UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE TECNOLOGIA

ANA PAULA SAYURI MATSUNAGA

Avaliação da qualidade dos testes de detecção de vulnerabilidades apoiada pela análise de cobertura de código

Limeira, 2016.

ANA PAULA SAYURI MATSUNAGA

Avaliação da qualidade dos testes de detecção de vulnerabilidades apoiada pela análise de cobertura de código

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Tecnologia da Faculdade de Tecnologia da Universidade Estadual de Campinas, para a obtenção do título de Mestra em Tecnologia.

Área de Concentração: Tecnologia e Inovação.

Orientadora: Profa. Dra. Regina Lúcia de Oliveira Moraes

Este exemplar corresponde à versão final da dissertação defendida pela aluna Ana Paula Sayuri Matsunaga e orientada pela Profa. Dra. Regina Lúcia de Oliveira Moraes

Limeira, 2016.

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Silvana Moreira da Silva Soares - CRB 8/3965

M429a

Matsunaga, Ana Paula Sayuri, 1988-
Avaliação da qualidade dos testes de detecção de vulnerabilidades apoiada pela análise de cobertura de código / Ana Paula Sayuri Matsunaga. – Limeira, SP : [s.n.], 2016.

Orientador: Regina Lúcia de Oliveira Moraes.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Engenharia de software. 2. Software - Testes. I. Moraes, Regina Lúcia de Oliveira,1956-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Evaluation of the quality of vulnerability detection tests support by code coverage analysis
Palavras-chave em inglês:
Software engineering
Computer software - Testing
Área de concentração: Tecnologia e Inovação
Titulação: Mestra em Tecnologia
Banca examinadora:
Regina Lúcia de Oliveira Moraes [Orientador]
Márcio Eduardo Delamaro
Eliane Martins
Data de defesa: 29-02-2016
Programa de Pós-Graduação: Tecnologia

DISSERTAÇÃO DE MESTRADO EM TECNOLOGIA

ÁREA DE CONCENTRAÇÃO: TECNOLOGIA E INOVAÇÃO

Avaliação da qualidade dos testes de detecção de vulnerabilidades apoiada pela análise de cobertura de código

Ana Paula Sayuri Matsunaga

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

_____

Profa. Dra. Regina Lúcia de Oliveira Moraes
FT/UNICAMP
Presidente

_____

Prof. Dr. Márcio Eduardo Delamaro
ICMC/USP

_____

Profa. Dra. Eliane Martins
IC/UNICAMP

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no processo de vida acadêmica do aluno.

Dedico este trabalho aos meus amados pais, Roberto e Geni.

Agradecimentos

À Deus, por ter me dado saúde, força e pelas bênçãos em minha vida para que fosse possível concluir o meu mestrado.

Agradeço à orientadora Profª Drª Regina Lúcia de Oliveira Moraes por ter acredito em mim, pela dedicação, orientação e incentivos durante essa caminhada. Obrigada por me ajudar a realizar este sonho.

Ao Nuno Antunes pela influência, apoio, e todo o aprendizado na execução do trabalho.

Ao Centro de Pesquisa e Desenvolvimento em Telecomunicações (CPqD) pelo tempo concedido para realizar este trabalho.

Agradeço aos meus pais, Roberto e Geni, por todo amor, apoio e confiança que depositaram em mim. Sei o quanto se sacrificaram durante toda a vida para que eu tivesse esta oportunidade, meu agradecimento especial a vocês!

Aos meus irmãos: Roberta, Tayssa e Francisco. Pelo carinho e companheirismo de toda a vida.

Agradeço ao meu noivo Luiz Guilherme, pela paciência e apoio durante a minha carreira acadêmica.

Enfim, agradeço a toda a UNICAMP, base de toda a minha formação.

Resumo

Inúmeros são os desafios relacionados à detecção de vulnerabilidades de sistemas disponíveis na Web que, frequentemente são implementados com vulnerabilidades no código. Esses desafios se estendem também aos *Web Services*. Um dos problemas recorrentes é a exploração das vulnerabilidades de um serviço por usuários mal-intencionados. Esses usuários têm como objetivo atacar o sistema para ter acesso a dados privados e acessos não autorizados. Por esse motivo, empresas investem constantemente na segurança de sistemas computacionais, com a finalidade de proteger suas informações, bem como os dados de seus clientes.

Prestadores de serviços disponibilizam ou comercializam ferramentas para auxiliar na detecção de vulnerabilidades em aplicações de software. Estas ferramentas, denominadas *Scanners* de Vulnerabilidades, foram alvo de vários estudos, cujos resultados apontaram uma baixa eficácia dessas ferramentas, devido às suas altas taxas de falsos positivos e baixa cobertura. A cobertura de código é frequentemente considerada uma métrica de qualidade de testes e, consequentemente, de dependabilidade do software. Embora essa afirmação possa ser válida no âmbito de testes de detecção de vulnerabilidades, na prática tal suposição ainda está para ser demonstrada.

Este trabalho propõe estudar a relação entre as medidas de cobertura de código e a qualidade dos resultados dos testes executados por ferramentas de detecção de vulnerabilidades, em especial vulnerabilidades de Injeção SQL. Essa relação é de extrema importância para auxiliar os desenvolvedores a avaliarem a qualidade dos detectores de vulnerabilidades e para comparar ferramentas alternativas. Para estudar esta relação, foram consideradas duas ferramentas de detecção de vulnerabilidades e múltiplas configurações de testes.

Os resultados mostram que a cobertura de código é de fato um meio eficaz para estimar a qualidade de testes de detecção de vulnerabilidades e é útil para comparar diferentes conjuntos de testes. No entanto, eles também mostram que algumas métricas são claramente melhores do que outras e, que métricas de domínios específicos são mais eficazes do que as genéricas.

Abstract

There are many challenges related to the detection of vulnerabilities in systems that are exposed in the Web, which are frequently implanted with code vulnerabilities. These challenges are also extended to Web Services. One of the most recurring problems is the exploitation of vulnerabilities in the software by users who do not have good intentions and aim to attack the system to access private data and unauthorized access. For that reason, companies have been constantly investing in system security, in order to protect their information and the information of their customers.

Service providers offer tools that help to detect vulnerabilities in software applications. These tools, called Vulnerability Scanners, have been the target of several studies. These studies show low efficacy in these tools due to their high rates of false positives and low coverage. Code coverage is frequently considered a metric of the quality of the tests and, consequently, of the software dependability. Although this affirmation may be valid in the context of vulnerability detection tests, in practice, this assumption has yet to be demonstrated.

This work proposes to study the relation between code coverage measurements and the quality of the results of tests executed by vulnerability detection tools, in particular SQL Injection. Such relation is very important for developers to evaluate the quality of vulnerability detectors are and to compare alternative tools. To study this relationship, were considered two state-of-the-art tools and multiple testing configurations.

Results show that code coverage is indeed an effective mean to estimate the quality of vulnerability detection tests and is useful to compare different sets of tests. However, they also show that some metrics are clearly better than others, and that domain specific metrics are more effective than generic ones.

Lista de Figuras

Lista de Tabelas

Sumário

1. INTRODUÇÃO

Disponível vinte e quatro horas por dia, sete dias na semana e acessível a partir de qualquer lugar, a Internet é alvo constante de ataques de usuários mal-intencionados. Esses usuários, denominados *hackers,* concentram seus esforços para atacar sistemas que possuam carrinhos de compras, formulários, páginas de *login*, listas de clientes, entre outros (ACUNETIX, 2013).

De acordo com a empresa Acunetix (ACUNETIX, 2013), mais de 70% dos sites têm vulnerabilidades que podem ser exploradas por *hackers*. Estão entre as principais consequências dos ataques a sistemas *Web*: violação de privacidade, roubo de identidade, comprometimento de sistema, alteração de informações, destruição de dados e perda financeira.

Além da dificuldade em detectar ataques que ameaçam os sistemas *Web*, a evolução constante das técnicas utilizadas pelos atacantes torna a proteção dos sistemas uma tarefa ainda mais difícil (OWASP, 2013). Diversos são os casos envolvendo ataques a sistemas *Web.* A empresa de software Adobe anunciou que uma série de ataques a seus sistemas tornaram cerca de 38 milhões de usuários vulneráveis, tendo seus dados e senhas expostos aos *hackers* (EXAME, 2013). A empresa *Sony Computer Entertainment* também sofreu um ataque em abril de 2011. Este ataque levou ao desligamento temporário dos serviços da *PlayStation Network & Qriocity*, expondo dados pessoais de inúmeros usuários (PLAYSTATION, 2012). Note que, mesmo empresas que trabalham com tecnologia têm seus produtos expostos às ações de invasores.

Para evitar os crimes virtuais, a preservação da confidencialidade, da integridade e da disponibilidade são características essenciais que a segurança da informação deve ter como preocupação (ISO/IEC 17799, 2000). Segundo a ISO/IEC 17799 (2000):

> "A segurança de informações protege as informações contra uma ampla gama de ameaças, para assegurar a continuidade dos negócios, minimizar prejuízos e maximizar o retorno de investimentos e oportunidades comerciais." (ISO/IEC 17799, 2000).

A Internet e as aplicações Web não têm sido as únicas tecnologias que sofrem com a audácia e perspicácia de atacantes. *Hackers* têm estendido suas habilidades de invasão a *Web Services*. Esses serviços são soluções comumente utilizadas na integração e comunicação entre aplicações que empregam diferentes tecnologias. *Web Services* permitem a interação de sistemas que já existem (sistemas legados) com sistemas que estão sendo desenvolvidos, bem como torna possível a integração de sistemas que utilizam plataformas diferentes. Os sistemas integrados podem ter, cada um, uma linguagem própria que é traduzida para a linguagem universal de *Web Services*, o XML (ALONSO *et al.,* 2004).

Devido a ataques constantes às aplicações Web, que acarretam um prejuízo financeiro considerável para empresas, esforços têm sido realizados para não permitir que *Web Services* se tornem alvos fáceis para os *hackers*. Algumas iniciativas já foram desenvolvidas para protegê-los desses ataques (ANTUNES *et al.*, 2009; ANTUNES; VIEIRA, 2011). Por exemplo, a indústria de tecnologia da informação tem disponibilizado ferramentas denominadas *scanners* de vulnerabilidades, que auxiliam os desenvolvedores na detecção de vulnerabilidades que permitem que ataques de segurança sejam bem-sucedidos. Essas ferramentas são capazes de identificar automaticamente vulnerabilidades de segurança em sistemas computacionais, ao testar as aplicações e podem ser usadas em qualquer etapa do desenvolvimento do software, seja por desenvolvedores ou testadores (BASSO, 2010). Por serem automatizadas, são usadas para realizar centenas ou mesmo milhares de testes que de outra forma se traduziriam em uma tarefa repetitiva e cara. Uma vez detectadas, as vulnerabilidades devem ser corrigidas para prover a proteção dos dados gerenciados pelos *Web Services* (STUTTARD; PINTO, 2007).

Conforme verificado por Fonseca *et al.* (2007), Vieira *et al.* (2009), Basso *et al.* (2010) e DOUPÉ *et al.* (2010) os *scanners* de vulnerabilidades são ferramentas pouco eficazes, pois apresentam um grande número de falsos positivos (vulnerabilidades detectadas pela ferramenta, que não existem na aplicação) e baixa cobertura (vulnerabilidades que existem na aplicação, mas que não são detectadas pela ferramenta).

Além disso, as ferramentas encontradas no mercado detectam as vulnerabilidades de segurança em aplicações baseadas em *Web Services,* mas não apresentam um relatório da cobertura obtida na análise da aplicação em foco. A informação de cobertura da análise efetuada é importante para que os desenvolvedores ou testadores possam ficar atentos às partes do código

que não foram cobertas e que podem abrigar vulnerabilidades que, por não terem sido analisadas e apontadas, irão permanecer na aplicação em sua fase operacional.

A cobertura de código tem se mostrado eficaz para estimar a qualidade dos testes executados (HORGAN, 1994; MCCABE, 2009; VINCENZI *et al.*, 2006). A forma de medir não é trivial e pode ser feita de diferentes maneiras: cobertura de linhas, cobertura de caminhos, cobertura ramos, entre outros critérios. É muito difícil, se não impossível, alcançar a cobertura de código completa. Sendo assim, procura-se alcançar o maior percentual possível de cobertura para garantir a qualidade do software testado.

A análise de cobertura de código pode representar uma abordagem alternativa para estimar a qualidade dos detectores de vulnerabilidades. Uma relação entre a possível quantidade de vulnerabilidades que não foram detectadas e a cobertura do código, daria ao usuário uma medida de confiabilidade do código em relação à segurança. Embora isto possa ser uma suposição válida, ainda está para ser confirmada ou validada.

1.1.OBJETIVO

Este trabalho concentra-se na fase de testes do ciclo de desenvolvimento de software e, em particular na perspectiva de teste de segurança automatizado, e propõe técnicas de análise da cobertura da parte do código testado por ferramentas que detectam vulnerabilidades. A validação da abordagem proposta utilizou a adaptação de ferramentas para a detecção de vulnerabilidades de injeção em infraestruturas de software baseada em serviços.

De maneira geral, temos três motivações:

1) oferecer uma abordagem para que os desenvolvedores possam lidar com a restrição do tempo de desenvolvimento imposta pelo mercado competitivo que acaba por obrigar os profissionais da área a focar na satisfação dos requisitos funcionais dos usuários não levando em conta os aspectos de segurança;

2) auxiliar as equipes de desenvolvimento de software a minimizar ataques direcionados a *Web Services* que, de forma semelhante a outras aplicações Web, são tão expostos que *hackers* provavelmente irão descobrir qualquer vulnerabilidade de segurança existente;

3) validar a abordagem utilizando ataques de injeção em cenários baseados em serviços, por serem esses os mais perigosos e comuns nesse contexto (OWASP, 2013). O objetivo principal deste trabalho é apresentar a relação entre a cobertura de código e a qualidade dos resultados dos testes de ferramentas de detecção de vulnerabilidades, oferecendo técnicas que auxiliem na melhoria da qualidade desses testes.

A nova abordagem proposta foi aplicada em duas ferramentas de detecção de vulnerabilidades diferentes, a Sign-WS (ANTUNES e VIEIRA, 2011) e RAD-WS (ANTUNES *et al.,* 2009). Foram definidos diferentes tipos de configuração para testar um conjunto de *Web Services* (com e sem as vulnerabilidades), enquanto métricas de cobertura eram coletadas.

Além das tradicionais métricas de cobertura de código, foi criada uma nova métrica que é específica para testes de segurança e inspirada na cobertura de instrução destino (SMITH et al., 2008), à qual foi denominada no âmbito deste trabalho de cobertura *hotspot*. Essa métrica está relacionada a um conjunto de comandos do código (*hotspot*), uma vez que cada tipo de ataque tem como alvo comandos específicos.

Dessa forma, a cobertura *hotspot* trata da relação desses comandos com os testes efetuados.

Na prática este estudo pretende responder às seguintes questões:

Q1. Existe uma relação entre os valores de cobertura de código e o número de vulnerabilidades reportado?

Q2. Existe uma relação entre os valores da cobertura *hotspot* e o número de vulnerabilidades reportado?

Q3. Métricas de cobertura de código e cobertura *hotspot* são eficazes para comparar a qualidade de diferentes testes de detecção de vulnerabilidades?

Q4. Métricas de cobertura *hotspot* são eficazes para estimar a eficácia de uma ferramenta de detecção de vulnerabilidades?

1.2. CONTRIBUIÇÕES DO TRABALHO

Os resultados desse trabalho trazem contribuições para grupos que se preocupam com segurança em *Web Services* e que buscam alcançar maior confidencialidade, integridade e disponibilidade em seus serviços.

A primeira contribuição do trabalho foi a definição de uma abordagem para análise da qualidade dos testes de detecção de vulnerabilidades com base na cobertura de código. Além disso, foi proposta a métrica, denominada cobertura *hotspot*, que é especificamente definida para o tipo de vulnerabilidade analisada, nesse caso, vulnerabilidades de injeção SQL. Foi possível demonstrar que há uma relação entre os resultados de cobertura, tanto cobertura de código quanto cobertura *hotspot*, e a quantidade de vulnerabilidades detectadas, elevando assim a qualidade dos testes. Além disso, métricas de cobertura *hotspot* podem apoiar a obtenção de estimativas da qualidade de campanhas de testes em termos absolutos, permitindo estimar comandos potencialmente vulneráveis que não foram exercitados durante os testes de segurança.

Outra contribuição importante foi a integração da abordagem proposta a ferramentas de detecção de vulnerabilidades existentes. Como será apresentada nas seções desta dissertação, nossa abordagem é flexível, ou seja, pode-se adaptar a abordagem proposta a diferentes ferramentas de detecção de vulnerabilidades para realizar os experimentos e integrar os componentes responsáveis pelo cálculo da proporção de cobertura obtida.

1.3.ORGANIZAÇÃO DO TEXTO

Este trabalho foi dividido em seções e encontra-se estruturado da seguinte forma:

Capítulo 1 - Introdução. Apresentou o contexto em que se insere a pesquisa, ressaltou sua importância, apresentando seus objetivos e contribuições.

Capítulo 2 – Embasamento Teórico. Apresenta uma revisão bibliográfica em que comenta os trabalhos que antecederam a presente proposta e que ajudaram a embasar a pesquisa desenvolvida. São abordados temas como: teste de software, dependabilidade, análise de cobertura de código, *bytecode* Java, vulnerabilidades de segurança e análise de correlação simples.

Capítulo 3 - Trabalhos Relacionados. Reservada para a apresentação dos Trabalhos Relacionados ao tema pesquisado, este capítulo foi subdivida em três principais temas: *Scanners* de vulnerabilidades, Avaliação de *Scanners* de Vulnerabilidades e Cobertura de código.

Capítulo 4 – Abordagem Proposta. Traz a abordagem proposta e todas as modificações que foram necessárias para apresentar a abordagem experimental desenvolvida neste trabalho.

Capítulo 5 – Estudo de Caso. Descreve os dois estudos de caso que foram realizados para validar a abordagem proposta; foram utilizadas duas diferentes ferramentas: Sign-WS e RAD-WS.

Capítulo 6 – Resultados. Este capítulo é dedicado a apresentar os resultados dos dois estudos de caso; para maior clareza e facilidade de análise dos resultados obtidos, o capítulo foi divido em duas seções, uma para cada estudo de caso.

Capítulo 7 – Conclusão. Apresenta as conclusões do estudo desenvolvido e o que pode ser feito como trabalhos futuros.

2. EMBASAMENTO TEÓRICO

Ao longo deste capítulo serão descritos conceitos importantes para facilitar o entendimento deste trabalho. Os conceitos que serão apresentados são: (i) dependabilidade, descrevendo essa característica do software que é buscada pelos desenvolvedores e testadores; (ii) teste de software, apresentando esta fase do desenvolvimento do software; (iii) análise da cobertura de código e técnicas para auxiliar no processo de teste de software; (iv) *bytecode* Java, recurso importante para possibilitar o trabalho em tempo de execução quando o foco são programas desenvolvidos na linguagem Java; (v) vulnerabilidades de segurança, descrevendo as características dessa falhas que ameaçam o sistema e, por fim, (vi) análise de correlação e regressão linear, método utilizado para analisar os resultados obtidos pelos estudos de casos executados na pesquisa.

2.1. DEPENDABILIDADE

Dependabilidade: do inglês *dependability*, é caracterizada como uma propriedade fundamental de sistemas de computação. Segundo Avizienis *et al.* (2000) é a capacidade de oferecer um serviço que pode ser confiável, ou seja, defeitos de alta severidade não podem ocorrer com frequência para os usuários. A dependabilidade foi divida em três partes que a caracterizam: (i) atributos, (ii) meios de atingi-la e (iii) ameaças.

Os atributos que compõe a dependabilidade são: disponibilidade (prontidão para exercer sua função), confiabilidade (continuidade de serviço correto), segurança (do inglês *safety*, i.e. não apresentar consequências catastróficas para o usuário ou para o sistema), integridade (não permitir alterações indevida de dados) e manutenibilidade (capacidade de realizar manutenção sem grandes indisponibilidades no sistema).

Para se garantir a dependabilidade podem ser usados quatro meios (AVIZIENIS, 2000):

- **Prevenção de falhas.** Alcançada no projeto ou desenvolvimento do hardware e software. Incluem programação estruturada, ocultação de informações e modularização para o software e regras rigorosas de projeto para hardware;

- **Tolerância a falhas.** Preserva o sistema em presença de falhas. É implementada por detecção de erros e recuperação subsequente do sistema;
- **Remoção de falhas.** Realizada durante a fase de desenvolvimento e durante a vida operacional de um sistema. Consiste na verificação, diagnóstico e correção da falha;
- **Previsão de falhas.** Realizada através de uma avaliação do comportamento do sistema em relação à ocorrência de falha. A avaliação tem dois aspectos: qualitativo, que identifica e classifica as falhas, e quantitativo, que mede os atributos de confiabilidade que foram satisfeitos.

As ameaças à dependabilidade que foram definidas no artigo de Avizienis *et al.* (2000) são defeitos, erros e falhas. Estes três termos foram definidos da seguinte forma (LEITE e ORLANDO (1987) *apud* LAPRIE (1995)): **Defeitos** (*failures*), quando o sistema não está em conformidade com a especificação, ou porque a especificação não descreve adequadamente a sua função. **Erros** (*errors*) é a ativação de uma falha que altera o estado interno e que pode levar a um defeito. **Falhas** (*faults*), manifestação no software de um engano cometido pelo desenvolvedor, podendo ser a causa de um defeito.

Cada vez mais os testes de software são feitos para que os sistemas computacionais atinjam níveis altos de dependabilidade. Essa importante etapa da construção do software é explicada na próxima seção.

2.2. TESTE DE SOFTWARE

A disciplina de Engenharia de Software tem evoluído nas últimas décadas, o que inclui busca por novas técnicas, métodos, critérios e ferramentas para produzir software de alta qualidade e baixo custo (MALDONADO *et al.,* 1998). Mesmo com todos os processos para garantir a qualidade, diversos tipos de problemas podem ocorrer e o software criado resulta em um produto diferente do esperado.

Atividades como Verificação, Validação e Teste (VV&T) buscam construir o produto de acordo com o especificado. A Verificação é o processo que avalia se a cada fase do software ou alguma função deste, o resultado satisfaz ao que foi requerido no início dessa mesma fase. A

Validação é o processo que avalia se o software está de acordo com os requisitos (MALDONADO *et al.,* 1998).

Teste é a principal atividade para garantir a qualidade do software. O processo de teste de software busca determinar se o software produzido funciona corretamente e está em conformidade com o especificado. As características e dimensões do sistema influenciam na complexidade desta atividade, levando a diversos problemas no sistema que, na maioria das vezes, são causados por erros humanos (DELAMARO *et al.,* 2007).

