operacional. Por esse motivo, sempre que é feita uma injeção de falhas, recomenda-se que o engenheiro de testes verifique os logs do sistema operacional para garantir que não ocorreram problemas com a injeção. Quando acontece o acionamento do mecanismo de cão-de-guarda, o resultado do *faultlet* deve ser ignorado.

Na figura 5.2 está uma parte do buffer de mensagens do kernel. Os pontos entre as linhas indicam que há diversas outras mensagens de erro que foram reportadas. Estas foram omitidas para melhor representação nesse relatório. Cada uma das mensagens refere-se a uma conexão que foi feita com o site desejado, e que acionou o cão-de-guarda.

```
        SET       16            R0       ; 12 é a origem e 16 o destino.
        READW   R0              R1       ; Site: www.ufrgs.br
        SET       0x8f360109 R0       ; IP 143.53.1.9 = 8f360109
        SUB       R0            R1
        JMPZ      R1            DOG
        ACP
DOG:    JMP     DOG
```

Figura 5.1: *Faultlet* para acionamento do cão-de-guarda

```
firm_vm: watchdog called for flow ipv4_out.
firm_vm: watchdog called for flow ipv4_out.
  .
  .
  .
firm_vm: watchdog called for flow ipv4_out.
firm_vm: watchdog called for flow ipv4_out.
```

Figura 5.2: Buffer de mensagens do sistema

5.2 Descarte de Pacotes

O segundo experimento tem por objetivo avaliar a capacidade do FIRMAMENT de descartar pacotes. Como pode ser visto na figura 5.3, primeiro é feito o tratamento para a identificação do tipo de pacote que se deseja capturar, no caso, um pacote UDP. Em seguida testa-se endereço de destino desse pacote, isso porque o *faultlet* será carregado no fluxo *ipv4_out*, caso fosse colocado como um regra no fluxo de entrada deveria se avaliar o remetente do pacote. Por último é feito um sorteio em que se avalia o número que é retornado e é feito o descarte de 10% dos pacotes enviados.

A comunicação foi estabelecida através de um sistema cliente/servidor. O servidor foi acionado no Emulador e fez o envio das mensagens para o cliente que estava na máquina hospedeira Windows. As mensagens enviadas continham uma numeração seqüencial, essa informação foi lida pelo cliente que também fez uma contagem de mensagens recebidas. O servidor foi programado para o envio de 10000 mensagens, com um tempo de 100ms entre cada uma.

Para o cálculo do tempo de execução tentou-se utilizar o comando time, contudo o comando não estava disponível para uso no Android. Por isso, o tempo foi medido pelo próprio servidor através da diferença de tempo do relógio entre o início e o final da execução do programa.

Na tabela 5.1 tem-se o resultado da execução com e sem a atuação do FIRMAMENT. Observa-se que a quantidade de pacotes descartados foi um pouco superior aos 10% previstos, mas isso se deve principalmente ao *random* utilizado para o sorteio dos números.

Tabela 5.1: Resultado do descarte dos pacotes

Rodada	Pacotes recebidos (%)	Tempo para envio (s)
Sem ação do FIRMAMENT	100	1020
Com ação do FIRMAMENT	89,63	1022

```
        SET     9       R0      ; deslocamento do pacote a ser lido
        READB R0        R1      ; R1 = pacote[R0] (pacote[9])
        SET     17      R0      ; protocolo a ser selecionado
        SUB     R0      R1      ; (6 para TCP, 17 para UDP..)
        JMPZ    R1      UDP     ; se igual (UDP), vai para UDP.
        ACP                     ; se diferente, o pacote é aceito
UDP:
        SET     16              R0      ; 12 é a origem e 16 o destino.
        READW   R0              R1
        SET     0xbd48139b R0   ; IP 155.19.72.189 = bd48139b
        SUB     R0              R1
        JMPZ    R1              IP
        ACP

; O pacote é UDP. Deve ser descartado com probabilidade de 10%.
; O sorteio do número é feito abaixo. Como os números aleatórios
; são calculados em módulo, a idéia é sortear um número entre -50 e 50,
; e verificar se ente número é menor que 40. Se o número estiver neste
; intervalo, ao se somar 40 ele ainda será negativo, sendo feito o
; descarte nesse caso. Os valores +50, -50 e 40 são multiplicados por 10
; pois o gerador de números aleatórios é mais justo com intervalos
; maiores, mas o algoritmo é o mesmo.

IP:
        SET     500     R0      ; módulo do número a ser sorteado
        RND     R0      R1      ; sorteio
        SET     400     R0      ; verifica se o numero sorteado
        ADD     R0      R1      ; é menor que -40
        JMPN    R1      DROP    ; caso afirmativo, descarta
        ACP                     ; caso contrário, aceita
DROP:   DRP
```

Figura 5.3: *Faultlet* para descarte de pacotes UDP.