A Figura 1, retirada de Pressman (2011), ilustra as etapas do processo de teste de software.

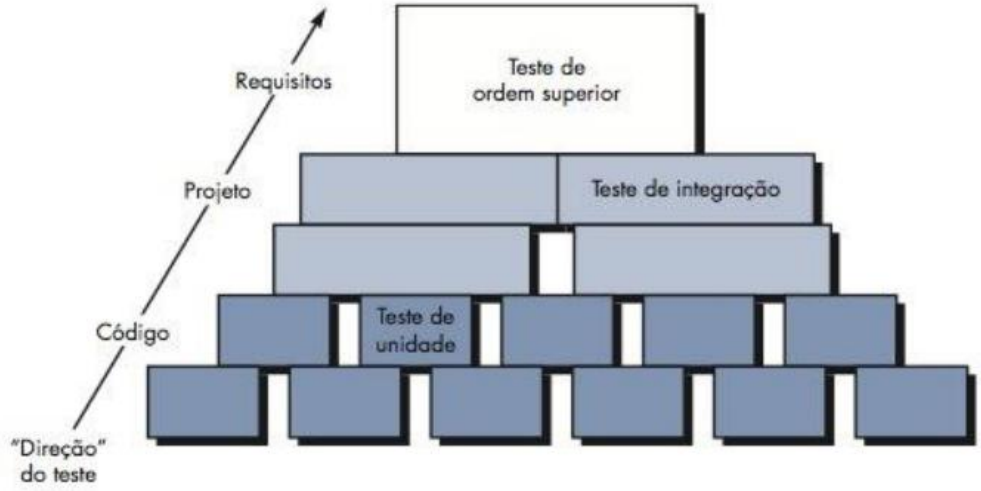

Figura 1 Etapas do teste de software (PRESSMAN, 2011).

A seta de "Direção" do teste mostra que os testes são iniciados com foco na menor unidade do software, em seguida tratam do projeto do software e por fim validam o sistema como um todo, dando ênfase aos requisitos. O software é testado em diversos momentos de sua construção para revelar erros ou defeitos. Pode-se dividir, de modo geral, este processo em três fases (DELAMARO *et al.,* 2007), conforme é mostrado na Figura 1:

i. Teste de unidade: realizado durante o processo de implementação e aplicado pelos próprios desenvolvedores. Deve garantir o funcionamento mínimo da menor unidade do software, que pode ser uma função ou classe, um componente ou o módulo do software. É baseado na estrutura interna do módulo e utilizado para verificar parâmetros de entrada e saída, condições limite, integridade dos dados e tratamento de erro.

ii. Teste de integração: após os componentes serem testados unitariamente, estes são integrados de acordo com a arquitetura do sistema e tem como objetivo detectar erros nas interfaces dos módulos do sistema. A integração dos componentes pode ocorrer de forma descendente (*top-down*), na qual começa com a unidade principal e aos poucos integra as unidades subordinadas; ou ascendente (*bottom-up*), que é uma abordagem inversa, a qual começa a integração pelas unidades de mais baixo nível da estrutura do programa. Por ser necessário o conhecimento das estruturas internas e das interações entre os módulos, o teste de integração tende a ser realizado pelos desenvolvedores.

iii. Teste de sistemas: após todas as partes do programa estarem integradas, é realizado o teste de sistema. Este teste foca nas ações visíveis para o usuário e tende a ser realizado por uma equipe independente que visa demonstrar que o software atende às funcionalidades esperadas pelo cliente. As funcionalidades são verificadas de acordo com os documentos de especificação de requisitos. Além dos requisitos funcionais, os requisitos não funcionais também devem ser verificados. Os testes sistêmicos são compostos por uma série de diferentes testes como, por exemplo, testes funcionais, teste de segurança, teste de esforço, teste de disponibilidade, entre outros. O foco do sistema em análise é que determina qual tipo de teste deve ser executado.

Além destas três fases de teste, há também um tipo de teste que é executado durante a manutenção do software, que é o teste de regressão. Este teste reexecuta um conjunto de testes já existentes a fim de encontrar novas falhas introduzidas devido à manutenção do sistema, que podem causar defeitos nas partes do software não modificados, devido as dependências existentes na implementação do produto de software.

De acordo com Meyers *et al.* (2011), um caso de teste bem-sucedido é aquele que é capaz de revelar a presença de defeitos no produto. Definir um conjunto de casos de teste que tenha alta probabilidade de encontrar defeitos é crucial para o processo de teste de software, uma vez que é impraticável realizar testes extensivos. Para auxiliar nessa atividade existem critérios de teste, que selecionam e avaliam determinado subconjunto de casos de teste de modo que estes aumentem as chances de revelar defeitos.

2.2.1. Técnicas e Critérios de Teste de Software

Existem basicamente três técnicas para se conduzir e avaliar a qualidade da atividade de teste. As principais características e as diferenças dessas técnicas são detalhadas a seguir:

- **Técnica Caixa Preta**

A técnica é conhecida como teste caixa preta porque o sistema é considerado uma caixa em que não é possível conhecer seus detalhes internos, e só é possível visualizar as entradas e as saídas. Nesta técnica, os casos de teste são projetados de acordo com a especificação do sistema, sem se preocupar com a implementação. Para este tipo de teste, é essencial uma especificação de requisitos bem elaborada (MALDONADO *et al.,* 1998; DELAMARO e MALDONADO, 2007). Os principais critérios desta técnica são (MALDONADO *et al.,* 1998):

- Particionamento em classes de equivalência: o domínio de entrada é dividido em classes de equivalência de acordo com a especificação do programa, de forma que é esperado que os elementos de uma mesma classe se comportem de maneira similar. Assim, é possível selecionar um subconjunto com dados de entrada representativos de cada classe, restringindo a quantidade de casos de testes a serem realizados, uma vez que para cada classe é suficiente a execução de um único caso de teste;
- Análise do valor limite: é um complemento do critério de particionamento em classes de equivalência. Ao invés de selecionar um dado de teste aleatório dentro da partição de equivalência, é selecionado o(s) dado(s) que explora(m) as condições limite, onde são altas as chances de se encontrar defeitos. Também é exigida a verificação das condições limites para os dados de saída;
- Grafo de Causa-Efeito: faz-se o levantamento de condições de entrada (causa) e as possíveis saídas (efeitos) e constrói-se um grafo de Causa-Efeito, que é convertido em uma tabela de decisão, a partir da qual são gerados os casos de testes. Com isso, é possível explorar combinações dos valores de entrada.

Conforme dito anteriormente, para a técnica caixa preta é necessária uma boa especificação de requisitos, uma vez que os casos de teste derivam desse insumo. A qualidade da especificação de requisitos, considerando-se o detalhamento, limita a qualidade dos testes que

serão aplicados. Além disso, não é possível assegurar que a parte em que se encontra falha no código tenha sido exercitada, uma vez que a cobertura é medida sobre a especificação de requisitos e não sobre a implementação.

- **Técnica Caixa Branca**

Também conhecido como teste estrutural, ao contrário do teste de caixa preta, é baseado no princípio de que se conhece a estrutura interna do programa para derivar os casos de teste. Deve-se garantir que os caminhos lógicos do software foram testados, exercitando decisões lógicas, *loops*, definições e usos de variáveis (MALDONADO *et al.,* 1998; DELAMARO e MALDONADO, 2007). Os principais critérios da técnica estrutural são (MALDONADO *et al.,* 1998):

- Baseado na complexidade: com base nesse critério, usa-se a complexidade do programa para gerar os casos de teste. Um exemplo deste critério é a utilização da complexidade de McCabe, que fornece medida quantitativa da complexidade lógica de um programa. Essa medida determina o número de caminhos independentes do programa para que todos os caminhos lógicos independentes sejam exercitados (PRESSMAN, 2011);
- Baseado em fluxo de controle: o critério utiliza-se das características de controle da execução do programa, como comandos e desvios e baseiam-se em um gráfico de controle de fluxo (CFG – *Control FlowGraph*) para derivar os casos de teste (VINCENZI *et al.*, 2006). Os critérios ***All-Nodes*** e ***All-Edges*** exigem que todos e todos os arcos do CFG sejam executados pelo menos uma vez. Estes critérios correspondem a instruções e decisões/ramificações respectivamente.
- Baseado em fluxo de dados: utiliza-se da análise do fluxo de dados e o gráfico *def-use* (DUG - *Def-Use Graph*) que complementa o CFG para derivar os requisitos de teste de fluxo de dados (RAPPS e WEYUKER, 1985) e traz informações sobre o conjunto de definições e usos subsequentes de variáveis. É possível extrair do DUG o critério ***All-Uses*****.** Dos CFG e DUG é possível extrair o critério ***All-Pot-Uses***, que exige que todos os caminhos entre a atribuição de *v* e todos os nós e arcos possíveis de se chegar por caminhos livres de novas definições de *v* são executados.

- **Técnica Baseada em Falhas**

Está técnica dá ênfase aos tipos mais frequentes de falhas que o programador ou projetista pode cometer durante o processo de implementação do software. Uma das primeiras formas de teste baseado em falhas foi a Análise de Mutantes, que realiza pequenas modificações em um programa *P* (mutantes) para avaliar quando um conjunto de teste é adequado para o teste de *P*. Assim, é possível determinar as diferenças de comportamento existentes entre o programa *P* e seus mutantes. Um dos problemas dessa técnica é o seu custo alto, uma vez que o número de mutantes gerados costuma ser alto e exige muito tempo para execução (MALDONADO *et al.,* 1998; DELAMARO e MALDONADO, 2007).

Uma técnica que vem sendo bastante utilizada e que é importante para o melhor entendimento deste estudo, é a técnica de Injeção de Falhas. Esta técnica auxilia o processo de teste sistêmico a garantir níveis maiores de dependabilidade e medir a eficácia dos mecanismos de tolerância a falhas. Dada a importância dessa técnica para o trabalho a ser desenvolvido, a próxima seção se dedica a explicar os seus conceitos.

2.2.2. Injeção de falhas

A injeção de falhas é uma técnica importante para sistemas que precisam garantir um determinado nível de dependabilidade. Essa técnica consiste em introduzir falhas ou erros no sistema alvo, a fim de monitorar seus efeitos e analisar suas respostas. Dessa forma, é possível verificar como o sistema se comporta em um cenário de falhas, avaliar a eficiência e o desempenho do sistema e sua capacidade de recuperação de falha (HSUEH, 1997). Caso o sistema não se comporte da maneira esperada na presença de alguma falha, poderá ser corrigido.

Hsueh *et al.* (1997) dividiram em três categorias as técnicas de injeção de falhas, que servem para avaliar o efeito das falhas em hardware e em software. São elas:

Injeção de falhas por simulação: é baseado na simulação para poder avaliar um sistema que está em fase de análise e projeto. As falhas são injetadas em um modelo do sistema alvo para identificar problemas de dependabilidade e verificar a eficácia de mecanismos de tolerância a falha e desempenho. Como ponto fraco desta técnica é citado que o modelo do sistema deve ser

uma representação fiel do sistema final, uma vez que os resultados desses testes serão sobre o modelo e não sobre o sistema.

Injeção de falhas por hardware: este tipo de técnica utiliza equipamentos adicionais para injetar falhas no sistema alvo, o que costuma ser caro para desenvolver e pode danificar o sistema alvo (LEME, 2001). Está dividido em duas categorias: injeção de falhas por hardware com contato, ou seja, que possui contato físico direto com o sistema alvo podendo realizar alterações de corrente elétrica e voltagem e a injeção de falhas por hardware sem contato, no qual o injetor não possui contato físico com o sistema alvo e uma fonte externa produz algum fenômeno físico no sistema alvo.

Injeção de falhas por software: é a técnica mais utilizada, uma vez que não requer hardware específico e pode injetar falhas em aplicações e sistemas operacionais. Há duas maneiras diferentes de injetar falhas por software: **Injeção em tempo de compilação:** o código-fonte ou o *assembly* deve ser modificado com a falha alterando as instruções do programa alvo. Não é necessário software adicional durante a execução, mas cria várias versões do programa. **Injeção em tempo de execução** necessita de software adicional, que é responsável por injetar falhas, monitorar os efeitos e disparar a injeção de falhas no programa alvo (LEME, 2001).

No artigo desenvolvido por Hsueh *et al.* (1997), foi apresentando o ambiente básico de injeção de falhas conforme a Figura 2. Para facilitar o entendimento da ilustração são definidos termos relacionados a Injeção de Falhas e que também são usados ao longo deste trabalho.

- *Gold run*: execução do sistema sem injeção de falha ou erro.
- *Workload*: é o nome dado a carga de trabalho do sistema, ou seja, as tarefas que devem ser executadas pelo sistema.
- *Faultload*: é o nome dado ao conjunto de falhas que serão injetadas no sistema alvo para simular as suas possíveis falhas;

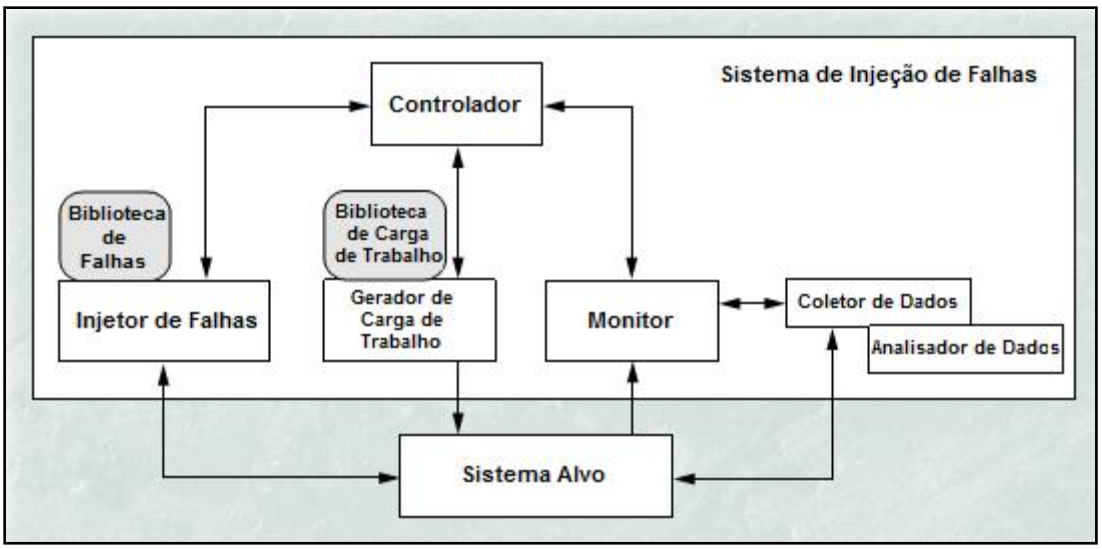

Figura 2 Ambiente de Injeção de Falhas (traduzido de Hsueh *et al.* (1997))

O ambiente de injeção de falhas funciona da seguinte forma: o Sistema Alvo executa as operações provenientes do módulo Gerador de Carga de Trabalho. A Biblioteca de Carga de Trabalho fornece para o módulo Gerador de Carga de Trabalho o *workload* que deve ser executado. O módulo Injetor de Falhas é responsável por injetar falhas no Sistema Alvo executando a carga de falhas durante a execução do sistema, que são uma série de falhas que foram projetadas e são representativas do contexto do sistema alvo. O módulo Biblioteca (referente à Figura 2) de Falhas representa o *Faultload*.

O monitoramente de toda execução é realizado pelo Monitor que também inicia o módulo Coletor de Dados. Os módulos Coletor de Dados e Analisador de Dados, coletam as informações necessárias e realizam a análise dos dados. Por fim, o módulo Controlador gerencia o experimento.

A verificação dos resultados da injeção de falhas pode ser feita pela comparação entre os resultados da *Gold Run*, e os resultados da execução do sistema em presença de falha.

Outra técnica que apoia o processo de teste de software e que também é alvo deste trabalho é a análise da cobertura de código, que avalia a qualidade dos testes efetuados. A próxima seção aborda este assunto.

2.3. ANÁLISE DA COBERTURA DE CÓDIGO

A análise da cobertura de código, também chamada de cobertura de teste, consiste em determinar quais áreas do código foram ou não exercitadas pelos casos de teste. A cobertura de teste pode ser baseada na especificação funcional de software ou na estrutura interna do programa. Como o primeiro depende da disponibilidade e regularidade da especificação, este último, também chamado de cobertura de caixa branca ou cobertura de código (MYERS, 2011), é usado com mais frequência. A cobertura de caixa branca combina as informações sobre o programa em execução com as informações do código que é objeto de análise, para gerar um relatório com a cobertura. Através do relatório é possível identificar trechos do código que não estão sendo devidamente testados e, com isso, adicionar novos casos de testes para aumentar a cobertura (CORNETT, 2013).

Esta cobertura de teste determina a medida quantitativa de cobertura de código, normalmente um porcentual da extensão do código analisado sobre a extensão total da aplicação, e é utilizada para averiguar a qualidade do conjunto de casos de teste. A cobertura de código também pode ser mensurada como a fração de códigos de programa que são executados pelo menos uma vez durante o teste (CAI e LYU, 2005).

A seguir foram definidas algumas métricas de cobertura de código, que são (YANG *et al.*, 2009; CORNETT, 2013; COBERTURA, 2015):

i. Cobertura de comando - cada comando do código do programa deve ser executado pelo menos uma vez para que a cobertura alcance 100%;
ii. Cobertura de ramificações - todas as ramificações devem ser executadas, inclusive as instruções *else* - ramo das instruções *if*;
iii. Cobertura de expressão: as instruções *if* recebem um tratamento maior para que cada expressão booleana possa receber pelo menos uma vez os valores *true* e *false*;
iv. Cobertura de *loop*: *loops* são propensos a falhas. Para cada loop deve ser verificado se não foi executado, se foi executado uma vez ou se foi executado o número máximo vezes;
v. Cobertura do operador ponto de interrogação (?): O operador (?) é uma expressão condicional; assim como na cobertura de ramificações, o operador (?) é 100% coberto

quando ambas as alternativas são executadas, 50% se apenas uma das alternativas e 0% caso nenhuma condição seja satisfeita;

vi. Cobertura de caminho: informa se todos os caminhos possíveis em cada função foram exercitados. O caminho é definido como uma sequência da entrada da função até a saída. Essa métrica possui como desvantagem o fato de a quantidade de caminhos ser exponencial ao número de ramificações e muitas vezes existem caminhos impossíveis de exercitar;

vii. Cobertura de função: informa se cada função ou procedimento foi invocado. É usada para assegurar cobertura em todas as áreas do software em testes unitários.

viii. Cobertura do fluxo de dados: considera somente subcaminhos de definição e usos subsequentes de variáveis.

Para auxiliar os desenvolvedores, foram criadas diversas ferramentas de cobertura de código, especialmente para aplicações Java. Na Tabela 1 são apresentadas algumas ferramentas que tem como função principal verificar a cobertura de código realizada pelos casos de teste unitários.

Tabela 1 - Relação de Ferramentas de Cobertura de Código.

Nome	Descrição
Cobertura	Esta ferramenta calcula as métricas de linhas, caminhos e ramificações. Tem como diferencial a apresentação da complexidade ciclomática McCabe de cada classe e a média da complexidade dos pacotes de um projeto. No final gera relatórios em HTML e XML (COBERTURA, 2015)
EMMA	A ferramenta apresenta cobertura de classes, métodos, linhas, blocos. É capaz de realizar a instrumentação das classes antes da execução ou durante a execução dos testes. Os relatórios gerados são nos formatos: HTML ou XML (EMMA, 2015)
Clover	*Clover* permite cobertura para código de aplicação e para testes de

	unidade e apresenta métricas de cobertura por comandos. Esta ferramenta é um *plugin* do ambiente de desenvolvimento eclipse (ECLIPSE, 2015) e o seu resultado é destacado diretamente no editor Java ou então pode ser gerado um relatório nos formatos HTML, XML e PDF (CLOVER, 2015).
JaBUTi	Realiza a cobertura de código usando critérios baseados em fluxo de controle e em fluxo de dados para programas na linguagem Java. A análise estática realizada pela ferramenta é a partir do código compilado, ou seja, do *bytecode* Java (VINCENZI *et al.*, 2003).

No entanto, algumas destas ferramentas trabalham em conjunto com bibliotecas de teste de unidade específicas e são muito limitadas em termos dos critérios analisados. A próxima subseção é dedicada a critérios de cobertura de código.

2.3.1. Critérios de cobertura de código

Como visto na técnica caixa branca, os critérios de teste de fluxo de controle baseiam-se em um CFG para derivar os casos de teste, como é o caso dos critérios *All-Nodes* e *All-Edges*, que requerem que cada nó e cada aresta do CFG sejam executados durante a campanha de teste (VINCENZI *et al.*, 2006). Estes critérios correspondem a cobertura de declaração e de decisão/ramificação, respectivamente. Um DUG complementa o CFG para derivar os requisitos de teste de fluxo de dados (RAPPS e WEYUKER, 1985) e traz informações sobre o conjunto de variáveis definidas e usadas em cada nó / aresta do CFG. O *def-use* representa o conjunto de instruções executadas a partir de uma atribuição de um valor a uma variável *v* e uma referência para *v*, sem outra atribuição para *v*. Com base no *def-use* é possível extrair um terceiro critério: *All-Uses*. Finalmente, um critério mais conservador pode ser extraído a partir do CFG e DUG: *All-Potential-Uses* (CRESPO *et al.,* 1997). Basicamente, este critério exige que todos os caminhos entre a atribuição de *v* e todos os nós e arestas possíveis de chegar por caminhos livres de definição de *v* são executados. A motivação é que superestimando os requisitos de teste possamos melhorar o rigor da atividade de teste.

Devido a linguagens com mecanismos de manipulação de exceção foram introduzidas notações de dependência de exceção (*ed - exception-dependence*) e de independência de exceção (*ei - exception-independence*), dividindo cada um destes critérios em dois conjuntos disjuntos. Basicamente, os nós dependentes de exceção são o subconjunto de nós do gráfico DUG que só pode ser alcançado após uma exceção ter sido levantada. Os nós restantes do grafo DUG são considerados independentes de exceção. O mesmo raciocínio se aplica a outros critérios.

Os critérios das técnicas baseada em fluxo de controle e baseada em fluxo de dados foram utilizados neste trabalho para medir a cobertura do código das aplicações. Para resumir, os critérios de cobertura de código que analisamos neste trabalho são (VINCENZI *et al.*, 2006; CRESPO *et al.,* 1997):

All-Nodes-ei **(independente de exceção)**: todos os nós que não fazem parte de caminhos de exceção sejam cobertos pelo caminho de teste;

All-Nodes-ed **(dependente de exceção)**: todos os nós de um caminho de exceção são cobertos pelo caminho de teste;

All-Edges-ei **(independente de exceção)**: todas as arestas que não fazem parte de caminhos de exceção sejam cobertas pelo caminho de teste;

All-Edges-ed **(dependente de exceção)**: todas as arestas de um caminho de exceção são cobertos pelo caminho de teste;

All-Uses-ei **(independente de exceção)**: todas as associações entre definições e usos de variáveis que não fazem parte de caminhos de exceção sejam executadas pelo menos uma vez;

All-Uses-ed **(dependente de exceção)**: todas as associações entre definições e usos de variáveis que fazem parte de caminhos de exceção sejam executadas pelo menos uma vez;

All-Pot-Uses-ei **(independente de exceção)**: um caminho sem exceção possível de se alcançar a partir de uma atribuição de *v* por caminhos livres de novas definições de *v* deve ser exercido pelo menos uma vez;

All-Pot-Uses-ed **(dependente de exceção)**: um caminho de exceção possível de se alcançar a partir de uma atribuição de *v* por caminhos livres de novas definições de *v* deve ser exercido pelo menos uma vez.

Para que seja possível trabalhar com técnicas de cobertura de código Java em tempo de execução, visando a obtenção das medidas com base nos critérios citados e sem grande intrusão, é necessário trabalhar no nível de *bytecodes*. Este conceito será explicado na próxima seção.

2.4. BYTECODE JAVA

Bytecode é o nome dado ao código que é gerado após a compilação de um programa desenvolvido na linguagem Java. Ao invés da compilação do código Java gerar um código executável em um determinado sistema operacional, ela gera *bytecodes* que são interpretados pela Máquina Virtual Java (*Java Virtual Machine* - JVM) que fará a execução (AHO, 2006). A Figura 3, retirada de Medina e Fertig (2006), ilustra esse conceito.

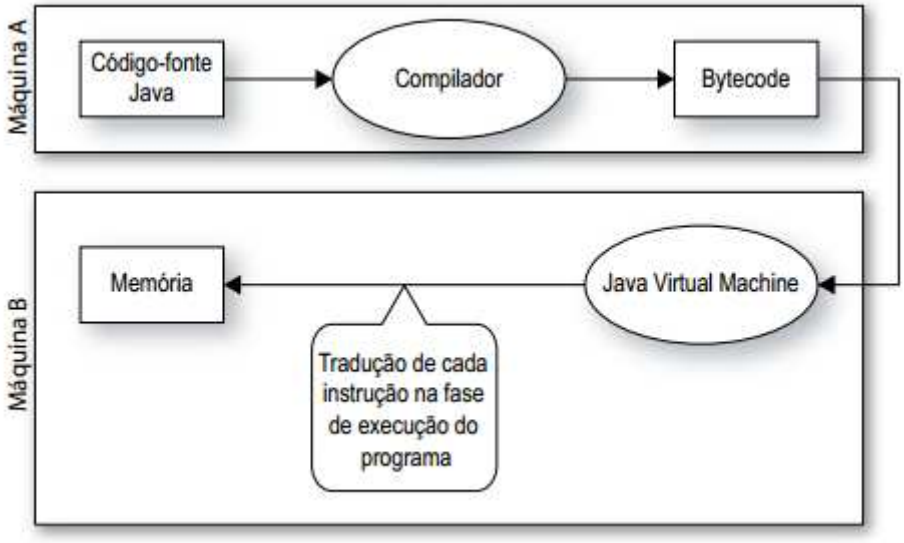

Figura 3 Execução de programa em Java (MEDINA e FERTIG, 2006).

Assim, o *bytecode* é um código intermediário entre o código-fonte e a aplicação final, característica que faz com que os programas Java sejam independentes de plataforma, ou seja, podem ser interpretados em qualquer sistema operacional que possua JVM.

A Figura 4 apresenta um exemplo de um código na linguagem Java, que corresponde ao arquivo .java e a Figura 5 traz esse código compilado, que corresponde ao arquivo com extensão *.class* formado pelos *bytecodes*.

Matematica.java

```
1  public class Matematica {
2
3      public int fibonacci(int numero) {
4          if (numero >= 0) {
5
6              return fibonacci(numero-1) + fibonacci(numero-2);
7          }
8          throw new IllegalArgumentException();
9      }
10
11 }
```

Figura 4 Exemplo de trecho código na linguagem Java

```
C:\Users\Família\workspace>javap -c Matematica
Compiled from "Matematica.java"
public class Matematica {
  public Matematica();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int fibonacci(int);
    Code:
       0: iload_1
       1: iflt          20
       4: aload_0
       5: iload_1
       6: iconst_1
       7: isub
       8: invokevirtual #2                  // Method fibonacci:(I)I
      11: aload_0
      12: iload_1
      13: iconst_2
      14: isub
      15: invokevirtual #2                  // Method fibonacci:(I)I
      18: iadd
      19: ireturn
      20: new           #3                  // class java/lang/IllegalArgumentException
      23: dup
      24: invokespecial #4                  // Method java/lang/IllegalArgumentException."<init>":()V
      27: athrow
}
```

Figura 5 *Bytecodes* correspondentes ao trecho de código da Figura 4

2.4.1. Estrutura do arquivo e formato dos comandos

Os *bytecodes* de uma classe ou de uma interface na linguagem Java armazenam informações específicas da classe. Sua estrutura básica que é uma estrutura de árvore possui os seguintes nós (BRUNETON *et al.*, 2002):

- *Constant Pool*: local onde são armazenadas as constantes. Seu objetivo é evitar a redundância, armazenando apenas uma vez os nomes das classes, métodos e valores iniciais. As instruções que as usam devem fazer referência ao índice do armazenamento;

- Descrição de campos: contém o nome e tipo dos campos e indicação se é público, privado ou estático;
- Descrição de métodos: contém o nome, os tipos de parâmetros e o valor de retorno dos métodos e indicação se é público, privado ou estático;
- Descrição de Atributos: armazena o nome do arquivo de origem na qual a classe foi compilada (S*ourceFile*), informações sobre classes internas (*InnerClasses*), atributos obsoletos (*Deprecated*) e código.

É possível visualizar o conteúdo de um arquivo Java compilado invocando o comando *javap* como a seguir:

***javap* –c NomeDaClasse**

O formato do código exemplo da Figura 5 é familiar para quem conhece o código *assembly*. Cada linha assume o seguinte formato (LINDHOLM *et al.*, 2013):

<index> <opcode> [<operand1> [<operand2>...]] [<comment>]

Onde:

<index> é o índice do identificador de *bytecode* (*opcode*), usado para transferência de controle;

<opcode> são identificadores de *bytecode* e especificam a operação a ser realizada. Tem tamanho de 1 byte, o que leva a origem do nome *bytecode*;

<operandN> são os operandos da instrução;

<comment> é a sintaxe de comentário de fim de linha, sendo opcional.

Alguns dos *opcodes* apresentados na Figura 5 e alguns dos principais *opcodes* e suas descrições estão listados na Tabela 2.

Tabela 2 - Principais *opcodes*

Descrição	***Opcodes***
Chamada de métodos - Invoca um método de acordo com o seu modificador que pode ser estático, método de uma interface, método de uma instância.	INVOKESTATIC, INVOKEINTERFACE, INVOKEVIRTUAL, INVOKESPECIAL
Retorno de métodos – as instruções de retorno de métodos se distinguem pelo tipo de valor a ser retornado (podem ser *boolean* , *byte*, *char*, *short*, *int* ou vazio)	LRETURN, FRETURN, DRETURN, ARETURN, RETURN
Mudança de fluxos condicional ou incondicional – faz com que a execução continue de acordo com o endereço especificado como parâmetro.	IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE.
Instruções aritméticas – soma, subtração, multiplicação, divisão.	IADD, LADD, FADD, DADD. SUB, LSUB, FSUB, DSUB. IMUL, LMUL, FMUL, DMUL. IDIV, LDIV, FDIV, DDIV.
Operação de carga e armazenamento – carrega ou guarda a variável local *n* na pilha.	ILOAD, ILOAD_<N>, LLOAD, LLOAD_<N>, FLOAD, FLOAD_<N>, DLOAD, DLOAD_<N>, ALOAD, ALOAD_<N>. ISTORE, ISTORE_<N>, LSTORE, LSTORE_<N>, FSTORE, FSTORE_<N>, DSTORE, DSTORE_<N>, ASTORE, ASTORE_<N>

A lista completa de identificadores de *bytecode* Java (*opcode*) e as operações que são executadas por estas instruções, podem ser encontradas em (LINDHOLM *et al.*, 2013).

2.4.2. Manipulação de *bytecodes*

Há diversas ferramentas, bibliotecas e *frameworks* no mercado para quem deseja manipular *bytecodes*. A análise e a manipulação de *bytecodes* são importantes para realizar otimização de código, análise estática e, como verificado nas ferramentas de cobertura de código, análise em tempo de execução. Algumas das principais ferramentas para manipular *bytecodes* são: ASM (2014), BCEL (2014) e Javassist (2014).

Optou-se neste trabalho por usar o *framework* ASM, uma vez que o BCEL é uma das ferramentas mais antigas para manipulação de *bytecodes* e apresenta uma sobrecarga significativa para o processador e para memória em transformações em tempo de execução e a ferramenta Javassist limita o conjunto de operações possíveis de se manipular (ASM, 2014).

O *framework* ASM foi desenvolvido para manipular e gerar dinamicamente *bytecodes* Java. Foi criado com o principal objetivo de ser o menor e mais rápido. Possui funcionalidades semelhantes a outros *frameworks* de manipulação de *bytecodes*, mas seu foco é na simplicidade de uso e no desempenho. Ele permite realizar qualquer operação de manipulação de classe. Devido a esses fatores, passou a ser reconhecido pela comunidade Java e ser usado por diversos projetos (ASM, 2014). Além disso, possui o modo *ASMifier* no *plugin Bytecode Outline* do ambiente de desenvolvimento Eclipse (ECLIPSE, 2015) para auxiliar no seu aprendizado. O *plugin Bytecode Outline* é melhor detalhado na Capítulo 5.

2.5. VULNERABILIDADES DE SEGURANÇA

Vulnerabilidade de segurança foi definida pelo *Open Web Application Security Project* (OWASP, 2013), como uma falha ou brecha no sistema que permite que atacantes o explorem mesmo que não tenham autorização para fazê-lo. São causadas, na maioria das vezes, por falhas de software. Devido a pressão para a entrega dos produtos de software e a má codificação, falhas são inseridas na implementação desses produtos, tornando as aplicações vulneráveis. Estas vulnerabilidades em aplicações *Web* que, por sua natureza são ubíquas, apresentam alta probabilidade de serem exploradas por *hackers*, causando perdas financeiras para as empresas (FONSECA e VIEIRA, 2008).

O trabalho desenvolvido pelo OWASP, comunidade sem fins lucrativos, dedicada a manter aplicações confiáveis, auxiliou no entendimento das vulnerabilidades de segurança existentes. O documento denominado OWASP top 10, apresenta a lista das vulnerabilidades de segurança mais críticas em aplicações *Web* baseado em conjuntos de dados de sete empresas que se especializam em segurança de aplicações (dados estes que abrangem mais de 500.000 vulnerabilidades em centenas de organizações e milhares de aplicações). O OWASP top 10 tem como objetivo aumentar a consciência sobre a segurança da aplicação e auxiliar as organizações a identificar as possíveis vulnerabilidades que podem ser exploradas por ataques (OWASP, 2013).

A última versão lançada pelo OWASP, em 2013, apresenta as seguintes vulnerabilidades como sendo as mais críticas em aplicações *Web*:

1. Injeção: falhas de injeção ocorrem quando uma informação maliciosa, que pode ser parte de um comando ou de uma consulta, é fornecida pelo atacante enganando o interpretador, fazendo com que seja liberado o acesso a informações não autorizadas.
2. Falha de autenticação e gerência de sessão: devido a essas funcionalidades não serem implementadas corretamente, o atacante pode assumir a identidade de outros usuários;
3. *Cross Site Scripting* (XSS): permite aos atacantes roubarem sessões e redirecionarem para sites maliciosos, através da execução de scripts no navegador dos usuários. Isso ocorre quando uma aplicação envia para o navegador informações fornecidas pelo usuário sem a devida validação.
4. Referência direta a objeto de forma insegura: devido à má codificação, permite o acesso e a manipulação não autorizada das referências de objetos implementados, tais como arquivos, diretórios, registro de banco de dados.
5. Configuração de segurança incorreta: erros de configuração em componente que formam o ambiente da aplicação. É necessária a configuração bem definida, implementada e preservada, o que inclui atualização de bibliotecas de códigos, visando atingir um nível adequado de segurança.
6. Exposição de dados sensíveis: dados sensíveis são informações tais como cartões de créditos, autenticação e identificação. Estes merecem proteção especial, como criptografia, para evitar crimes como o roubo de identidade e fraude do cartão de crédito.

7. Perda de nível de funções do Controle de Acesso: permite que atacantes forjem pedidos para terem acesso a funcionalidades sem a necessária permissão. A aplicação *Web* verifica o controle de acesso antes de disponibilizar a interface para o usuário, porém a mesma verificação deve ser feita no servidor para que não ocorra este problema.
8. *Cross Site Request Forgery* (CSRF): força o navegador do usuário, com informações de autenticação, a gerar solicitações que o aplicativo da *Web* vulnerável considera ser solicitação legítima.
9. Uso de componentes vulneráveis conhecidos: o uso de componentes, bibliotecas e *frameworks* sem a verificação de qualidade de código podem inserir uma gama de possíveis vulnerabilidades na aplicação.
10. Redirecionamentos e encaminhamentos inválidos: devido a não validação de parâmetros no redirecionamento e encaminhamento de usuários para outras páginas e sites, os atacantes podem redirecionar os usuários para sites maliciosos ou encaminhar para páginas não autorizadas.

Para que essas e outras vulnerabilidades possam ser tratadas, ferramentas detectoras de vulnerabilidades são disponibilizadas para desenvolvedores e profissionais de teste.

2.5.1. Ferramentas de detecção de vulnerabilidades

Conforme descrito anteriormente, há duas abordagens principais para testar aplicações de software, os testes caixa branca e os testes caixa preta. Para ambas as abordagens, existem ferramentas de detecção de vulnerabilidades que executam suas funcionalidades de acordo com essas duas abordagens.

Na técnica Caixa Branca, os casos de teste são derivados a partir da estrutura de controle que compõe o código fonte de componentes (PRESSMAN, 2011). A análise pode ser feita manualmente ou através de uma ferramenta denominada analisador de código fonte. Esta ferramenta analisa o código sem executá-lo, procurando por padrões de código que podem levar a vulnerabilidade e também buscam possíveis falhas de codificação (ANTUNES *et al.,* 2009).

Está técnica possui como ponto fraco a dificuldade de se ter disponível o código fonte e também a dificuldade de encontrar as vulnerabilidades de segurança devido à complexidade do código fonte (FONSECA *et al.,* 2007).

A Tabela 3 apresenta alguns exemplos das ferramentas que analisam o código fonte:

Tabela 3 - Relação de Ferramentas de Análise de Código

Nome	Descrição
Coverity Security Advisor	Ferramenta voltada para aplicações *Web* desenvolvidas na linguagem Java. Identifica erros de injeção, como injeção de SQL e *cross-site scripting* (XSS). Dá orientações para solucionar problemas de segurança de forma rápida e eficiente sem a necessidade de desenvolvedores experientes no assunto (COVERITY, 2013).
IBM Security AppScan Source	Identifica vulnerabilidades em código fonte de aplicações *Web* e aplicações móveis, incluindo JavaScript, Java e Objective- C. Usado durante o ciclo de desenvolvimento de software utilizando as melhores práticas de segurança (IBM, 2013).
HP Fortify Static Code Analyzer	Identifica as causas das vulnerabilidades de segurança priorizando o resultado e dá orientação do código correto para preencher as lacunas de segurança (HP, 2013).
FindBugs	Ferramenta *open source* que usa a análise estática para procurar por erros em código desenvolvidos na linguagem Java. Pode encontrar em torno de 300 padrões de erros (FINDBUGS, 2016).

Já no Teste Caixa Preta, os *scanners* de vulnerabilidades que utilizam esta abordagem são considerados de mais fácil aplicação por conseguirem executar centenas de testes de maneira automática, evitando a tarefa repetitiva de testes manuais (ANTUNES *et al.,* 2009). Estes *scanners* possuem três etapas, que são (FONSECA *et al.,* 2007):

i. Configuração - nesta etapa são definidas a *Uniform Resource Locator* (URL) da aplicação e as configurações de parâmetros necessários;

ii. Rastreamento - nesta etapa é produzido um mapa com a estrutura interna do software. É feita uma busca por todas as páginas *Web* da aplicação. Caso alguma página não seja encontrada, a execução de alguns casos de teste pode ser afetada;

iii. Varredura - etapa em que os testes são realizados. As funções do navegador do usuário são simuladas automaticamente pelo *scanner*, como cliques em *links* e preenchimento de formulários. As requisições e respostas geradas são analisadas para que no final seja exibido um relatório que apresente as vulnerabilidades detectadas naquela aplicação.

Trabalhos anteriores (BASSO, 2010; FONSECA *et al.,* 2007) mostram que diferentes *scanners* apresentam resultados bastante diferentes, o que indica que o uso somente de um deles pode não ser suficiente. Além disso, estes estudos também apontam que os *scanners* apresentam baixa cobertura e um alto número de falsos positivos, o que reduz a confiança nestas ferramentas.

Na Tabela 4 são apresentadas algumas ferramentas que trabalham desta forma.

Tabela 4 - Relação de Ferramentas Caixa Preta.

Nome	**Descrição**
Acunetix Web Vulnerability Scanner	Realiza testes avançados na detecção de vulnerabilidades de injeção SQL e XSS. Rastreia e analisa sites de conteúdo flash, SOAP e AJAX (ACUNETIX, 2013).
Burp Suite	Intercepta e inspeciona o tráfego entre o navegador e o aplicativo de destino. Realiza ataques para detectar vários tipos de vulnerabilidades (PORTSWIGGER, 2013).
HP WebInspect	Testa aplicativos da Web desde o desenvolvimento até a produção. Identifica as vulnerabilidades e dá detalhes das linhas do código da análise realizada em tempo real (HP, 2013).
Sign-WS	Ferramenta que utiliza assinatura de ataque e monitoramento de interface para detectar vulnerabilidade de injeção SQL, apresentando bons resultados em cobertura de detecção e falsos positivos.(ANTUNES e VIEIRA, 2011).

RAD-WS	Possui duas fases: Aprendizado e Ataque. Na primeira fase exercita o serviço sob teste afim de aprender os comandos SQL/Xpath, a segunda fase injeta ataques, permitindo assim detectar as vulnerabilidades existentes, combinando os comandos previamente aprendidos com os executados na fase de ataque. (ANTUNES *et al.,* 2009).

A próxima seção é dedicada a explicar o método usado para analisar os resultados dos estudos de caso usados no presente trabalho.

2.6. ANÁLISE DE CORRELAÇÃO E REGRESSÃO LINEAR

Para medir o relacionamento entre as variáveis que foram utilizadas como medida nos experimentos feitos no âmbito desta pesquisa foi utilizada a análise de correlação simples (BONETT e WRIGHT, 2000), calculando o coeficiente de correlação de Pearson (*r*) entre cada uma das variáveis e também o declive (*m*) da linha ajustada da regressão linear (método dos mínimos quadrados).

O coeficiente da correlação de Pearson é um método amplamente utilizado para se medir o grau de correlação linear entre duas variáveis. Desenvolvido por Karl Pearson no final do século XIX, este coeficiente varia entre -1 e +1, sendo que +1 indica uma correlação linear positiva perfeita, 0 indica que não há correlação entre as variáveis e -1 indica uma correlação negativa perfeita.

Se há duas varáveis x e y, então o coeficiente de correlação linear (*r*) das observações de uma amostra é dado por (SILVA, 1998):

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Onde (x_i,y_i) denota as coordenadas da i-ésima observação, $\overline{X}$ sendo a média de uma amostra de x_i valores, $\overline{Y}$ é a média de uma amostra de y_i valores e *n* o tamanho das amostras (SILVA, 1998).

O diagrama de dispersão é o método gráfico que marca os valores de duas variáveis, levando a uma representação inicial de seu relacionamento. O diagrama ajuda a decidir como uma variável controlável pode influenciar uma característica do resultado. A Figura 6 apresenta três gráficos de duas variáveis com diferentes graus de intensidade de relacionamento linear. Se todos os pontos desses diagramas parecem cair nas proximidades de uma reta, a correlação é denominada linear.

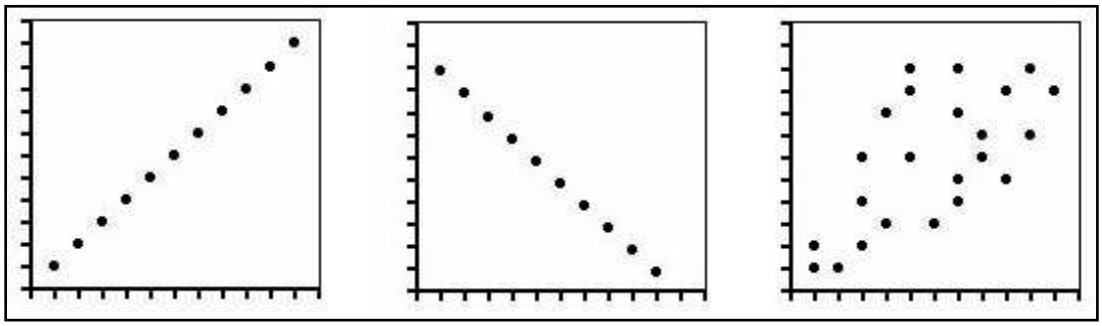

Figura 6 Exemplos de gráficos de dispersão indicando diferentes graus de dispersão

Na Figura 6 o primeiro gráfico indica correlação linear positiva perfeita, o segundo indica correlação linear negativa perfeita e o terceiro indica correlação linear fraca.

Se um diagrama de dispersão sugere uma relação linear, é de interesse representar este padrão através de uma reta. Usa-se o Método dos Mínimos Quadrados (MMMQ) para encontrar o melhor ajuste em um gráfico de dispersão minimizando a soma dos quadrados das diferenças entre os valores estimados e os observados (BONETT e WRIGHT, 2000). A reta de mínimo quadrado tem a equação (SPIEGEL, 1994):

$$Y = a_0 + a_1X$$

Onde dados um conjunto de pontos (X_1, Y_1) $(X_2\ Y_2)$, ..., $(X_n\ Y_n)$ dessa reta, as constantes a_0 e a_1 podem ser determinadas.A ideia é descobrir quais são os valores dos coeficientes a_0 e a_1,

de forma que a soma dos quadrados das distâncias seja menor possível, por isso o nome Método dos Mínimos Quadrados.

O capítulo a seguir mostra os trabalhos que serviram como base para a presente dissertação, ela foi subdivida em duas subseções que abrangem: Scanners de Vulnerabilidades e Cobertura de Código.