5.3 Atraso de Pacotes

O próximo experimento visa simular o atraso de pacotes enviados pela rede. É importante simular-se esse tipo de comportamento, pois é comum a ocorrência de atrasos nas transmissões da Internet, seja por causa de tráfego, ou mesmo devido ao atraso no processamento do pacote recebido.

Esse experimento foi dividido em duas etapas, primeiro usando-se um atraso constante de 20ms e na segunda etapa é usado um atraso variável de 12 ±5ms.

Na figura 5.4 tem-se o *faultlet* utilizado para a inserção do atraso de 20ms em cada uma das mensagens enviadas. Primeiro é feita uma filtragem pelo tipo de pacote que sofrerá a ação do injetor, no caso pacotes UDP. Em seguida selecionam-se somente as

mensagens destinadas ao IP 155.19.72.189, que em hexadecimal é representado por “BD48139B”. Caso ambas as condições sejam verdadeiras, então o pacote é atrasado, do contrário, é aceito sem sofrer influência do injetor.

O envio e recebimento dos pacotes foram feitos com um sistema do tipo cliente/servidor semelhante ao do experimento de descarte de pacotes. O servidor foi executado no Emulador, enquanto o cliente foi executado na máquina hospedeira Windows.

Inicio-se a execução do cliente e do servidor, contudo, ao término da execução não se constatou diferença significativa no tempo para o envio das mensagens, havendo ou não a interferência do *faultlet*. Isso se deve ao fato de que a primeira mensagem sofrer um atraso de 20ms, assim como a segunda, mas assim como num pipeline, depois do atraso inicial, o fluxo se mantém constante, não se percebendo o atraso que foi inserido nas outras mensagens.

Para solucionar esse problema foi necessário alterar-se o servidor e o cliente. Após a modificação o servidor envia a mensagem para o cliente e aguarda a resposta. O cliente teve que ser modificado para que quando recebesse uma mensagem, enviasse uma resposta ao servidor. Dessa forma, cada uma das 10000 mensagens enviadas sofreu um atraso de 20ms, logo, o atraso total esperado na execução do programa era de 200 segundos, aproximadamente.

```
        SET     9       R0      ; deslocamento do pacote a ser lido
        READB R0        R1      ; R1 = pacote[R0] (pacote[9])
        SET     17      R0      ; protocolo a ser selecionado
        SUB     R0      R1      ; (6 para TCP, 17 para UDP..)
        JMPZ    R1      UDP     ; se igual (UDP), vai para UDP.
        ACP                     ; se diferente, o pacote é aceito
UDP:
        SET     16              R0      ; 12 é a origem e 16 o destino.
        READW   R0              R1
        SET     0xbd48139b      R0      ; bd48139b = 155.19.72.189
        SUB     R0              R1
        JMPZ    R1              IP
        ACP
IP:     SET     20      R0      ; igual: atrasa o pacote em 20ms
        DLY     R0
```

Figura 5.4: *Faultlet* para atraso constante de pacotes UDP em 20ms.

A figura 5.5 apresenta o *faultlet* para a segunda etapa do experimento. O início do *faultlet* é igual ao anterior, havendo diferença no que se refere ao tempo que será atrasado o pacote. Após ser confirmado o tipo de pacote e o endereço do destinatário, é feito um sorteio com números de -5 a +5. O resultado é somado ao valor constante 12 que fica armazenado no registrador R0. Esse valor desse registrador será usado como atraso para o pacote. O sistema cliente/servidor utilizado foi o mesmo que foi modificado para a primeira etapa desse experimento.

Na tabela 5.2 foi feita uma comparação com os tempos de execução de cada uma das situações. Primeiro executou-se o cliente/servidor sem a ação do injetor, depois se obteve o tempo para o envio das mensagens com o atraso constante de 20ms por mensagem, e por último tem-se o tempo de execução inserindo-se um atraso variável de 12±5ms. Note que os tempos em que houve a ação do injetor não foram exatamente como os planejados. Essas diferenças devem-se principalmente as imprecisões nas medidas e às diferenças de carga do sistema durante a execução dos testes.

```
        SET     9       R0      ; deslocamento do pacote a ser lido
        READB   R0      R1      ; R1 = pacote[R0] (pacote[9])
        SET     17      R0      ; protocolo a ser selecionado
        SUB     R0      R1      ; (6 para TCP, 17 para UDP..)
        JMPZ    R1      UDP     ; se igual (UDP), vai para UDP.
        ACP                     ; se diferente, o pacote é aceito
UDP:
        SET     16              R0      ; 12 é a origem e 16 o destino.
        READW   R0              R1
        SET     0xbd48139b      R0      ; bd48139b = 155.19.72.189
        SUB     R0              R1
        JMPZ    R1              IP
        ACP
IP:     SET     12      R0      ; atraso médio
        SET     5       R1      ; módulo do número aleatório.
        RND     R1      R2      ; sorteio: R2 = ramdon (-5, +5)
        ADD     R2      R0      ; atrasa pacote entre 7 e 17ms
        DLY     R0
```

Figura 5.5: *Faultlet* para atraso médio de pacotes UDP em 12ms.