3. TRABALHOS RELACIONADOS

Aqui são apresentados os trabalhos que de alguma forma apresentam assuntos relacionados ao tema proposto. Este capítulo está separado em dois temas: a seção 3.1 apresenta os trabalhos sobre *Scanners* de Vulnerabilidades, onde são apresentados diversos estudos que avaliaram a eficácia dessas ferramentas; a seção 3.2 versa sobre Cobertura de Código, onde são apresentados estudos de ferramentas e métricas de cobertura de código.

3.1. *SCANNERS* DE VULNERABILIDADES

Há várias abordagens disponíveis para realizar a detecção de vulnerabilidades (STUTTARD e PINTO, 2007), incluindo testes de penetração, inspeções e avaliações de software, análise estática, análise dinâmica e detecção de anomalia em tempo de execução. Algumas destas técnicas estão disponíveis sob a forma de ferramentas automatizadas. Neste trabalho estamos particularmente interessados em técnicas baseadas em teste.

Ferramentas de teste de penetração são bem conhecidas por permitirem testar aplicações por questões de segurança (STUTTARD e PINTO, 2007). Essas ferramentas fornecem uma maneira automática para buscar vulnerabilidades, evitando centenas ou mesmo milhares de testes manuais para cada tipo de vulnerabilidade. A maioria destas ferramentas são comerciais, mas há também algumas versões gratuitas com uso limitado, restringindo a maioria das funcionalidades das ferramentas comerciais.

Em Antunes e Vieira (2011) foi proposta uma nova abordagem para detectar vulnerabilidades de injeção em *Web Services* utilizando-se de técnicas de caixa preta. A ferramenta protótipo, denominada *Attack Signature and Interface Monitoring* (Sign-WS) foi implementada com o objetivo de melhorar o processo de teste de penetração visando a detecção de vulnerabilidades de injeção SQL em SOAP *Web Services*. Essa nova abordagem proposta por Antunes e Vieira (2011) é baseada em teste de penetração com Assinaturas de Ataques e Monitoramento de Interface (IM).

Assinaturas de Ataques, no contexto deste trabalho, são sinais introduzidos em comandos SQL maliciosos, de tal maneira que se o ataque de injeção SQL for bem-sucedido, este sinal é

observado na interface do serviço que é monitorada pelo módulo *Interface Monitoring* (IM) (ver Figura 7), como parte do comando que está sendo executado. A Figura 7, retirada de Antunes e Vieira (2011), apresenta um resumo do seu processamento.

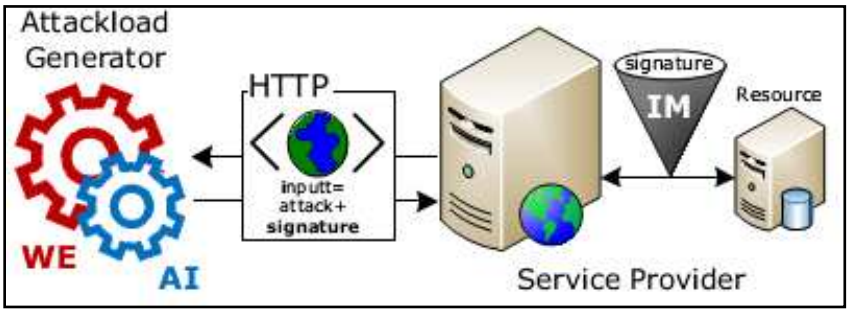

Figura 7 Ferramenta Sign-WS de (ANTUNES e VIEIRA, 2011).

Para gerar os testes é necessário informar o arquivo WSDL (*Web Services Description Language*), este arquivo define o formato das mensagens a serem trocadas pelos serviços. Através do WSDL são recuperadas a lista de operações, os parâmetros e os tipos de dados associados ao serviço. O módulo que emula o *workload* (WE) gera um conjunto de solicitações válidas para cada operação e parâmetro do *Web Service* e seleciona um subconjunto de testes. Posteriormente as solicitações são modificadas pelo injetor de ataque (AI), onde os parâmetros de entrada das operações são modificados com ataques baseados nas assinaturas de ataques. Durante todo o processo, as interfaces do *Web Service* são monitoradas (IM) para capturar o comando executado, analisar e encontrar as assinaturas que representam vulnerabilidades.

Para que as consultas sejam monitoradas o *driver* JDBC (*Java Database Connectivity*) foi modificado para interceptar os comandos SQL enviados para o servidor de banco de dados, então foi necessário referenciar a versão modificada ao invés da original para executar os testes de penetração no serviço. Essa alteração afeta o comportamento normal do serviço.

Segundo Antunes e Vieira (2011), a avaliação experimental consistiu em 21 serviços, com 80 operações, em um total de 158 vulnerabilidades de injeção SQL. A ferramenta desenvolvida foi executada e seus resultados foram comparados com outros três detectores de vulnerabilidades baseados em testes de penetração de uso comercial. Os resultados mostraram que a ferramenta alcançou maior cobertura (74% de vulnerabilidades detectadas) e não apresentou falsos positivos. Demonstrando que o resultado obtido pela ferramenta desenvolvida

é muito melhor do que os das ferramentas comerciais, sendo assim um método mais eficaz para detectar vulnerabilidades de injeção.

Antunes *et al.* (2009) apresentaram um estudo para analisar a eficácia na detecção de vulnerabilidades de injeção SQL e XPATH em *Web Services*. Nesse trabalho também foi criada uma abordagem automática para detectar vulnerabilidades baseada em detecção de anomalia e que inclui dois passos principais, que serão melhor descritos a seguir. A ideia é executar um *workload* da aplicação sem injetar falhas, interceptando os comandos SQL/XPATH emitidos pela aplicação, removendo a parte que varia. Dessa forma, armazena-se a parte que não varia, aprendendo os comandos executados, motivo pelo qual essa fase foi denominada de aprendizagem. Após esta etapa, a aplicação é executada com um conjunto de ataques de injeção SQL e XPATH. Para detectar as vulnerabilidades, os comandos SQL e XPATH são comparados com os valores válidos anteriormente aprendidos, fase essa denominada ataque.

A ferramenta CIVS-WS ilustrada na Figura 8, foi desenvolvida por Antunes *et al.* (2009). Combina teste de penetração e a detecção de anomalias em tempo de execução para descobrir vulnerabilidades de injeção em *Web Services*.

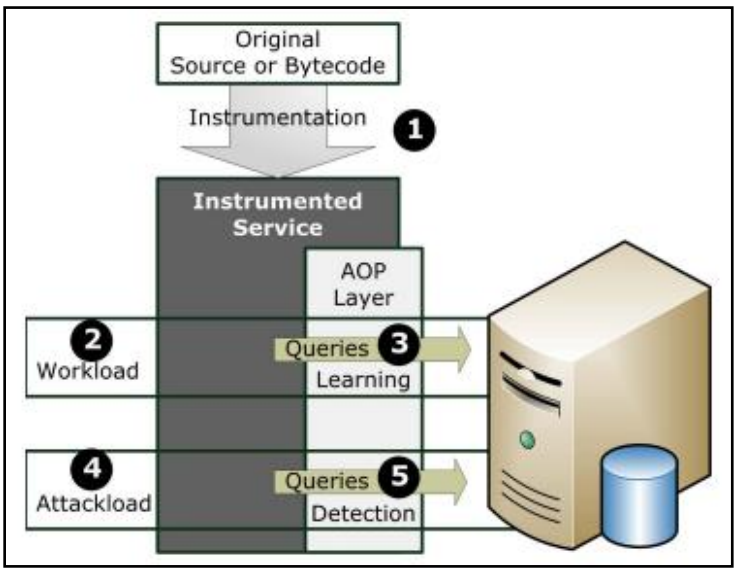

Figura 8 Ferramenta CIVS-WS (ANTUNES *et al.*, 2009)

Para demonstrar a eficácia dessa abordagem foram utilizados um conjunto pequeno de *Web Services*, apenas nove serviços. Dentre as ferramentas testadas a CIVS-WS mostrou-se mais eficaz, detectando somente vulnerabilidades existentes atingindo uma cobertura de 100% e 0% de falsos positivos. Este resultado está diretamente ligado à fase de aprendizagem, no qual a carga de trabalho gerada foi suficiente para aprender os comandos com vulnerabilidades. Mais

tarde os autores denominaram a ferramenta como *Runtime Anomaly Detection* (RAD-WS), termo este que será usado durante a dissertação.

3.2. AVALIAÇÃO DE *SCANNERS* DE VULNERABILIDADES

Estudos mostram que a eficácia das ferramentas de teste de penetração na busca por vulnerabilidades em ambientes Web é muito baixa (DOUPÉ *et al.,* 2010; ANTUNES e VIEIRA, 2014). O Teste de Penetração é uma técnica que estressa aplicações emitindo grande quantidade de requisições, e fornece a visão do comportamento do *Web Service* em tempo de execução. Normalmente, desenvolvedores confiam nas ferramentas que são baseadas em testes de penetração, mas estudos mostraram que a eficácia dessa técnica tem suas limitações (ANTUNES e VIEIRA, 2010).

FONSECA *et al.* (2007) propuseram uma forma de avaliar e comparar *scanners* de vulnerabilidades usando técnicas de injeção de falhas em aplicações Web. No trabalho foi usada a técnica *Generic Software Fault Injection Technique* (G-SWFIT) (DURÃES e MADEIRA, 2006) que se concentra em injetar os tipos mais frequentes de falhas em aplicações Web. Esta técnica reproduz no código alvo as sequências de instruções que representam tipos mais comuns de falhas de software de alto nível.

O experimento envolveu três *scanners* de vulnerabilidades e duas aplicações *Web*. Os diferentes *scanners* produziram resultados bastante diversos e todos eles deixaram uma porcentagem de vulnerabilidades não detectadas. Com a ajuda de um testador humano, foram detectadas 17 vulnerabilidades não encontradas por nenhuma ferramenta, mostrando a baixa cobertura destas ferramentas. A taxa de falsos positivos se apresentou muito alta, variando de 20% a 77% nos experimentos realizados.

No trabalho desenvolvido por Vieira *et al.* (2009) foi sugerido o uso de boas práticas de programação como forma de prevenir o comprometimento de *Web Services*. As seguintes práticas foram citadas: executar a revisão de código de segurança, executar testes de penetração e usar analisadores de vulnerabilidades. Estas formas de prevenção se enquadram no teste de caixa preta e no teste de caixa branca.

Neste mesmo trabalho foram utilizadas quatro ferramentas comerciais e 300 *Web Services* disponíveis publicamente para estudar a eficácia de *scanners* de vulnerabilidades e os tipos mais comuns de vulnerabilidades em ambientes de *Web* Services. A primeira observação realizada é de que diferentes *scanners* detectaram diferentes vulnerabilidades, e que a vulnerabilidade de injeção SQL foi a única detectada por todas as ferramentas. Somente dois *scanners* detectaram vulnerabilidades de injeção XPATH e de execução de código, apenas uma ferramenta detectou vulnerabilidades de estouro de *buffer*, falha de autenticação e divulgação do caminho do servidor. O número de falsos positivos detectados foi elevado, 35% e 40% em dois casos e a cobertura muito baixa, inferior a 18% foi observada em dois dos *scanners*, enquanto os outros dois *scanners* apresentam uma cobertura superior a 87%. Isso mostra que estas ferramentas foram implementadas de formas diferentes e que para ter uma boa cobertura o usuário deverá usar várias ferramentas em vez de apenas uma.

Embora haja muitas obras disponíveis sobre a avaliação das ferramentas de detecção de vulnerabilidades, a primeira proposta de uma abordagem de *benchmarking* para este domínio foi apresentado por Antunes e Vieira (ANTUNES e VIEIRA, 2015). Esta técnica foi demonstrada usando quatro ferramentas de teste de penetração, três analisadores de código estático e a ferramenta RAD-WS, que combina teste de penetração com detecção de anomalias em tempo de execução.

O trabalho apresentado em Antunes e Vieira (2015) propõe uma abordagem para a avaliação comparativa de ferramentas de detecção de vulnerabilidades utilizando-se das seguintes medidas:

- *Precision*: proporção do número de vulnerabilidades detectadas corretamente pelo número de todas as vulnerabilidades detectadas;
- *Recall*: proporção de vulnerabilidades detectadas corretamente pelo número de todas as vulnerabilidades existentes;
- *F-Measure*: compara a eficácia de ferramentas que implementam detecção de vulnerabilidades de diferentes formas, com base no número de vulnerabilidades relatadas para a mesma parte do código. Representa a média harmônica das medidas *Precision* e *Recall*.

As ferramentas foram executadas e as medidas foram calculadas. Os resultados mostraram que a ferramenta RAD-WS foi a que apresentou o maior *F-Measure*, seguida de duas

ferramentas analisadoras de código estático, mostrando ser uma opção melhor do que as demais ferramentas de teste de penetração que foram utilizadas nos testes. Uma análise separada das ferramentas permitiu verificar que as ferramentas de teste de penetração detectaram menos de 35% de todas as vulnerabilidades conhecidas e também apresentaram o maior número de falsos positivos. Por outro lado, uma ferramenta analisadora de código estático detectou 100% de verdadeiros positivos, mas apontou 36% de falsos positivos.

Os resultados mostram que o *benchmark* proposto pode ser facilmente usado para avaliar, comparar e classificar os testadores de penetração, analisadores de código estático e detectores de anomalias. O problema é que o sucesso da abordagem depende da qualidade das cargas de trabalho utilizadas e do que é considerado valor de referência das vulnerabilidades existentes. Uma maneira possível de superar essas limitações é compreender a relação entre a cobertura de código e as saídas das ferramentas de detecção de vulnerabilidade, proporcionando novas formas de avaliar e comparar as técnicas de detecção de vulnerabilidade.

Outro trabalho que corrobora com os artigos já apresentados é o desenvolvido por Basso *et al.* (2010), no qual foi proposta uma nova abordagem para avaliar *scanners* de vulnerabilidade usando técnicas de injeção de falhas e modelos de árvores de ataque. Foi analisado o efeito de vulnerabilidades de segurança na presença de falhas de software injetadas no código-fonte de aplicações, estudo esse que auxilia na rápida detecção de falhas de segurança permitindo eliminar ou reduzir a gravidade da exploração aumentando a dependabilidade dos sistemas sob teste. O modelo de árvore de ataque foi utilizado para guiar os ataques à aplicação, facilitando a análise de segurança. Nas árvores de ataque o nó raiz representa o objetivo final do atacante e os nós filhos são sub-objetivos que precisam ser alcançados para que o nó pai tenha sucesso. No trabalho de Basso *et al.* (2010) as árvores de ataque foram usadas para descrever a possibilidade de atacar três tipos de vulnerabilidades: *SQL Injection*, *Cross-Site Scripting* (XSS), e *Cross-Site Request Forgery* (CSRF).

Os testes começam com uma *Gold Run*, onde vulnerabilidades já podem ser detectadas. Os resultados da *Gold Run* são coletados pelo *scanner* para serem comparados com os resultados de quando as falhas são injetadas, servindo de referência para verificar o efeito das novas falhas injetadas. Após a injeção, o *scanner* é executado novamente na aplicação para avaliar se houve alterações. As diferenças do *Gold Run* e da execução com injeção de falhas são exploradas com o apoio da árvore de ataque, para verificar se a nova vulnerabilidade realmente existe ou se é um

falso positivo. Os mesmos cenários são executados na aplicação original (sem falhas) para verificar se a vulnerabilidade já existia, indicando falta de cobertura. Os resultados mostraram que falhas injetadas afetaram o comportamento das aplicações e também dos *scanners*, que apresentaram baixa cobertura e alto índice de falsos positivos. Além da injeção de falhas, foi observado que as tecnologias utilizadas pelas aplicações também afetam os resultados obtidos pelo *scanner*.

Em Doupé *et al.* (2010) foi realizada outra avaliação de scanners de vulnerabilidades web que utilizam a técnica caixa preta. Ao todo onze scanners comerciais e *open source* foram avaliados. A avaliação foi composta por diferentes tipos de vulnerabilidades em aplicações web realistas com diferentes desafios para as ferramentas.

Os resultados da avaliação mostraram que a varredura é uma tarefa crítica e fundamental para o sucesso do *scanner* assim como a técnica de detecção de vulnerabilidades, e que muitas classes de vulnerabilidades são completamente ignoradas por essas ferramentas. Sendo assim, conclui-se que são necessárias pesquisas para melhorar a detecção dos *scanners* de vulnerabilidades.

3.3.COBERTURA DE CÓDIGO

Trabalhos anteriores mostraram que o relacionamento entre cobertura de testes e confiabilidade não é novo, incluindo algumas vantagens desta relação para avançar as técnicas de teste.

Em Horgan *et al.* (1994), os autores descreveram a utilização do ATAC (*Automatic Test Analysis for C3*), uma ferramenta de cobertura de teste para programas em C que mede cobertura de blocos, decisão, caminho, *c-use*, *p-use*, *all-uses* e *du-path*. Dois programas foram usados como estudo de caso e os resultados mostraram que a análise de cobertura de código durante os testes forneceu uma métrica de qualidade de teste, permitindo alcançar maior confiabilidade devido à compreensão da estrutura do programa.

Yang *et al.* (2009) apresentaram um estudo em que comparam 17 diferentes ferramentas de cobertura e as características que elas oferecem. Além disso, o estudo visou avaliar a ferramenta que desenvolveram, a *eXtreme Visual-Aid Novel Testing And Generation*

(eXVantage), que é um conjunto de ferramentas para cobertura de código, depuração, desempenho e emissão de relatórios.

As ferramentas sob estudo foram comparadas de acordo com três características:

i. Linguagens de programação: linguagem de programação que a ferramenta suporta. Das ferramentas analisadas a maioria tem suporte a linguagem C/C++, Java, ou ambas, e poucas tem suporte a COBOL, PHP e .Net;

ii. Instrumentação de programas: a instrumentação de código-fonte é a mais usada pelas ferramentas alvo deste estudo. Outro tipo de instrumentação bastante utilizado para realizar a cobertura é a instrumentação de *bytecodes*. A eXVantage usa instrumentação de código-fonte para C/C++ e instrumentação de *bytecode* Java;

iii. Características adicionais: além da cobertura de código, características como depuração, rastreamento da causa de defeitos, identificação de partes fortemente executadas também foram encontradas nas ferramentas estudadas. A ferramenta eXVantage gera um relatório da falhas detectadas.

Também foram realizadas comparações em relação a capacidade de identificar e priorizar módulos/classes/métodos críticos; em gerar testes automáticos e relatórios; e sobre os critérios de medição de cobertura que usam. Verificou-se que os níveis de medição de cobertura encontrados nas ferramentas sob estudo são: Comando/Linha/Bloco, Ramificação, Decisão, Método e Classe.

A ferramenta eXVantage se diferencia das outras ferramentas por gerar automaticamente casos de teste alcançando pontos críticos do programa. Por fim, foi verificado que cada ferramenta tem seus prós e contras, deixando claro que, não existe uma única ferramenta adequada para todos os sistemas. A melhor ferramenta para determinado sistema vai depender das especificidades do sistema em questão. A comparação realizada ajuda na seleção de uma ferramenta de cobertura apropriada para as necessidades e ambiente da aplicação.

O trabalho desenvolvido por Vincenzi *et al.* (2003) também descreve uma ferramenta para auxiliar no processo de teste de software. A ferramenta apresentada é denominada *Java Bytecode Understanding and Testing* (JaBUTi) que foi desenvolvida para análise de cobertura

em programas desenvolvidos em Java, porém sem a necessidade do código-fonte para executar suas atividades. Para conseguir trabalhar com o código executável foi necessário utilizar a biblioteca de manipulação de *bytecodes* BCEL.

A ferramenta avalia a cobertura de acordo com os critérios de fluxo de controle e fluxo de dados. As principais funcionalidades que essa ferramenta realiza são: instrumenta e apresenta os *bytecodes* da classe sob teste; coleta dados da execução dinâmica durante a execução das classes; gera relatórios a respeito de critérios, classes, métodos e casos de teste; atribui pesos aos requisitos de teste, fornecendo sugestões para orientar os testadores a aumentar a cobertura de uma forma eficaz. A JaBUTi também apresenta uma interface gráfica com boa usabilidade, onde os *bytecodes*, resultados e os grafos *def-use* são exibidos.

Muitas abordagens utilizadas para estimar se uma unidade de código foi suficientemente testada são baseadas em cobertura de código, que mede quais fragmentos de código foram exercitados pelo teste (VANOVERBERGHE *et al.,* 2012). A cobertura de código permite identificar qual a porcentagem de código que foi verificada em determinado conjunto de testes executado. Entretanto, a cobertura de código não mede até que ponto o conjunto de testes verificou alguns pontos da funcionalidade e acaba por não ser autossuficiente. Por esse motivo, Vanoverberghe *et al.* (2012) propôs a abordagem de cobertura por estado (*state coverage*).

Essa métrica mede a taxa de atualizações de estado que são lidas por comandos em relação ao número total de atualizações de estados. Outra contribuição do autor foi a apresentação de algoritmos que medem a cobertura por estado. Essa abordagem é baseada na hipótese de que toda atualização para o estado de execução deve, eventualmente, ser seguido por um comando que lê o valor atualizado.

Assim como a cobertura por código, a cobertura por estado é simples de entender, eficiente e fácil de assimilar. Durante uma análise preliminar da abordagem, em bibliotecas *open-source*, a cobertura por estado ajudou a identificar inúmeras propriedades não verificadas e auxiliou na detecção de erros.

Skruch (2011) desenvolveu uma métrica de cobertura para quantificar a completude de testes do tipo caixa-preta em sistemas dinâmicos de tempo contínuo. O propósito da métrica é assegurar ao engenheiro de testes que um dado conjunto de testes é suficiente e indicar onde testes adicionais são necessários. O desafio é viabilizar a criação de um conjunto de testes que avalie adequadamente o comportamento esperado de um sistema dinâmico de tempo contínuo.

A adequação do conjunto de testes é inferida através de análises realizadas por métricas de cobertura. Entretanto, atualmente, não existem métricas padrões para avaliar a completude de testes do tipo caixa-preta em sistemas dinâmicos de tempo contínuo, que são definidos por equações diferenciais ordinárias. Esses sistemas são usualmente representados por um formato de modelo de espaço e de estado, que consiste em um conjunto de entradas, saídas e variáveis de estado relacionadas à álgebra e equações diferenciais.

A métrica, baseada em um modelo de espaço de estado do sistema que está sendo testado, foi denominada *state space coverage* (SKRUCH, 2011). Segundo o autor, a criação da métrica veio para suprir a necessidade de quantificar a cobertura de sistemas dinâmicos, que utilizam equações diferenciais e, usualmente têm um número infinito de estado, sendo as métricas clássicas (como comando, ramificação, caminho médio) não apropriadas para esse tipo específico de sistema.

A solução proposta pelo autor foi testada em um controlador de nível de combustível, e demonstrou eficácia em testes de regressão e de conformidade. A utilização da abordagem é independente de plataforma ou linguagem de programação e apresenta fácil implementação.

Alguns estudos comprovam que o teste de cobertura é um bom indicador quando o objetivo é detectar falhas em casos de teste (FRANKL e WEYUKER, 1988; HORGAN *et al.,* 1994). Esses mesmos estudos comprovam que uma alta cobertura de código traz um software altamente confiável e com baixas taxas de falhas. Esses experimentos mostram que cobertura de código e detecção de falhas em programas crescem ao longo do tempo, conforme os testes vão progredindo.

Mesmo com a relação observada entre cobertura de código e detecção de falhas, Cai e Lyu (2005) levantaram a seguinte questão: Pode este fenômeno do crescimento simultâneo ser atribuída a uma dependência causal entre a cobertura de código e detecção de falhas, ou é apenas uma coincidência, devido à natureza cumulativa de ambas as medidas? Para responder a essa questão de pesquisa, Cai e Lyu (2005) realizaram um estudo para comprovar que o efeito do teste de cobertura na detecção de falhas depende do perfil do teste realizado.

A hipótese dos autores é que a efetividade de detecção de falhas na cobertura de código varia e depende dos diferentes perfis de teste. Para avaliar o desempenho da cobertura de código os autores empregaram dois métodos: Teste de Cobertura e Teste de Mutação. Foi investigada a

relação entre cobertura de código e capacidade de detecção de falhas sob diferentes perfis de teste. O estudo realizado mostrou que a correlação entre as duas medidas é maior em casos de testes excepcionais e, menor em casos de testes normais.

Em Crespo *et al.* (1997) foi realizado um experimento para verificar a relação entre cobertura de código e a confiabilidade do software. Além de outros critérios, o estudo utiliza "*Potencial-Uses*", que os autores consideram como sendo mais forte do que os critérios restantes e os critérios *All-Nodes* e *All-Edges* considerados mais fracos.

Para realizar o experimento foi selecionado o software "*space*", que calcula e fornece dados de entrada para um sistema de cálculo de distribuição física de antenas. Vinte mil dados de teste, de acordo com o perfil operacional do usuário, foram gerados para detectar falhas e remove-las. Algumas das relações investigadas e os resultados obtidos através do experimento foram:

i. Relação da cobertura de código e defeitos removidos: foi observado o crescimento da cobertura com a remoção de defeitos. No critério *All-Nodes*, o mais fraco, a cobertura atingiu 71,78% contra 25,22% do critério PDU, o mais forte;
ii. Relação da cobertura do código e dados de teste: houve um crescimento acentuado na cobertura do código para os primeiros dados de teste, mas após o 108º dado de teste a cobertura cresceu pouco, tendendo a estabilização;
iii. Relação da cobertura de código e confiabilidade do software: observou-se que existe uma relação entre a cobertura que contribui para estimar a confiabilidade.

Ficou comprovado pelo experimento que existe uma relação entre medida de cobertura e a confiabilidade.

O estudo desenvolvido por McCabe (2015) combinou as técnicas Fuzz e McCabe IQ. A técnica de caixa preta *Fuzzing* visa descobrir de forma rápida e a baixo custo vulnerabilidades de segurança. McCabe IQ permite alocar recursos de teste de forma eficiente e eficaz. Com a aplicação dessa técnica aos testes *fuzz*, é possível identificar os módulos com mais erros e os caminhos não testados no código.

Como resultado desta combinação foi possível ganhar mais confiança nos testes e no software, aumentando a cobertura de código e a extensão de alcance dos testes *fuzz*.

Em DOUPÉ *et al.* (2012) a cobertura de código foi utilizada como parte de uma avaliação que propõe um *scanner* de vulnerabilidades para aplicações Web, através da técnica de caixa preta. Foi proposta uma nova maneira de inferir o estado interno de aplicações Web, ao navegar pelas aplicações e observar as suas saídas produzindo um modelo. Os experimentos realizados em DOUPÉ *et al.* (2012) envolveram três diferentes *scanners* de vulnerabilidade contra o *scanner* de vulnerabilidade proposto pelos autores. Os resultados mostraram que a ferramenta proposta é capaz de exercitar mais código das aplicações e também descobrir vulnerabilidades que outros *scanners* de vulnerabilidade não encontraram. No entanto, a métrica utilizada no estudo é equivalente a cobertura de linha (*All-Nodes*), que como visto anteriormente, é um critério bastante fraco.

Tendo como foco a segurança, o trabalho de Halfond e Orso (2006) propõe um novo critério de cobertura de comandos (*command-form*) que verifica todas as formas de interações entre uma aplicação e seu banco de dados, a fim de detectar vulnerabilidades do tipo Injeção SQL em consultas geradas dinamicamente. Este critério corresponde ao número de formas de comandos cobertos dividido pelo número total de possíveis formas de comandos.

Neste estudo foi apresentado o protótipo DITTO (*Database Interaction Testing TOol*), no qual é necessário que testadores criem conjuntos de casos de testes para a aplicação sob teste, assim o protótipo mede a adequação de um conjunto de testes com respeito ao critério de comandos, e fornece *feedback* da cobertura aos testadores. Foram apresentados dois estudos de caso preliminares que mostraram que o DITTO pode realizar medições de cobertura para aplicações de banco de dados reais.

O trabalho de Smith *et al.* (2008) propõe duas métricas de cobertura específicas para injeção SQL: cobertura de instruções destino, que é a porcentagem de instruções SQL que são executadas em pelo menos um teste; e cobertura de variáveis de entrada, que é a porcentagem de variáveis de entrada usadas em pelo menos um teste. No contexto de injeção SQL são as variáveis enviadas para o sistema de gerenciamento de banco de dados. Para determinar se é possível calcular valores para as duas métricas do estudo, foi realizado um estudo de caso usando a ferramenta *iTrust*, uma aplicação web projetada para armazenar e distribuir registros de saúde. Os resultados mostraram que a cobertura de instrução destino obteve um percentual de 96,7% e a cobertura de variável de entrada alcançou 98,5%.

Neste capítulo foi possível verificar que as ferramentas de *scanners* de vulnerabilidades apresentam baixa cobertura (BASSO, 2010; ANTUNES e VIEIRA, 2010), e que existem diversos critérios para se obter a cobertura de código. Afim de relacionar os resultados de cobertura de código e de vulnerabilidades detectadas, o presente trabalho tem como proposta incorporar informação de cobertura da análise efetuada pelo *scanner* de vulnerabilidade. A proposta de pesquisa é detalhada no próximo capítulo.

4. ABORDAGEM PROPOSTA

A análise de cobertura de código mostrou-se uma maneira eficaz para estimar a qualidade dos testes executados. Neste estudo será avaliado se esta análise também é eficaz no contexto de testes de vulnerabilidades de segurança. Para isso desenvolvemos uma abordagem que servirá como base para realizar os nossos experimentos.

No âmbito de detecção de vulnerabilidades, existem algumas características específicas que devem ser discutidas antes de detalhar os experimentos, como a efetividade das ferramentas de detecção de vulnerabilidades. Essa característica depende de dois fatores:

- Taxa de verdadeiros positivos (*true positives):* proporção entre o número de vulnerabilidades corretamente detectadas e o número total de vulnerabilidades existentes;
- Taxa de falsos positivos (*false positives*): proporção entre o número de vulnerabilidades detectadas erroneamente e o número total de vulnerabilidades detectadas.

Neste trabalho optamos por não considerar os falsos positivos, pois: 1) falsos positivos são pouco relacionados com a qualidade dos testes (ou mesmo do percentual de cobertura atingido) e mais relacionados com o processo de identificação de vulnerabilidades e as heurísticas usadas, e 2) atualmente, existem ferramentas de teste de caixa cinza que são o estado da arte para *Web Services* e evitam falsos positivos (ANTUNES *et al.,* 2009; ANTUNES e VIEIRA, 2011). Por estes motivos, decidimos focar nosso estudo na detecção de vulnerabilidade e, portanto na taxa de verdadeiros positivos.

No que diz respeito a vulnerabilidades de software, é conhecido que as diferentes linhas de uma aplicação não têm a mesma importância (SMITH *et al.,* 2008 ; ANTUNES e VIEIRA, 2015). O fato é que cada ataque tem por objetivo um determinado tipo de instrução, um *hotspot*, que corresponde a um conjunto especifico do código da aplicação que é propenso a um tipo específico de vulnerabilidade. Pensando nisso, para aumentar a especificidade da experiência para este domínio, propomos uma métrica extra inspirada em Smith *et al.* (2008), que retrata a proporção de *hotspot* da aplicação que foram executados durante os testes.

A próxima seção visa descrever a abordagem e os detalhes das métricas que foram usadas e criadas para alcançar os objetivos propostos na nossa pesquisa.

4.1. ABORDAGEM EXPERIMENTAL

Para alcançar os objetivos desta pesquisa projetou-se a abordagem apresentada na Figura 9. Através dessa abordagem pretendemos responder as perguntas da pesquisa (Q1 a Q4) apresentadas no Capítulo 1. Cada etapa desta abordagem é descrita a seguir:

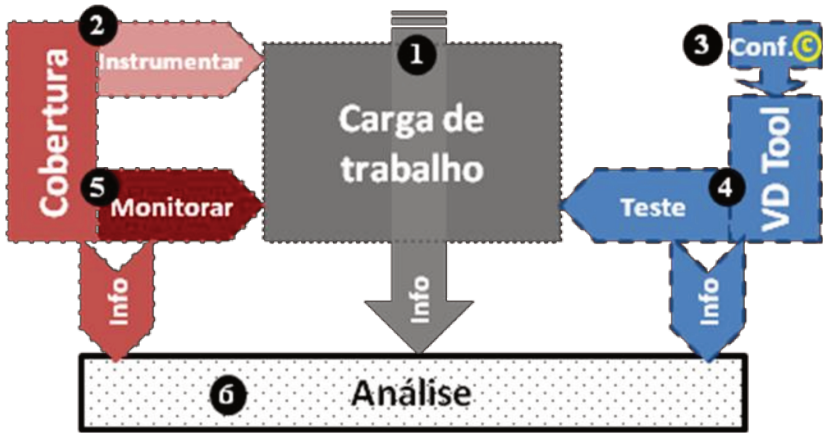

Figura 9 Abordagem experimental para avaliar a cobertura de código

1. Carga de trabalho (*workload)*: a primeira parte consiste em definir a carga de trabalho que um conjunto de aplicações deve executar. Para a nossa pesquisa são funcionalidades características de *Web Services*, com vulnerabilidades de segurança que devem ser testadas por ferramentas de detecção de vulnerabilidades.

2. Instrumentação: nesta fase é obtido o código executável dos *Web Services* que foram desenvolvidos em Java, sendo assim utilizados os arquivos .class ou .jar. A partir desses arquivos realiza-se a instrumentação a fim de inserir "sondas" que servirão para realizar a cobertura do código. Nesta fase também foram coletadas informações adicionais para realizar a medição da cobertura *hotspot*.

3. Configuração: nesta etapa é necessário definir Configurações (*C)* para ferramenta de detecção de vulnerabilidade de acordo com os testes que se deseja realizar. Para as diferentes Configurações definidas (C_1 a C_n), as etapas de 4 a 6 devem ser realizadas repetidamente.

4. Testes: esta fase é executada por uma ferramenta de detecção de vulnerabilidades que exercita a aplicação a fim de executar os testes configurados na fase anterior. Essa abordagem é flexível, o que nos permite utilizar diferentes tipos de ferramentas de detecção de vulnerabilidades. Assim, as experiências podem ser facilmente reproduzidas para outras ferramentas, modificando apenas a ferramenta detectora de vulnerabilidades.

5. Monitoramento: nesta fase, a aplicação sob teste é monitorada. Ao se executar as aplicações instrumentadas com as "sondas", que foram inseridas na segunda fase da abordagem, é gerado um arquivo *trace* contendo as informações dos trechos de código que foram executados nos testes. Também é gerado um arquivo com os *hotspots* executados durante a fase de Testes.

6. Análise: com os testes finalizados, os arquivos de *traces* e os arquivos que contém os *hotspot* são analisados para se verificar a cobertura dos testes e a cobertura *hotspot* e a qualidade dos testes realizados.

Pode-se realizar novas configurações na aplicação alvo e esse processo deve ser iniciado novamente a partir da terceira fase, até que não tenhamos mais configurações a serem executadas. Assim é possível coletar informações dos testes executados considerando diferentes configurações da ferramenta de detecção de vulnerabilidades (DV).

As próximas seções apresentam maiores detalhes com relação ao que foi realizado para avaliarmos duas métricas diferentes: Cobertura de Código e Cobertura de *Hotspots*.

4.2. COBERTURA DE CÓDIGO

A cobertura de código, como já dito anteriormente, é a proporção do código que foi executado. Diversas ferramentas disponíveis realizam esta tarefa, no entanto, muitas delas são baseadas em teste de unidade e não estão prontas para serem utilizadas para analisar a cobertura de código em aplicações durante o processo de teste por ferramentas externas, como é o caso dos detectores de vulnerabilidades.

Essa foi uma das exigências do presente estudo, pois queremos garantir que a ferramenta de teste seja independente das ferramentas de análise de cobertura e também que a abordagem,

mais tarde, possa ser perfeitamente aplicável a outras ferramentas de detecção de vulnerabilidades.

Por estes motivos, foi selecionada uma ferramenta open source JaBUTi, já existente na literatura e que atende a esses requisitos, pois implementa diferentes critérios de cobertura de código (VINCENZI *et al.*, 2003). JaBUTi está disponível para *download* e é prático de usar. Sendo de código aberto, realizamos as modificações necessárias para cumprir os objetivos propostos. Os impactos na ferramenta consistiram na inclusão de dois novos menus, como mostra a Figura 10. A versão modificada será referenciada desse ponto ao final do texto por JaBUTi*.

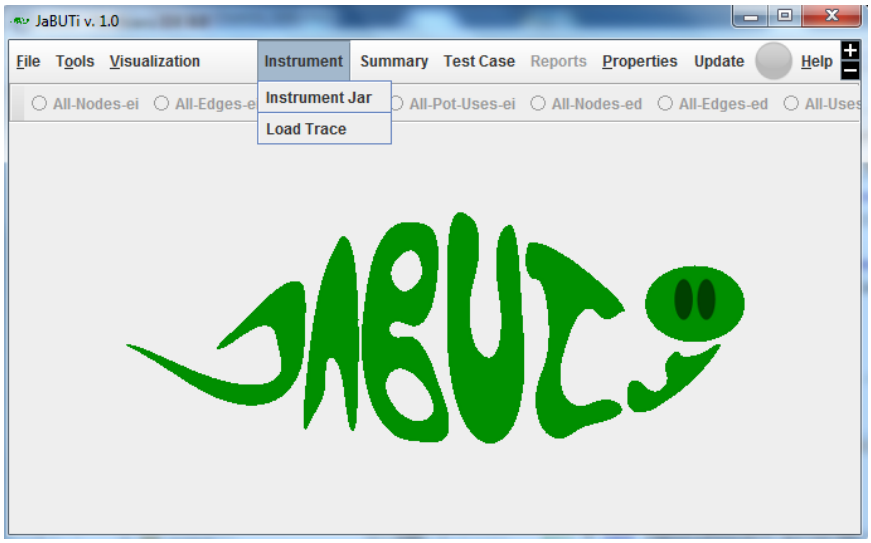

Figura 10 Ferramenta JaBUTi*

1. **Menu *Instrument Jar*** (JaBUTi* Código Instrumentado): foi criado um novo menu que permite instrumentar as classes Java de forma independente. A partir de um arquivo *.jar* ou *.class* (código compilado Java), é gerado um arquivo instrumentado no qual são inseridas "sondas" nas linhas do código compilado. A Figura 11 apresenta um trecho do arquivo compilado do Java, que representa um serviço que será testado pela ferramenta de detecção de vulnerabilidades. A Figura 12 apresenta o mesmo arquivo após a instrumentação, com a inserção das "sondas". É possível verificar os arquivos completos nos Anexos 1 e 2, respectivamente. Ao executar casos de testes na aplicação instrumentada um arquivo *trace* é criado com informações do código executado (ver Figura 17: JaBUTi * Código Instrumentado);

```
List<Book> vec = new ArrayList();
Connection con = Database.pickConnection();
try
{
  PreparedStatement statement = con.prepareStatement("SELECT * FROM tpcw_item, tpcw_author WHERE
  tpcw_item.i_a_id = tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM <= 50 ORDER BY
  tpcw_item.i_title ");

  statement.setString(1, search_key);
  ResultSet rs = statement.executeQuery();
  while (rs.next()) {
    vec.add(new Book(rs));
  }
  rs.close();
  statement.close();
  con.commit();
}
```

Figura 11 Exemplo de código compilado de um serviço

```
long l = DefaultProber.getNest();
DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "0");
List vec = new ArrayList();
Connection con = Database.pickConnection();
try
{
  DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "12");
  PreparedStatement statement = con.prepareStatement("SELECT * FROM tpcw_item, tpcw_author "
  + " WHERE tpcw_item.i_a_id = tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM <= 50 "
  + " ORDER BY tpcw_item.i_title ");

  statement.setString(1, search_key);
  ResultSet rs = statement.executeQuery();
  for (;;)
  {
    DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "40");
    if (!rs.next()) {
      break;
    }
    DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "50");
    vec.add(new Book(rs));
  }
  DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "69");rs.close();
  statement.close();
  con.commit();



  DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "89");
  Database.relaseConnection(con);
}
```

Figura 12 Exemplo de código do serviço após a instrumentação

2. **Arquivo *Trace*:** foi modificada a lógica das "sondas" para gravar os arquivos de *trace* em um local do sistema pré-definido. A Figura 13 apresenta um exemplo de um trecho do arquivo *trace* gerado;

```
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch1997889433:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch:0:0:0
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch1997889433:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch:0:0:4
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch1997889433:pt.uc.dei.wsvdbench.tpcw.DoSubjectSearch:1:48457605:0
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:0:0:0
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:0:0:4
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:1:48457606:0
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:1:48457607:0
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:1:48457607:16
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:1:48457607:25
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:1:48457607:26
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:1:48457606:12
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:1:48457606:96
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:1:48457606:98
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:0
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:4
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:28
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:37
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:69
:Thread[http-127.0.0.1-80-3,5,jboss]:STATIC:pt.uc.dei.wsvdbench.util.Database:2:48457608:166
:Thread[http-127.0.0.1-80-3,5,jboss]:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0473395939:pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0:1:48457606:114
```

Figura 13 *Trace* gerado durante a execução dos testes.

3. **Menu *Load Trace*** (JaBUTi * Analisador *Trace*): este menu é responsável por interpretar o arquivo de *trace* gerado durante a execução dos testes (ver Figura 17: JaBUTi * Analisador *Trace*). Ao final, o analisador da JaBUTi * permite o cálculo das métricas de cobertura para os critérios de teste de fluxo de controle (*All-Nodes-ei, All-Edges-ed, All-Edges-ei, All-Nodes-ed*) e critérios de fluxo de dados (*All-Uses-ei, All-Uses-ed, All-Pot-Uses-ei, All-Pot-Uses-ed*).

A partir do arquivo *trace* que foi gerado e interpretado é possível acessar o Menu *Summary* (ver Figura 10) e escolher entre os submenus: cobertura por Critério, Classe ou Método. A partir dos submenus são exibidos relatórios referentes à cobertura do código. Estas funcionalidades não sofreram alterações e foram reutilizadas para gerar a cobertura de código. A Figura 14 ilustra o relatório exibido ao acessar o submenu de cobertura por Critério.

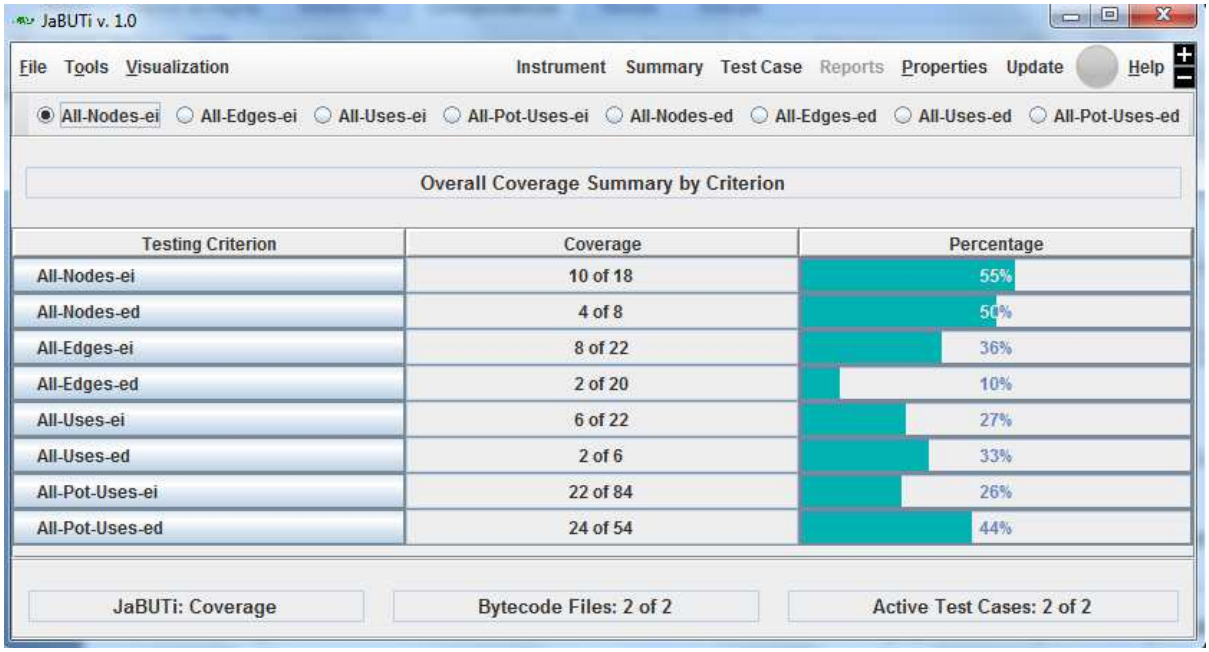

Figura 14 Relatório de cobertura de código por critério da ferramenta JaBUTi

Analisando a Figura 14 conclui-se que para o arquivo de *trace* interpretado a cobertura para o critério *All-Nodes-ei* foi de 10 nós em 18, para o critério *All-Nodes-ed* foi de 4 nós para 8 e assim por diante.

4.3. COBERTURA *HOTSPOT*

Além da cobertura de código, foi criada outra forma de analisar a cobertura, que chamaremos de Cobertura *Hotspot*. Esta métrica de cobertura *hotspot* foi inspirada na cobertura de instrução (*statement coverage*) destino (SMITH *et al.,* 2008), citada na Seção 3.2, porém é mais genérica, já que um *hotspot* pode ser um conjunto de instruções (nó) que pode exigir um determinado número de execuções, a fim de ser considerado coberto.