Tabela 5.2: Resultado do atraso das mensagens

Rodada	Tempo para envio (s)	Tempo estimado com ação do FIRMAMENT (s)	Diferença entre estimado e realizado (%)
Sem ação do FIRMAMENT	1024	-	-
Com ação do FIRMAMENT (atraso constante)	1190	1224	3
Com ação do FIRMAMENT (atraso variável)	1127	1144	1,5

5.4 Comentários sobre os Testes

Os testes forma realizados utilizando-se o protocolo de transporte UDP com o servidor sendo executado no Emulador e o cliente na máquina hospedeira do

desenvolvimento. Como o servidor envia mensagens diretamente para um IP e porta específicos, caso se desejasse inverter a situação, com o cliente no Emulador, seria necessário fazer o redirecionamento de portas da máquina de desenvolvimento para a porta em que o cliente estaria sendo executado no Android.

O Emulador possui um comando para a finalidade de redirecionamento de portas. Primeiro, já com o Emulador rodando, é necessário estabelecer-se uma conexão via *telnet* com Emulador. Isso é feito com o comando:

telnet localhost <porta do Emulador>

Normalmente a porta em que o Emulador está executado é a porta 5554. Depois da estabelecida a conexão, deve-se adicionar o redirecionamento com o comando:

redir a*dd <protocolo>:<porta do host>:<porta do Emulador>*

Desse modo, todas as requisições recebidas na porta do host especificada re-encaminhadas diretamente e serão tratadas somente pelo Emulador. Quando se usa o browser do Emulador para acessar a internet, esse redirecionamento é feito de modo transparente pelo usuário.

6 CONCLUSÃO

A execução de testes é uma etapa de suma importância na validação de uma aplicação. Verificar o seu comportamento na presença de falhas, tão comuns em sistemas distribuídos e de redes, é necessário para validá-las frente à sua especificação. Uma técnica bastante utilizada para este fim é a injeção de falhas de comunicação. As ferramentas de injeção de falhas permitem ao desenvolvedor colocar um protótipo, ou uma implementação funcional, sob situações reais de ocorrência de falhas reais e avaliar o comportamento do mesmo.

Em dispositivos móveis também deve haver essa preocupação, tendo em vista a disseminação desses dispositivos e a sua crescente complexidade. Os atuais Smartphones possuem um sistema operacional completo. O mais novo desses sistemas é o Android. Desenvolvido pela Open Handset Allience, um grupo formado por diversas empresas de tecnologia, entre elas o Google. O sistema operacional Android é baseado no kernel Linux 2.6, mas ele não tem as mesmas funcionalidades que o Linux normalmente utilizado em computadores pessoais.

O objetivo desse trabalho foi o porte de um injetor de falhas para o sistema operacional Android. O injetor escolhido foi o FIRMAMENT, desenvolvido pelo Grupo de Tolerância a Falhas da UFRGS. O FIRMAMENT atua no kernel do sistema operacional, na forma de um módulo carregável. A ferramenta executa micro-programas que são processados para cada mensagem enviada ou recebida, emulando situações de falhas de comunicação.

Primeiro foi vista uma revisão sobre os conceitos básicos de tolerância a falhas, injeção de falhas, os tipos de falhas que podem ser inseridos pelas ferramentas. Após, fez-se uma revisão sobre as ferramentas de injeção de falhas de comunicação que já foram publicadas na literatura.

Num segundo momento apresentaram-se os recursos que serão utilizados para a realização do projeto. O Netfilter é o *framework* que fornece pontos de acesso para a pilha de protocolos do sistema operacional. Ele é a base para o funcionamento do FIRMAMENT. Em seguida apresentou-se o ambiente Android, que será utilizado nesse trabalho.

No capítulo seguinte foram descritas as diversas ferramentas que foram necessárias para se portar o FIRMAMENT, assim como o modo pelo qual essa integração foi feita. As dificuldades que foram encontradas durante esse processo também foram relatadas.

Os testes realizados confirmando o porte da ferramenta estão descritos no Capítulo 5. Demonstra-se a funcionalidade do mecanismo de cão-de-guarda, o descarte de pacotes, bem como um experimento para o atraso de pacotes. Ao final, comenta-se uma limitação do Emulador utilizado e um método para se compensar essa limitação.