A **Cobertura *Hotspot*** nada mais é do que a porcentagem das possíveis vulnerabilidades existentes que a ferramenta detecta. Não há ferramentas disponíveis para realizar a análise da cobertura *hotspot*. Assim foi desenvolvida uma solução específica adaptada apenas para vulnerabilidades de injeção *SQL* e para os respectivos *hotspot*: declarações que executam comandos SQL no banco de dados.

Essa solução tem duas fases principais, a Pré-análise e a Análise:

1. **Pré-análise**: foi desenvolvido o "Procurar *Hotspot*" (ver Figura 17) que é executado durante a instrumentação. Esta fase usa o código compilado Java da aplicação alvo (.jar), que contém os *bytecodes* Java. Para interpretar as classes compiladas foi utilizada a biblioteca ASM, "esta biblioteca permite analisar e manipular os *bytecodes* Java"[1]. Assim, os *bytecodes* serão analisados em busca de trechos do código que representem comandos SQL, pois estes podem conter potenciais vulnerabilidades do tipo injeção SQL. Informações como o comando SQL, nome da classe, nome do método e linha da classe serão mapeadas em um arquivo XML (*eXtensible Markup Language*) que será utilizado na próxima etapa.

   A Figura 15 apresenta um exemplo do arquivo XML gerado a partir da análise da classe da Figura 11.

```
<Vulnerability>
    <Location>DoSubjectSearch_Vx0.java:26</Location>
    <Method>doSubjectSearch</Method>
    <Code>SELECT * FROM tpcw_item, tpcw_author WHERE tpcw_item.i_a_id =
    tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM &lt;= 50 ORDER BY
    tpcw_item.i_title </Code>
</Vulnerability>
```

Figura 15 Arquivo XML referente a comandos SQL encontrados na aplicação.

2. **Análise**: esta fase irá trabalhar em conjunto com a ferramenta de detecção de vulnerabilidades. Foi utilizada a técnica citada em Antunes e Vieira (2011), no qual o *Driver Java Database Connectivity* (JDBC) foi instrumentado usando Programação Orientada a Aspecto (AOP - *Aspect-Oriented Programming*). AOP foi utilizada de forma a não impactar o comportamento normal da aplicação. Foi gerada uma nova biblioteca do *driver* JDBC, modificada com AOP, que deve ser utilizada em substituição ao *driver* original. Para isso, o *classpath* deve ser modificado para apontar para a versão AOP do *driver*. Após a instrumentação as consultas ao banco de dados foram monitoradas durante a execução da ferramenta de detecção de vulnerabilidades. Os comandos SQL executados durante a execução dos casos de teste são então interceptados e um arquivo XML é gerado com os comandos SQL que foram efetivamente executados. A Figura 16 exibe um exemplo de um arquivo XML gerado durante os testes. Este arquivo contém informações

sobre o comando SQL, nome da classe, linha da classe e a quantidade de vezes que o comando foi executado.

```
<Vulnerability>
    <Location>DoSubjectSearch_Vx0.java:26</Location>
    <Code>SELECT * FROM tpcw_item, tpcw_author WHERE tpcw_item.i_a_id =
    tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM &lt;= FOUND_A_NUMBER ORDER
    BY tpcw_item.i_title </Code>
    <Qty>5639</Qty>
</Vulnerability>
```

Figura 16 Comandos SQL interceptados durante a execução dos testes.

Com os arquivos XML gerados na pré-análise e na análise, é possível realizar a comparação dos *hotspots* da aplicação pelo *hotspots* executados durante os testes e assim obter a métrica de cobertura *hotspot*. Esse processo é chamado de "Combinar *Hotspot*" (Figura 17), e é realizado de forma manual. Para automatizar esse processo seria necessário utilizar processamento de texto, o que está fora do escopo dessa pesquisa e poderá ser evoluída futuramente, em algum trabalho que envolva esse contexto.

Para calcular a métrica de cobertura *hotspot* vamos relacionar o número de *hotspots* executados pelo menos *n* vezes e o número total de *hotspots* encontrados. Chamaremos essa métrica de *n+hits*, sendo que neste trabalho os resultados serão de apenas quatro diferentes valores de *n*:

- 1 - O *hotspot* foi testado, mesmo que minimamente;
- 5 - O *hotspot* foi executado em número maior do que a execução única;
- 10 - O *hotspot* é executado um número considerável de vezes no âmbito de ataques contra ele;
- 50 - O *hotspot* é exercido por um grande número de testes, fornecendo boas garantias de que as vulnerabilidades existentes serão encontradas.

A partir das alterações descritas nas subseções 4.2 e 4.3 tem-se uma nova visão da nossa abordagem experimental, que é representada na Figura 17.

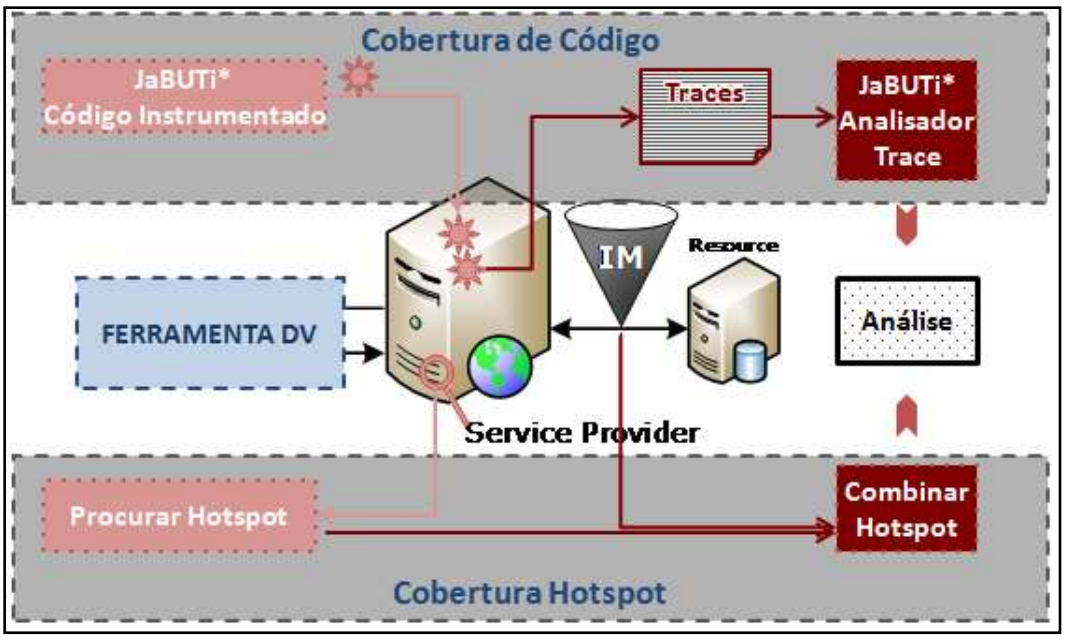

Figura 17 Abordagem experimental com as adaptações

Analisando a Figura 17 observa-se que a "Cobertura de Código" no bloco cinza destaca os itens citados na seção 4.2 que, resumidamente, são: **JaBUTi* Código Instrumentado** – responsável por inserir as "sondas" no código fonte dos serviços; ***Trace*** **–** arquivo que contém informações sobre o código executado durante os testes e **JaBUTi* Analisador *Trace*** – analisa o arquivo trace e calcula as métricas de cobertura de código.

Já o bloco cinza "Cobertura *Hotspot*" da Figura 17 ilustra os itens da seção 4.3 que, resumidamente, são: **Procurar *Hotspot*** – fase da pré-analise que busca comandos no código compilado do serviço que representem instruções SQL e **Combinar Hotspot** – durante a execução da Ferramenta DV (Detecção de Vulnerabilidades) os comandos SQL são interceptados pelo módulo **IM** e comparados com os comandos encontrados na pré-análise afim de verificar quais comandos foram executados e quais não foram, obtendo assim a cobertura *hotspot*. Com as informações das métricas de cobertura de código e cobertura *hotspot* coletadas é realizada a análise dos resultados.

No próximo capítulo serão apresentados dois estudos de caso com duas diferentes ferramentas para o mesmo conjunto de *Web Services*. As ferramentas são representadas na Figura 17 como ferramenta DV. Dessa forma, será exercitada a abordagem proposta, assim como será feita a análise das métricas descritas neste capítulo.

5. ESTUDOS DE CASO

Definida a abordagem experimental e realizada as alterações para que se torne possível obter a cobertura de código e a cobertura *hotspot*, este capítulo irá apresentar os experimentos realizados para a necessária validação da abordagem.

Para isso é necessário definir um conjunto de serviços que contenham vulnerabilidades. É preciso também definir ferramentas de detecção de vulnerabilidades que serão responsáveis por executar os testes automáticos. Assim, a abordagem proposta será executada com o objetivo de analisar as métricas de cobertura de código e cobertura *hotspot* alcançadas.

Para os experimentos relacionados neste capítulo foi selecionado um ambiente baseado em *Web Services* que foi desenvolvido na linguagem Java e que contêm vulnerabilidades do tipo Injeção de SQL (*SQL Injection*). Este mesmo conjunto de serviços será testado em dois experimentos diferentes. As ferramentas selecionadas para realizar a validação dos dois experimentos foram: Sign-WS e RAD-WS. A configuração *(C)* é totalmente independente de uma ferramenta, assim, os experimentos podem facilmente ser reproduzidos para outras ferramentas de detecção de vulnerabilidades, sem a necessidade de outras modificações que não seja a substituição da ferramenta.

As subseções a seguir fornecem detalhes sobre as partes variáveis da abordagem: *Web Services* e as Ferramentas de Detecção de Vulnerabilidades utilizadas nos experimentos.

5.1.SERVIÇOS

Os *Web Services* usados como *workload* para os testes de detecção de vulnerabilidades foram adotados a partir do *benchmark VDBenchWS-pd* da ferramenta de detecção de vulnerabilidades de injeção SQL proposto em (ANTUNES e VIEIRA, 2015).

O conjunto de serviços está disponível online e é composto por 21 serviços, adaptados a partir de três *benchmarks* desenvolvidos pela *Transactions processing Performance Council*, são eles: TPC-App, TPC-C e TPC-W (ver detalhes sobre a especificação destes valores de referência em (TPC, 2009)) incluindo ambas as versões vulneráveis (em que foram implantadas vulnerabilidades propositalmente) e não vulneráveis (em que não se tem vulnerabilidades

conhecidas) dos serviços. Estes serviços incluem um total de 80 operações (#Op.) com 158 vulnerabilidades de injeção SQL conhecidas (#Vuln.).

A Tabela 5 apresenta para cada serviço, o número de vulnerabilidades conhecidas, o número de linhas de código por operação (LoC/Op), e a média da Complexidade Ciclomática (Méd. C.) do código (calculada usando a ferramenta *SourceMonitor* (CAMPWOOD, 2015)) (LYU, 1996). Como podemos ver, os serviços são bastante diversificados em termos de complexidade, tamanho, e número de vulnerabilidades. Esta diversidade é um fator determinante, uma vez que permite fazer uma análise mais detalhada. Detalhes estão disponíveis em Antunes e Vieira (2015).

Tabela 5 Detalhes dos *Web Services* testados

BM	Nome do Serviço	# Op.	# Vuln.	LoC/Op	Méd. C.
TPC-App	ProductDetail	2	0	242	5
	NewProducts	2	1	206	4.5
	NewCustomer	6	35	1230	5.6
	ChangePaymentMethod	2	2	198	5
TPC-C	Delivery	9	10	2043	21
	NewOrder	7	15	2317	33
	OrderStatus	7	18	1463	13
	Payment	13	34	4251	25
	StockLevel	4	6	320	4
TPC-W	AdminUpdate	2	2	162	5
	CreateNewCustomer	6	27	978	3
	CreateShoppingCart	2	0	414	2.67
	DoAuthorSearch	2	1	88	3
	DoSubjectSearch	2	1	90	3
	DoTitleSearch	2	1	90	3
	GetBestSellers	2	1	124	3
	GetCustomer	2	1	92	4
	GetMostRecentOrder	2	1	258	6
	GetNewProducts	2	1	100	3
	GetPassword	2	1	80	2
	GetUsername	2	0	80	2
Total		**80**	**158**	**14826**	-

5.2. FERRAMENTAS DE DETECÇÃO DE VULNERABILIDADES

Para a validação da abordagem quanto a flexibilidade da nossa abordagem, são utilizadas duas diferentes ferramentas para realizar os experimentos, são elas RAD-WS e Sign-WS. Essas ferramentas e as suas configurações são detalhadas nas próximas seções.

5.2.1. SIGN-WS

A Figura 18 representa uma ilustração simplificada de um experimento utilizando a abordagem proposta.

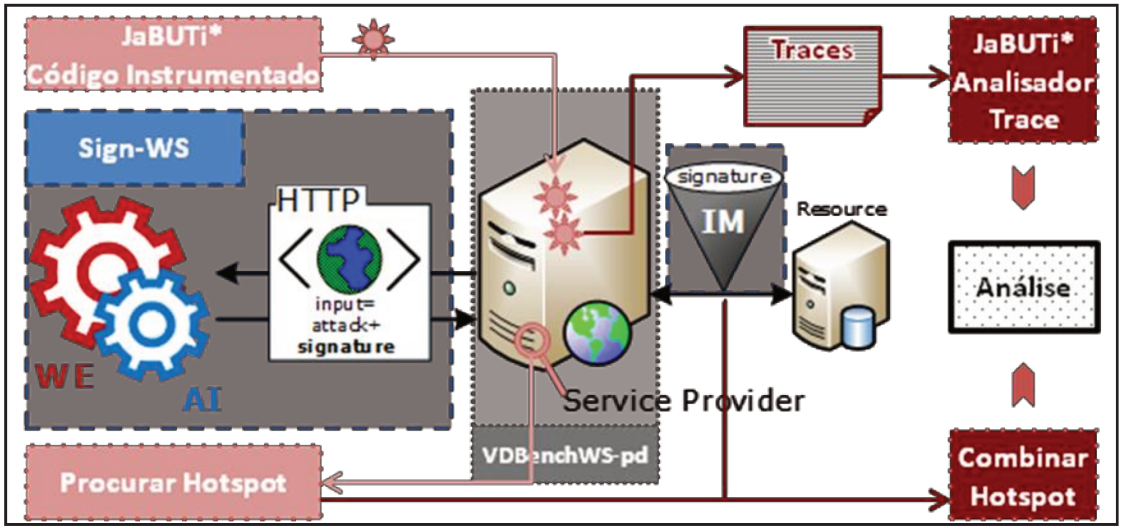

Figura 18 Representação da abordagem experimental usando a ferramenta Sign-WS

A ferramenta Sign-WS, está destacada em linhas tracejadas na Figura 18. Escolhemos utilizar esta ferramenta, pois: 1) é open source, fácil de configurar e modificar; 2) é eficaz, detectando mais vulnerabilidades e evita relatar falsos positivos; 3) de acordo com os autores as vulnerabilidades que a ferramenta não é capaz de relatar são "localizadas em lugares no código de difícil acesso". Os detalhes dessa ferramenta podem ser encontrados no Capítulo 3 ou em (ANTUNES e VIEIRA, 2011).

Sign-WS é baseada em testes de penetração reforçada com assinaturas e usa o monitoramento de interface (IM) para detectar a presença dessas assinaturas, permitindo assim a detecção de vulnerabilidades de injeção. A abordagem implementada pela ferramenta supera as limitações de testes de penetração, melhorando o processo de teste e sem a necessidade de acessar ou modificar o código do serviço a ser testado. Essa abordagem permite alcançar maior efetividade, uma vez que fornece as informações necessárias para aumentar o número de vulnerabilidades detectadas e evitar relatórios de falsos positivos.

O módulo Emulador de Workflow (WE) analisa a descrição dos *Web Sevices* e gera um conjunto de requisições válidas (com tamanho *w*). Estas requisições são depois modificadas pelo módulo Injetor de Ataques (IA) que gera um conjunto de ataques com tamanho *w* * *p* * *am*, onde *p* é o número de parâmetros e *am* é o número de diferentes modelos de ataque. Durante este processo, as interfaces dos *Web Services* são monitoradas (módulo IM na Figura 18) para detectar as assinaturas que representam vulnerabilidades.

Para realizar os experimentos foram definidas quatro configurações diferentes em relação a qualidade e quantidade de testes realizados. Para isso foi necessário configurar o módulo WE, sendo possível definir a quantidade de solicitações válidas e também selecionar o tipo de solicitação: pedidos com entradas aleatórias ou pedidos com domínios pré-definidos dos *Web Services*. Com base nesses dois parâmetros, foram definidas as seguintes configurações:

- *Domain Large* (DL) - a ferramenta é configurada para gerar um conjunto **grande** (*w* = 10) de solicitações válidas com base em **domínios pré-definidos**.
- *Random Large* (RL) - a ferramenta é configurada para gerar **aleatoriamente** um conjunto **grande** (*w* = 10) de solicitações válidas.
- *Domain Small* (DS) - a ferramenta é configurada para gerar um conjunto **pequeno** (*w* = 1) de solicitações válidas usando **domínios pré-definidos**.
- *Random Small* (RS) - a ferramenta é configurada para gerar **aleatoriamente** um conjunto **pequeno** (*w* = 1) de solicitações válidas.

5.2.2. RAD-WS

Outra ferramenta utilizada nos experimentos foi a RAD-WS. Os detalhes dessa ferramenta podem ser encontrados no Capítulo 3 ou (ANTUNES *et al.,* 2009). A Figura 19 ilustra como ficou a abordagem adaptada a essa ferramenta.

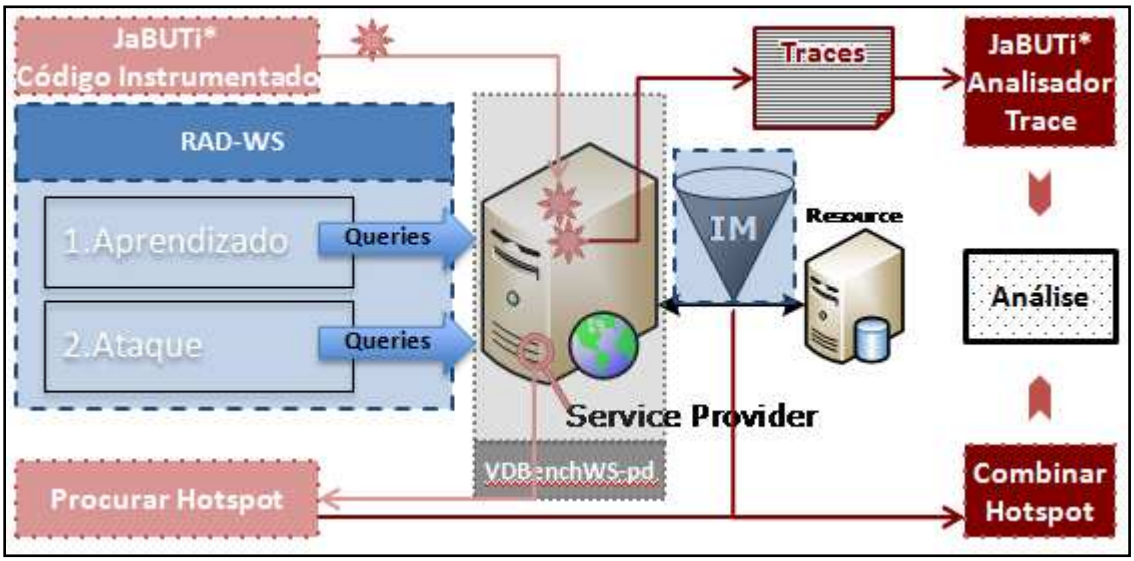

Figura 19 Representação da abordagem experimental usando a ferramenta RAD-WS

A ferramenta RAD-WS está destacada em linhas tracejadas na Figura 19 e a sua execução é separada em duas fases principais: Aprendizado e Ataque. Na fase de Aprendizado é gerado um *workload* baseado nas operações, nos parâmetros, nos tipos de dados de entrada e nos domínios de entrada a partir do arquivo WSDL do serviço. Como é impraticável executar o *workload* para todas as combinações geradas, fica a cargo do usuário definir o tamanho do subconjunto de testes a serem executados durante essa etapa. Essa execução da fase de aprendizagem tem a finalidade de conhecer os comandos SQL executados.

A fase de Ataque gera um conjunto de solicitações com valores mal-intencionados e com parâmetros de entrada que executam injeções SQL. Nesta fase o usuário também pode definir o número de chamadas de teste que serão executadas pelo *attackload*.

Para a ferramenta RAD-WS foram definidas oito diferentes Configurações *(C)* de execução os testes. As configurações na fase de ataque possuem a mesma quantidade que a da ferramenta Sign-WS (DL e RL = 10 solicitações; DS e RS =1), mas como a ferramenta RAD-WS possui a fase de aprendizado ela foi configurada da seguinte forma:

- *D-p250-a10* - a ferramenta é configurada para gerar um conjunto de 250 requisições na fase de aprendizado e na fase de ataque usa 10 requisições como base. São usados **domínios pré-definidos** nos parâmetros das solicitações válidas.

- *R-p250-a10* – a ferramenta é configurada para gerar um conjunto de 250 requisições na fase de aprendizado e na fase de ataque usa 10 requisições como base, porém é usado um conjunto **aleatório** de solicitações válidas.

- *D-p25-a10* - a ferramenta é configurada para gerar um conjunto de 25 requisições na fase de aprendizado e na fase de ataque usa 10 requisições como base. São usados **domínios pré-definidos** nos parâmetros das solicitações válidas.

- *R-p25-a10* - a ferramenta é configurada para gerar um conjunto de 25 requisições na fase de aprendizado e na fase de ataque usa 10 requisições como base, porém é usado um conjunto **aleatório** de solicitações válidas.

- *D-p250-a1* - a ferramenta é configurada para gerar um conjunto de 250 requisições na fase de aprendizado e na fase de ataque usa 1 requisição como base. São usados **domínios pré-definidos** nos parâmetros das solicitações válidas.

- *R-p250-a1* - a ferramenta é configurada para gerar um conjunto de 250 requisições na fase de aprendizado e na fase de ataque usa 1 requisição como base, porém é usado um conjunto **aleatório** de solicitações válidas.

- *D-p25-a1* - a ferramenta é configurada para gerar um conjunto de 25 requisições na fase de aprendizado e na fase de ataque usa 1 requisição como base. São usados **domínios pré-definidos** nos parâmetros das solicitações válidas.

- *R-p25-a1* - a ferramenta é configurada para gerar um conjunto de 25 requisições na fase de aprendizado e na fase de ataque usa 1 requisição como base, porém é usado um conjunto **aleatório** de solicitações válidas.

6. RESULTADOS E DISCUSSÕES

Neste capítulo serão analisados os resultados da execução dos experimentos. Os resultados serão apresentados em duas subseções distintas, uma dedicada à ferramenta Sign-WS e outra a RAD-WS.

Para esta análise será utilizado como referência as vulnerabilidades detectadas, ou seja, a taxa de verdadeiros positivos (*True Positives Rate* - TPR), métrica importante para a qualidade dos testes.

6.1.SIGN-WS

Os resultados gerais para as quatro configurações da ferramenta Sign-WS são exibidos na Figura 20. A legenda mostra as linhas dos oito critérios de cobertura de código e as quatro métricas de cobertura *hotspot* que foram consideradas e discutidas nas subseções 2.3.1 e 4.3 respectivamente, como também uma linha para a métrica TPR.

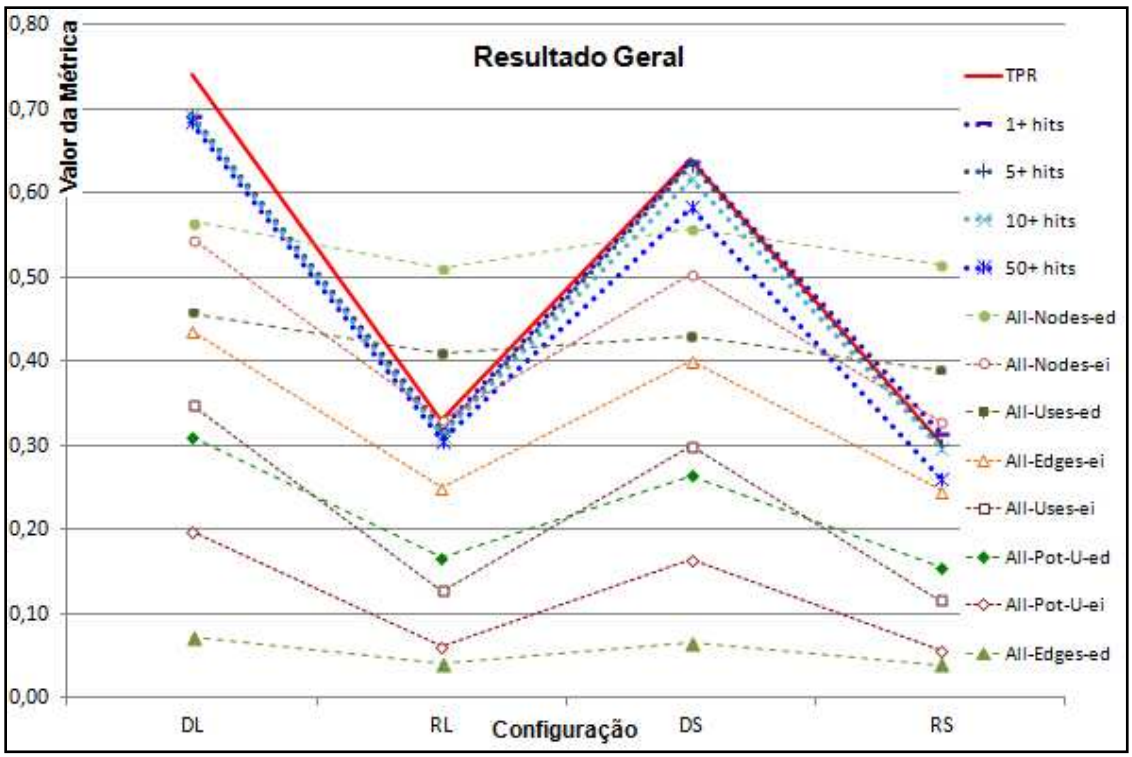

Figura 20 Resultado geral para as quatro configurações - Sign-Ws

Analisando os resultados é possível observar, primeiramente, que as diferentes configurações da ferramenta de teste Sign-WS resultam em diferentes valores de cobertura e de números de vulnerabilidades detectadas. É possível observar também que as configurações DL e DS obtiveram um percentual de cobertura melhor que as demais configurações, porém, como era previsto, a melhor configuração foi a DL que usa solicitações de entrada de domínio pré-definidas e um conjunto maior de solicitações. Essa configuração alcançou o maior valor da métrica na curva TPR.

De forma geral, as curvas aparentam bastante semelhança, o que sugere uma certa correlação ou mesmo alguma proporcionalidade entre as métricas de cobertura e a qualidade dos testes de detecção de vulnerabilidade. Foi observado que existem algumas métricas que são claramente mais correlacionadas à TPR. Por exemplo, os critérios *All-Nodes-ed* e *All-uses-ed* parecem de menor utilidade do que o critério *All-Edges-ei* ou *10+ hits*. A métrica *5+ hits* parece bastante correlacionada à TPR.

Com esta análise foi possível observar que a cobertura de código e a métrica de cobertura *hotspot* podem ser eficazes na comparação de diferentes conjuntos de testes de detecção de vulnerabilidades. Assim é respondida a questão **Q3** do Capítulo 1.

Para uma seleção mais criteriosa das métricas de cobertura de código e *hotspot* foi feito um estudo mais detalhado dos resultados obtidos, que são apresentados a seguir.

- **Resultado Cobertura de Código**

Para ter uma visualização mais correta sobre os dados, decidimos usar a TPR como *baseline* e dividir os valores de todas as variáveis pelo correspondente valor de TPR. Como mostra a Figura 21 que, devido a esta operação, apresenta a TPR como uma linha horizontal com o valor 1. Com esta análise será possível compreender melhor quais métricas estão mais proporcionais à TPR. Quanto mais horizontal é a linha que retrata uma métrica, mais correlacionada é esta métrica à TPR. Olhando a Figura 21, podemos ver que, se forem selecionadas as métricas corretamente, pode-se dar uma resposta afirmativa à questão **Q1** (ver Capítulo 1).

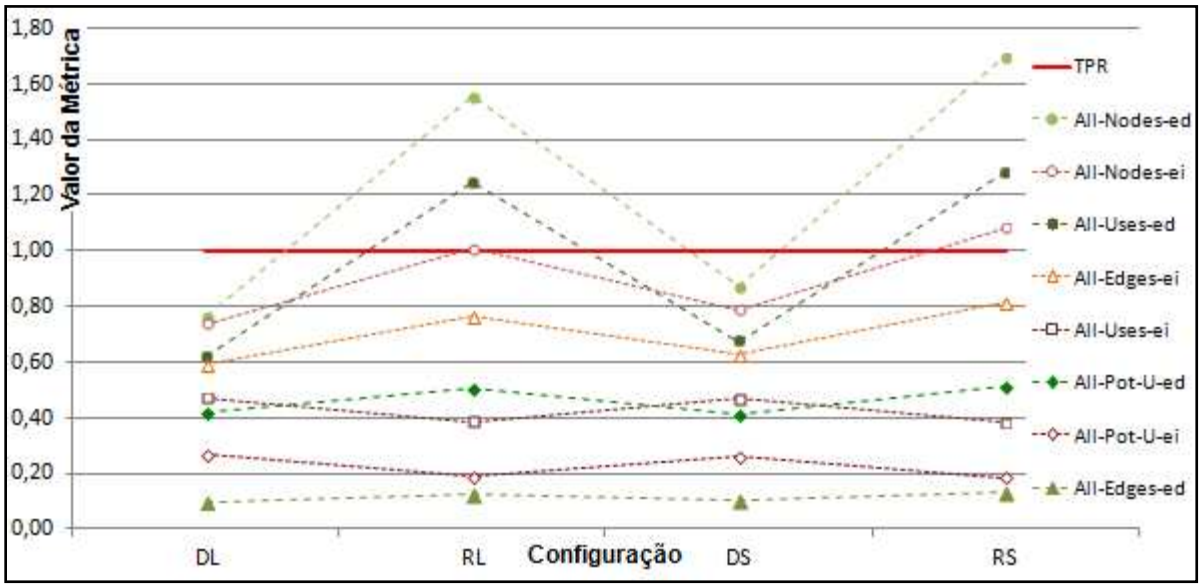

Figura 21 Resultado da métrica de cobertura de código dividido pela respectiva TPR – Sign-WS

As métricas *All-Nodes-ed* e *All-uses-ed* são as que têm menor relação com a TPR. Isto acontece porque estas métricas se referem a nós e a associações *def-use* que estão no contexto de tratamento de exceções, isto é, códigos que contêm vulnerabilidades geralmente não estão localizados nesses blocos dependentes de exceções. Embora apresentem valores muito baixos, *All-Edges-ed* e *All-Pot-uses-ed* apresentam uma boa correlação à TPR. As métricas –ed apresentam um índice elevado nas configurações aleatórias (RL e RS) comparados com as configurações de domínios pré definidos (DL e DS), o que leva a concluir que nestas configurações, os nós dependentes de exceção apresentaram maior cobertura.

Para esclarecer quais métricas são mais proporcionais à TPR foi realizada a análise de correlação simples citada no Capítulo 2. Para cada métrica foi calculado o coeficiente de correlação de Pearson (*r*) entre a TPR e também o declive (*m*) da linha ajustada da regressão linear (calculada usando o método dos mínimos quadrados). O declive (*m)* retrata quantas unidades da TPR aumenta para cada unidade que aumenta da métrica considerada.

A Figura 22 apresenta o gráfico de dispersão da métrica de cobertura de código. No gráfico é apresentada a legenda para todas as métricas de cobertura de código. O gráfico de dispersão é usado principalmente para visualizar a relação/associação entre os critérios de métrica de cobertura de código e a TPR.

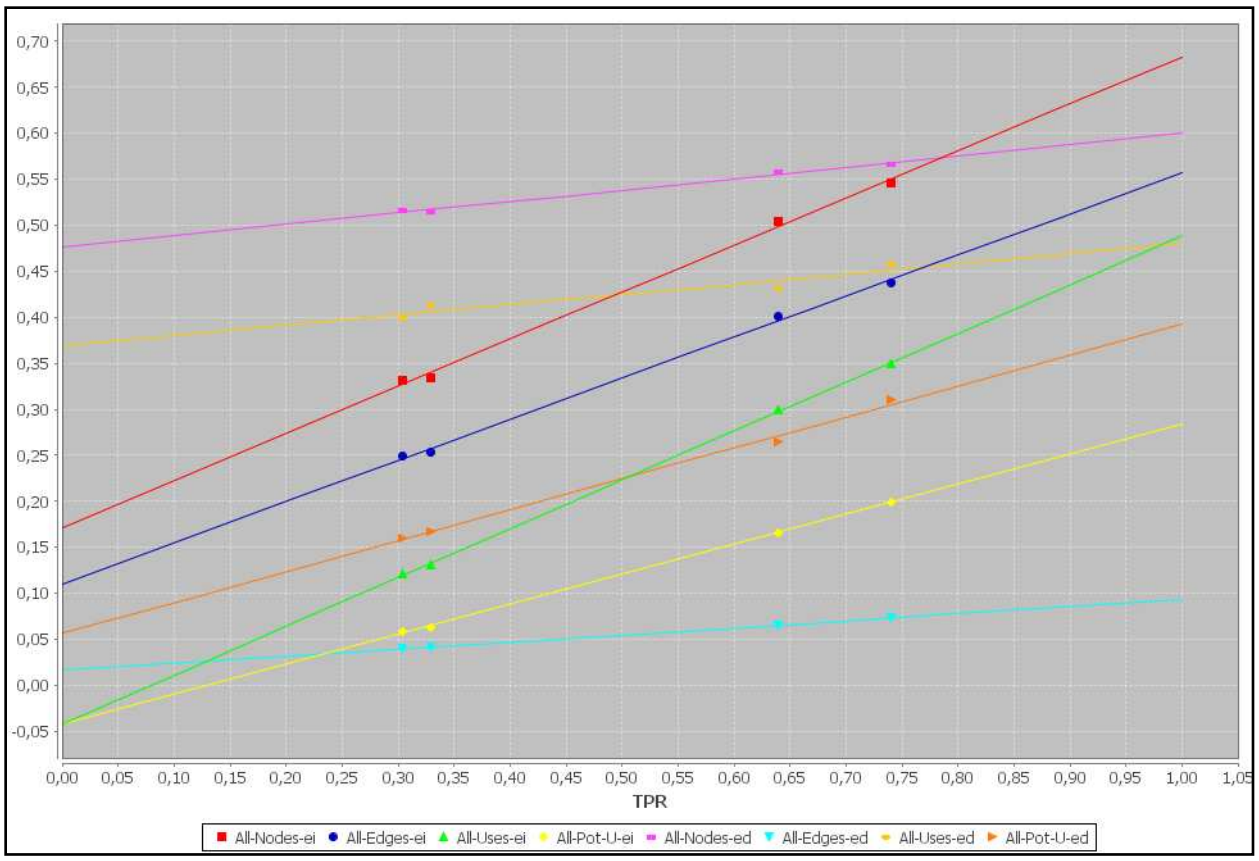

Figura 22 Gráfico de dispersão das métricas de cobertura de código – Sign-WS

À primeira vista observa-se que, na Figura 22, todos os pontos de todas as métricas estão próximos às linhas correspondentes. Por exemplo, observando a métrica *All-Nodes-ei* em vermelho, nota-se que os pontos em vermelho estão próximos da linha traçada em vermelho, o que indica uma alta correlação da métrica de cobertura de código e a TPR. Isso ocorre também com todas as métricas. Outro fator a analisar é o declive das retas, que mostra que as métricas *All-Nodes-ed*, *All-Edges-ed* e *All-Uses-ed* possuem um declive menor quando comparadas às outras métricas, indicando uma correlação menor.

Para uma análise mais minuciosa, os resultados foram detalhados na Tabela 6, que apresenta os resultados da correlação de Pearson (r) e também o declive (m) da linha ajustada. Os resultados da Tabela 6 juntamente com a Figura 22 mostram que todas as métricas apresentam um coeficiente de correlação muito alto em relação à TPR e apenas o critério *All-Uses-ed* fica abaixo de 0,99. Isto confirma que todos os critérios de fato têm uma forte correlação

com o número de vulnerabilidades detectadas. Os resultados também confirmam nossas observações a respeito dos critérios dependentes de exceção: *All-Nodes-ed* e *All-uses-ed,* que apresentam menor correlação à TPR.

Tabela 6 Correlação entra a Cobertura de Código e a TPR – Sign-WS

	All-Nodes		***All-Edges***		***All-Uses***		***All-Pot-Uses***	
	-ei	***-ed***	***-ei***	***-ed***	***-ei***	***-ed***	***-ei***	***-ed***
r	0,99881	0,99509	0,99889	**0,99980**	**0,99980**	*0,94807*	0,99973	0,99791
m	0,51142	0,**12462**	0,44695	*0,**07683***	**0,53020**	0,**11084**	0,32594	0,33558

Já os critérios *All-Pot-Uses-ed* e *All-Edges-ed* apresentam um coeficiente maior, embora *All-Edges-ed* apresente a terceira maior correlação, também apresenta o menor *m*. Isto mostra que, como visto na Figura 21, os valores de *All-Edges-ed* e TPR crescem de forma diferente em termos absolutos.

Analisando os critérios independentes de exceção, observa-se que *All-Nodes-ei* apresentam a pior correlação, enquanto *All-Uses-ei* e *All-Pot-Uses-ei* apresentam a maior correlação com a qualidade dos testes de detecção de vulnerabilidades. Isto é esperado uma vez que os critérios *All-Nodes* são considerados mais fracos em relação à estimativa da qualidade de testes e *All-Uses* e *All-Pot-Uses* são considerados mais fortes. Embora as correlações de ambos os *All-Uses-ei* e *All-Pot-Uses-ei* à TPR são os mais elevados, o primeiro apresenta um maior valor de *m*. Desta forma, os resultados não fornecem evidências para apoiar a alegação de que *All-Pot-Uses-ei* é mais forte do que *All-Uses-ei*.

- **Resultado da cobertura *hotspot***

Os resultados da métrica de cobertura *hotspot* foram detalhados e mais uma vez foram divididos pela TPR (*baseline*), gerando a Figura 23. Note que a escala do eixo vertical foi "ampliada", quando comparado com a Figura 22.

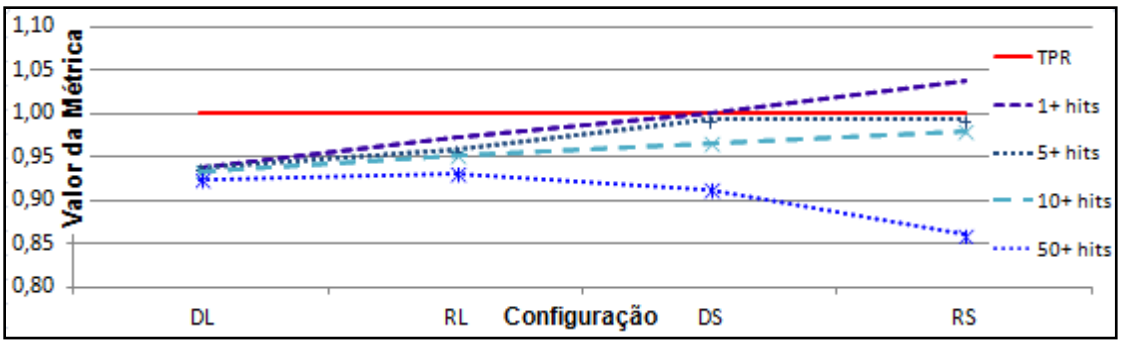

Figura 23 Resultado da métrica da cobertura *hotspot* dividido pela respectiva TPR – Sign-WS

Analisando a figura é possível observar uma maior proximidade entre a TPR e a cobertura *hotspot* quando comparado com a proximidade entre a TPR e a cobertura de código (Figura 22). De maneira geral, todas as métricas de cobertura *hotspot* parecem proporcionais à TPR, respondendo afirmativamente à pergunta **Q2** (ver Capítulo 1).

Do ponto de vista da detecção de vulnerabilidades, os resultados mostram que a ferramenta utilizada (Sign-WS) é capaz de exercitar cada *hotspot* entre 5 e 10 vezes para detectar as vulnerabilidades relacionadas. Ao mesmo tempo, os resultados para *1+ hits* mostram que, em alguns casos, uma única chamada ao comando *hotspot* pode não ser suficiente. Por outro lado, o resultado para *50+ hits* também proporciona uma correlação interessante, mas apenas para as configurações L (com grandes conjuntos de solicitações: DL e RL). Isso é compreensível e esperado, pois um conjunto menor de solicitações da ferramenta de detecção de vulnerabilidades realiza testes menores e, portanto, produz um número menor de métricas *n+ hits*.

Para detalhar a análise, repetiu-se para cada métrica de cobertura *hotspot* a correlação de Pearson (r) e o declive (m) da linha ajustada da regressão linear como apresenta a Figura 24 e detalhadamente a Tabela 7.

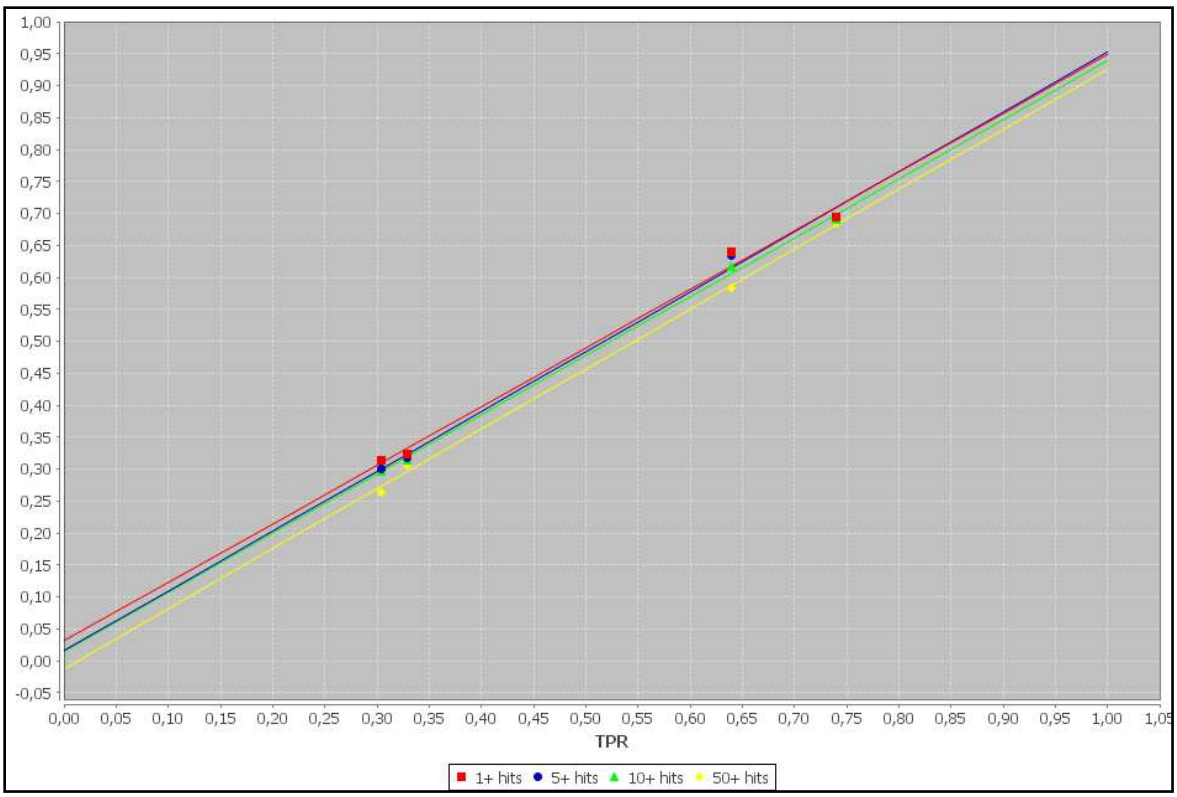

Figura 24 Gráfico de dispersão das métricas de cobertura *hotspot* – Sign-WS

A Figura 24 mostra que existe uma alta correlação das métricas de cobertura *hotspot* à TPR, pois os pontos estão muito próximos das retas correspondentes e apresentam um declive muito perto de 1. Já na Tabela 7 é possível observar que, embora os valores de *r* sejam menores do que a melhor métrica de cobertura de código, eles são acompanhados com valores de *m*, muito próximos de 1, confirmando que as métricas de cobertura *hotspot* são mais próximas da TPR do que qualquer métrica de cobertura de código vista anteriormente. Observa-se também que, embora *1+ hits* e *5+ hits* parecem mais próximas da TPR, a métrica *50+ hits* é que apresenta os melhores valores para *r* e *m*, mostrando que é a mais correlacionada à TPR.