Com o decorrer desse trabalho pode-se constatar que o sistema operacional Android, apesar de utilizar um kernel, arquitetura, e bibliotecas próprias, é compatível em diversos aspectos com o sistema operacional Linux utilizado em computadores pessoais. Assim como o FIRMAMENT pode ser portado para o Android, é razoável dizer que outras ferramentas também poderão ser reutilizadas, desde que sejam compiladas com os devidos cuidados.

REFERÊNCIAS

AVIZIENIS, A.; Laprie, J.; Randell B,; Landwehr C. **Basic Concepts and Taxonomy of Dependable and Secure Computing**. IEEE trans. on dependable and secure computing, V. 1, n. 1, jan 2004, pp 11-33

KRISHNA, P., VAIDYA, N., AND PRADHAN, D.; **Recovery in distributed mobile environments**. In Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems (1993), pp. 83-88.

ARLAT, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Laprie, J.; Fabre, J.; Martins, E.; Powell, D. **Fault-injection for dependability validation: a methodology and some applications**. IEEE Trans. on Software Eng., Special Issue on Experimental Computer Science, New York, vol. 16, n.2, p. 166-82, Feb. 1990.

HSUEH, Mei-Chen; Tsai, T. K.; Iyer, R. K. **Fault Injection Techniques and Tools**. Computer, pp. 75-82, abril 1997

CRISTIAN, F. **Understanding fault-tolerant distributed systems**. Communications of the ACM, vol. 34, n.2, p. 56-78, Feb. 1991

DAWSON, S; Jahanian, F.; Mitton, T. **ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations**. Proceedings of IPDS'96. Urbana-Champaign, USA. 1996

RIZZO, L. **Dummynet: a simple approach to the evaluation of network protocols**. ACM SIGCOMM Computer Communication Review. Vol. 27, n.1, p. 31-41, Janeiro 1997

STOTT, D. T.; Floering, B.,; Burke, D.,; Kalbarcyzk, Z.; Iyer, R. K. **NFTAPE: a Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors**. IEEE International Computer Performance and Dependability Symposium, pp. 91-100. 2000

CARSON, M.; SANTAY, D. **NIST Net – A Linux-based Network Emulation Tool**. **ACM SIGCOMM Computer Communications Review,** vol.33, pp.111-126, Julho 2003.

DE, P.; Anindya Neogi, Tzi-cker Chiueh. **VirtualWire: A Fault Injection and Analysis Tool for Network Protocols.** icdcs, pp.214, 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), 2003

CHANDRA, R,; Lefever, R. M.; Joshi, K. R.; Cukier, M.; Sanders, W. H. **A Global-State-Triggered Fault Injector for Distributed System Evaluation.** IEEE transactions On Parallel And Distributed Systems, v. 15, n. 7, July, p. 593-605. 2004

DREBES, R. J. FIRMAMENT: Um módulo de injeção de falhas de **comunicação para linux**. 2005. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RUSSEL, R.; Welte, H. (2002). **Linux net filter hacking HOWTO**. 2002. Disponível em: <http://www.netfilter.org/documentation/>

ANDROID-SOURCE. Disponível em: < http://source.android.com/ >. Acessado em: junho, 2009

HOARAU, W.; Sebastien Tixeuil, Fabien Vauchelles, **FAIL-FCI: Versatile fault injection**, Future Generation Computer Systems, Volume 23, Issue 7, Pages 913-919, August 2007.

ANDROID. Disponível em: < http://www.android.com/ >. Acessado em: junho, 2009.

ADMOB-mobile-metrics-april-09. Disponível em: < http://metrics.admob.com/ >. Acessado em: junho, 2009.

RIBEIRO, J.C. **mCrash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties**; Organization: University of Coimbra, Portugal; Supervisor: Prof. Mário Zenha-Rela; Period: 2005-2008; Presentation Date: 17th of December, 2008.

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.

ARSTECH. Disponível em: < http://arstechnica.com/open-source/reviews/2009/02/an-introduction-to-google-android-for-developers.ars >. Acessado em: junho, 2009

DEV-ANDROID. Disponível em: < http://developer.android.com/index.html >. Acessado em: agosto, 2009.

ANDROID-KERNEL. Disponível em: < http://groups.google.com/group/android-kernel >. Acessado em: setembro, 2009.

ANDROID-ADB. DisponíveL em: < http://developer.android.com/guide/developing/tools/adb.html >. Acessado em: setembro, 2009

ANDROID-BEGINNERS. Disponível em: < http://groups.google.com/group/android-beginners >. Acessado em: setembro, 2009.

MOTZ. Disponível em: < http://honeypod.blogspot.com/2007/12/compile-android-kernel-from-source.html >. Acessado em: agosto, 2009.

ANDROID-GIT. Disponível em: < http://android.git.kernel.org/ >. Acessado em: setembro, 2009.