Tabela 7 Correlação entra a Cobertura *Hotspot* e a TPR – Sign-WS

	1+ hits	***5+ hits***	***10+ hits***	***50+ hits***
r	*0,99634*	0,99724	0,99914	**0,99919**
m	*0,91643*	0,93621	0,92527	**0,93695**

De acordo com os resultados observados, as métricas de cobertura *hotspot* podem vir a constituir de fato uma forma de estimar a eficácia de uma ferramenta de detecção de

vulnerabilidades se usada de forma padronizada, respondendo assim à questão **Q4** (ver Capítulo 1).

Tais observações nos levam a argumentar que é importante definir métricas de cobertura que são específicas para o domínio de testes de detecção de vulnerabilidade. Na verdade, os resultados alcançados por essas métricas *ad-hoc* (*n* + *hits*) são bastante promissores e mostram que pode ser possível propor métricas melhores que levem em conta o número de testes realizados, o número de entradas testadas, entre outros.

- **Serviços agrupados por lógica de negócio**

Foi utilizada outra forma de avaliar os resultados relatados anteriormente, no qual agrupo-se os *Web Services* por características relevantes. Essa análise também serve para discutir a validade dos resultados. Uma das formas de agrupar os serviços é pela lógica de negócio, ou seja, o contexto de negócio que está por trás da implementação do serviço. Como visto na Seção 5.1 os *Web Services* usados como *workload* são derivados de um *benchmark* (BM) previamente adaptado a partir de TPC (2009) (TPC-C, TPC-W, e TPC-app), onde cada um possui uma implementação de um negócio.

A Tabela 8 e a Figura 25 apresentam um resumo dos resultados agrupados pelos serviços. Os números correspondem ao total de TPR, as métricas de cobertura *hotspot* e as quatro principais métricas de cobertura de código de acordo com as análises anteriores. Assim conseguimos analisar os resultados em diferentes configurações e para cada *benchmark*. Na Tabela 8 também são apresentados o (*r)* e (*m)* calculados na análise de correlação para cada agrupamento e na Figura 25 é apresentado o gráfico de dispersão para o agrupamento de serviços pela lógica de negócio de todas as configurações.

Tabela 8 Resultados divididos pela lógica de negócio - Sign-WS

BM	Cfg	*TPR*	*1+ hits*	*5+ hits*	*10+ hits*	*50+ hits*	*AU-ei*	*APU-ed*	*AE-ed*	*APU-ed*
TPC-App	**DL**	1,000	1,000	1,000	1,000	1,000	0,762	0,739	0,714	0,650
	RL	0,079	0,227	0,227	0,227	0,227	0,237	0,411	0,272	0,202
	DS	0,947	0,909	0,909	0,909	0,909	0,724	0,632	0,654	0,608
	RS	0,079	0,227	0,227	0,227	0,227	0,234	0,420	0,276	0,201
r		1,0	0,919	0,936	0,947	0,958	0,869	0,697	0,745	0,853
m		1,0	0,740	0,761	0,777	0,807	0,643	0,377	0,427	0,599
TPC-C	**DL**	0,506	0,592	0,592	0,592	0,586	0,242	0,280	0,320	0,153
	RL	0,325	0,225	0,225	0,225	0,224	0,071	0,135	0,181	0,039
	DS	0,337	0,532	0,532	0,510	0,465	0,195	0,239	0,285	0,122
	RS	0,277	0,212	0,212	0,210	0,173	0,066	0,133	0,176	0,036
r		1,0	0,782	0,782	0,811	0,865	0,834	0,837	0,823	0,833
m		1,0	1,565	1,565	1,589	1,700	0,739	0,624	0,602	0,494
TPC-W	**DL**	1,000	0,907	0,907	0,895	0,884	0,709	0,528	0,650	0,655
	RL	0,595	0,729	0,705	0,682	0,651	0,509	0,438	0,546	0,476
	DS	1,000	0,895	0,872	0,860	0,849	0,619	0,451	0,629	0,574
	RS	0,595	0,733	0,663	0,651	0,616	0,401	0,335	0,528	0,378
r		1,0	0,999	0,982	0,988	0,989	0,904	0,750	0,983	0,902
m		1,0	0,421	0,507	0,521	0,574	0,516	0,255	0,253	0,463

Podemos observar que: 1) as diferentes qualidades de testes levam a diferentes resultados da TPR e de cobertura; 2) há uma alta correlação entre as métricas e os resultados de detecção de vulnerabilidades; e 3) essa correlação é mais forte para a cobertura de *hotspot*. A exceção é o resultado da configuração DS em TPC-C, no qual a TPR obtida é muito inferior ao que algumas das métricas sugerem. Isto pode ser explicado pelo impacto do serviço *Order Status* (ver Tabela 5 de serviços), pois este requer uma grande diversidade de testes para exercitar todas as vulnerabilidades. Embora com informações de domínio pré-definidas, a configuração DS utiliza um conjunto pequeno de requisições de entrada, o que torna incapaz de fornecer essa diversidade: a verificação é possível através da queda das métricas *5+hits* e *50+ hits*, comportamento não observado nos outros cenários. Nestes pequenos grupos, um serviço faz uma grande diferença.

Outro ponto a destacar é que em TPC-App a discrepância entre os resultados das configurações aleatórias (RL, RS) e as configurações com domínio pré-definidos (DS, DL) é maior que em TPC-C e TPC-W. Isto ocorre porque os serviços em TPC-App usam dados mais

rigorosos nas requisições de domínios pré-definidos, exigindo testes que usam dados adequados, tornando os testes aleatórios quase sem valor.

Na Figura 25 é possível verificar que apesar de as métricas apresentarem uma boa correlação e declive, os pontos das métricas não estão tão próximos da linha correspondente como verificado anteriormente. Cada uma das amostras de TPR e das métricas agrupam dados acerca de menos serviços, o que leva a uma menor convergência dos valores e, portanto que a sua variância aumente. Ainda assim, é importante notar que a correlação se mantém elevada, o que reafirma as observações anteriores.

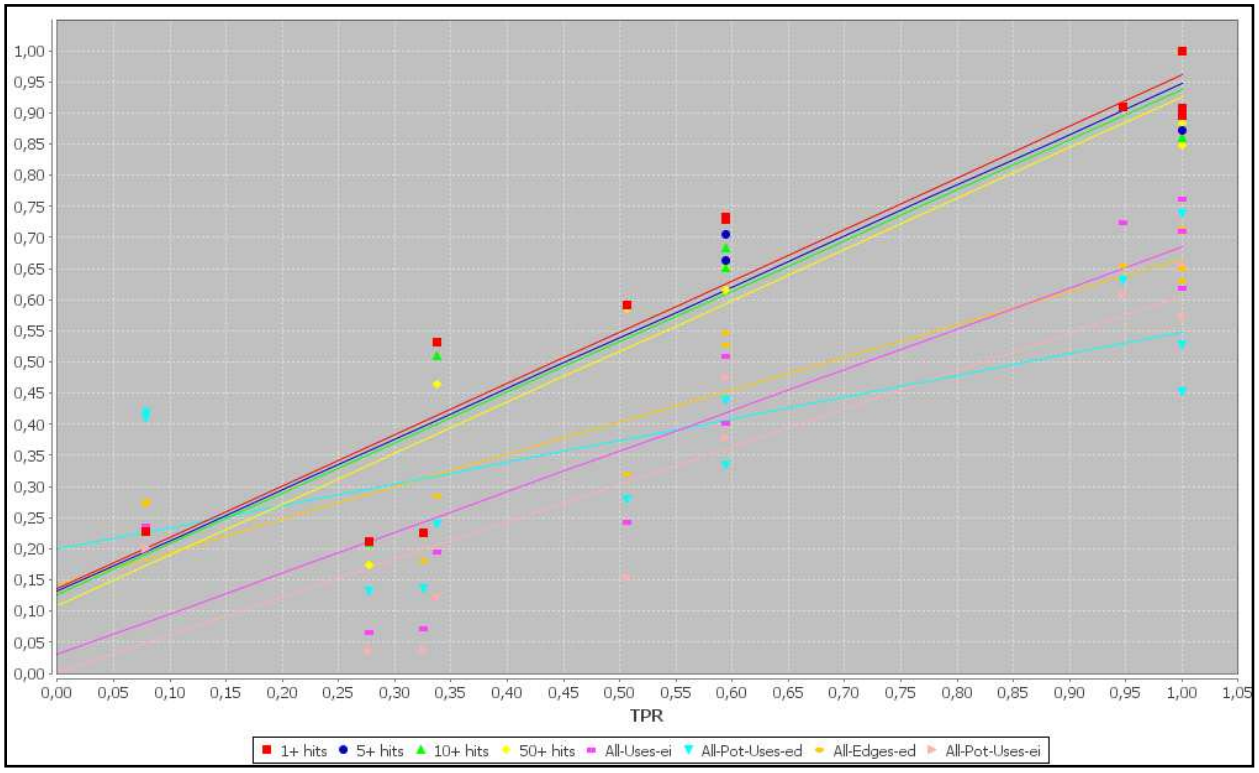

Figura 25 Gráfico de dispersão pela lógica de negócio - Sign-WS

- **Serviços agrupados pela complexidade ciclomática**

Para entender a relação entre a complexidade do código e os resultados observados, foi realizada outra análise. Desta vez, os serviços foram agrupados de acordo com a complexidade ciclomática (CC) média do código. Observando a Tabela 5, é possível ver que existem quatro serviços com CC muito mais elevados do que o restante (entre 13 e 33). Dos serviços restantes,

oito apresentam CC entre 4 e 6, enquanto os outros nove situam-se entre 2 e 3. Seguindo este agrupamento foram obtidos os resultados apresentados na Tabela 9.

Tabela 9 Resultados agrupados pela complexidade ciclomática – Sign-WS

CC	Cfg	*TPR*	*1+ hits*	*5+ hits*	*10+ hits*	*50+ hits*	*AU-ei*	*APU-ed*	*AE-ed*	*APU-ei*
13-33	DL	0,494	0,582	0,582	0,582	0,575	0,238	0,278	0,313	0,152
	RL	0,299	0,211	0,211	0,211	0,210	0,067	0,132	0,173	0,038
	DS	0,312	0,520	0,520	0,500	0,454	0,190	0,236	0,277	0,121
	RS	0,247	0,198	0,198	0,196	0,161	0,062	0,129	0,168	0,035
r		1,0	0,783	0,783	0,809	0,863	0,837	0,838	0,825	0,833
m		1,0	1,465	1,465	1,485	1,583	0,687	0,584	0,562	0,458
4-6	DL	0,958	0,909	0,909	0,909	0,909	0,680	0,484	0,664	0,595
	RL	0,229	0,364	0,364	0,364	0,364	0,245	0,357	0,308	0,206
	DS	0,917	0,848	0,848	0,833	0,818	0,649	0,450	0,617	0,559
	RS	0,229	0,364	0,364	0,364	0,333	0,243	0,360	0,311	0,205
r		1,0	0,999	0,999	0,998	0,996	1,000	0,983	0,998	1,000
m		1,0	0,729	0,729	0,718	0,730	0,594	0,154	0,469	0,525
2-3	DL	1,000	0,972	0,972	0,958	0,944	0,813	0,586	0,711	0,782
	RL	0,545	0,759	0,731	0,704	0,667	0,563	0,473	0,582	0,552
	DS	1,000	0,958	0,931	0,917	0,917	0,701	0,490	0,686	0,678
	RS	0,545	0,764	0,681	0,667	0,639	0,430	0,344	0,559	0,426
r		1,0	0,999	0,982	0,988	0,995	0,904	0,750	0,983	0,902
m		1,0	0,448	0,540	0,555	0,611	0,573	0,284	0,281	0,531

A primeira observação confirma o que foi descrito anteriormente, com uma exceção: as configurações do grupo de 13-33 apresentaram resultados muito semelhantes para o TPC-C da seção anterior. Isso acontece porque esse conjunto é composto dos quatro serviços mais complexos (máximo de 5) da implementação de TPC-C. Nos serviços com maior complexidade, todas as configurações apresentaram resultados piores, enquanto que nos serviços de menor complexidade todas as configurações apresentaram melhores resultados, mesmo com duas configurações com resultados perfeitos. Esta análise confirma que a cobertura de código é mais importante no código com maior complexidade. No entanto, isso não é uma regra absoluta, como podemos ver que o grupo 4-6, menos complexos do que 13-33, as configurações RL e RS apresentam os piores resultados. Isto é devido às requisições de domínio discutidas na seção anterior.

6.2. RAD-WS

Os resultados para as oito configurações da ferramenta Rad-WS são apresentados na Figura 26 e na Figura 27. As figuras ilustram os resultados gerais para a cobertura de código e cobertura *hotspot* respectivamente.

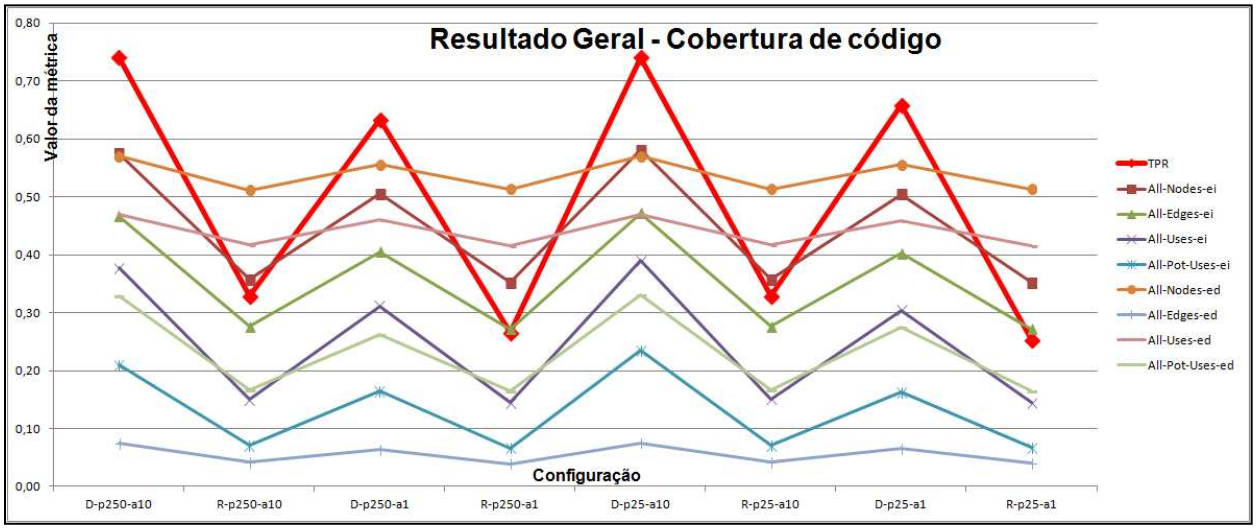

Figura 26 Resultado geral para a cobertura de código - Rad-WS

Analisando a Figura 26 observamos que as diferentes configurações da ferramenta de teste resultaram em diferentes coberturas e também em diferentes números de vulnerabilidades detectadas.

As configurações que apresentam os melhores resultados são as que usam valores de domínios pré-definidos para enviar as requisições (configurações **D**). Porém as configurações D-p250-a10 e D-p25-a10 são as que alcançam os melhores valores da TPR e também de todas as métricas de cobertura por ter um maior número de requisições na fase de ataque.

Além de verificar que todas as configurações **D** apresentam resultados melhores que as **R**, é possível notar que a configuração D-p25-a1 que possui um conjunto pequeno de solicitações é melhor do que a configuração R-p250-a10 que possui um conjunto de solicitações maior, mostrando que casos de testes mais direcionados aos domínios superam testes mais exaustivos no contexto de testes de segurança.

O formato geral das curvas parece similar, o que sugere uma relação entre a qualidade dos testes de detecção de vulnerabilidades e as métricas de cobertura. Os critérios *All-Nodes-ed* e *All-Uses-ed* parecem ser os menos relacionados à TPR do que o critério *All-Nodes-ei* (não seguem o mesmo padrão das linhas), o mesmo ocorreu com a ferramenta Sign-WS.

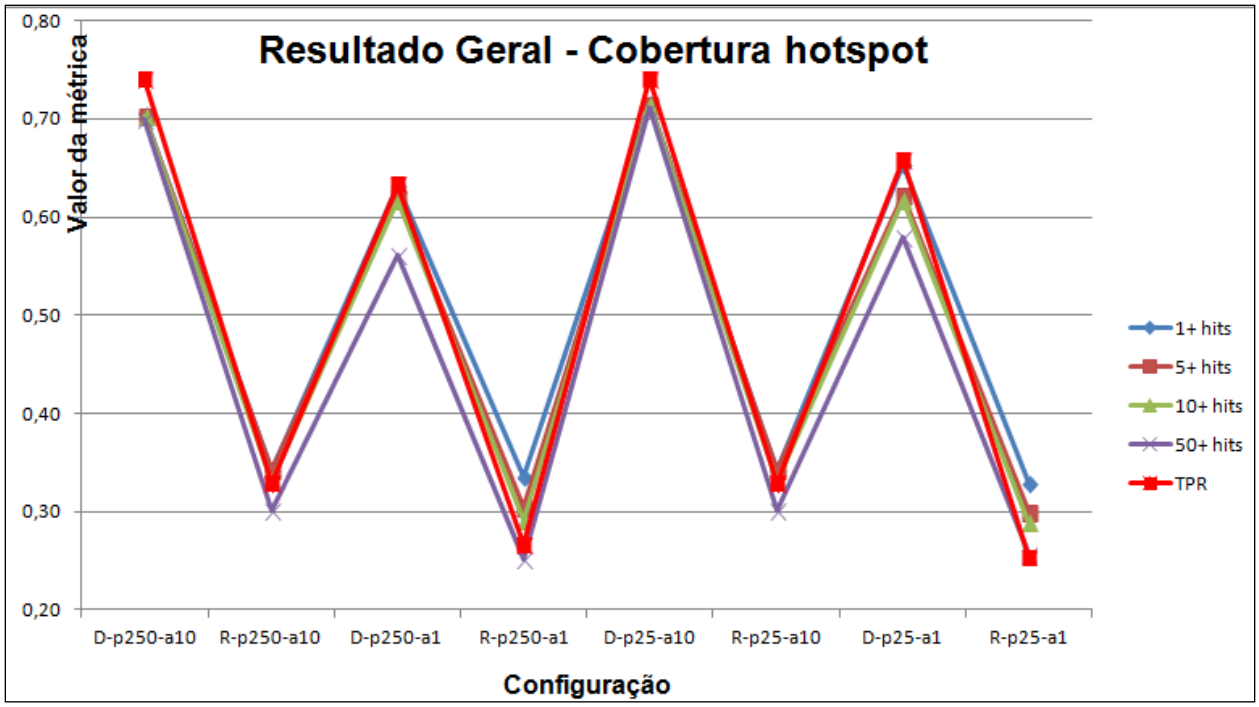

Figura 27 Resultado geral da cobertura *hotspot* - Rad-WS

A Figura 27 mostra que todas as linhas das métricas de cobertura *hotspot* possuem formatos similares à TPR, sugerindo uma alta relação entre os critérios de cobertura *hotspot* e a qualidade dos testes. Para a cobertura *hotspot* vale a mesma observação feita anteriormente, no qual todas as configurações **D** possuem coberturas mais altas que as **R**.

A análise geral dos resultados reforça que a cobertura de código e a cobertura *hotspot* podem ser eficazes na comparação dos conjuntos de testes de detecção de vulnerabilidades, respondendo novamente a questão **Q3** do Capítulo 1.

Assim como foi feito na análise da ferramenta Sign-WS, as subseções a seguir irão detalhar a análise dos resultados.

- **Resultado Cobertura de Código**

Os resultados da cobertura de código foram divididos pela TPR, cujo resultado pode ser visualizado na Figura 28. A linha horizontal em vermelho é a curva da TPR (*baseline*) que possui valor 1. Novamente, quanto mais horizontal a linha das métricas de cobertura de código, mais relacionada é a métrica à TPR.

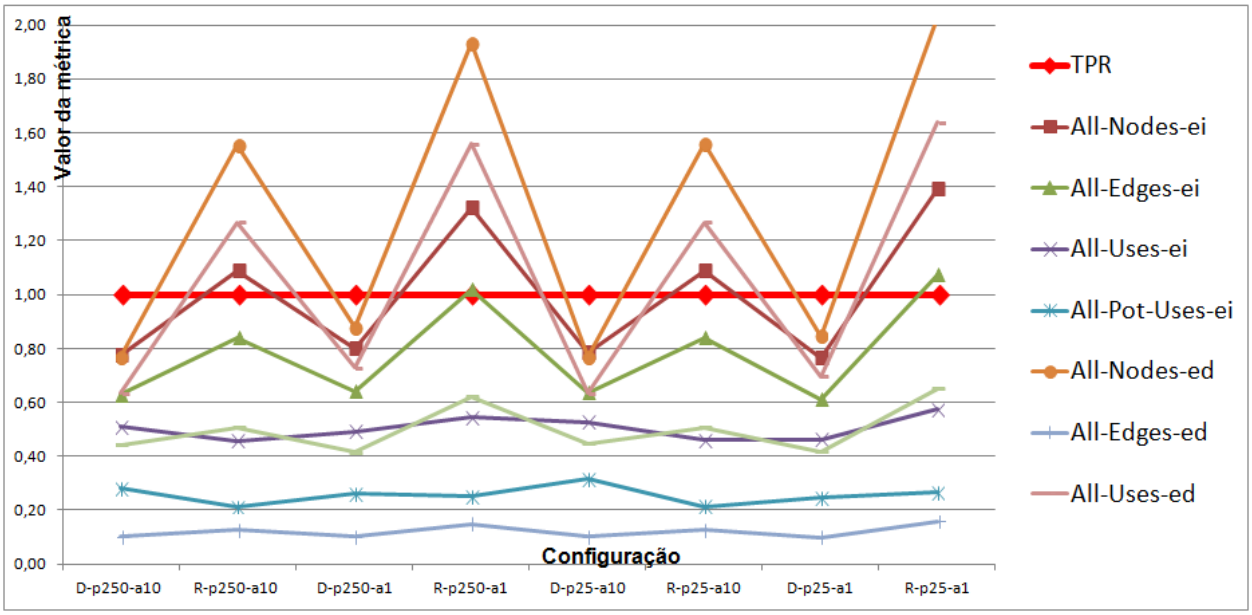

Figura 28 Resultado da métrica de cobertura de código dividida pela respectiva TPR – Rad-WS

Analisando a Figura 28 é possível verificar que as métricas *All-Nodes-ed* e *All-Uses-ed* parecem ser menos correlacionadas à TPR, pois são métricas que estão associadas ao fluxo dependente de exceção, onde normalmente não se encontram vulnerabilidades de segurança. As outras métricas *–ed (All-Edges-ed* e *All-Pot-Uses-ed)*, apresentem uma boa correlação, porém com valores de cobertura muito baixos. Novamente obtivemos um resultado das métricas *–ed* similar a ferramenta anterior.

Das métricas *–ei*, a que parece estar mais correlacionada à TPR é *All-Uses-ei*, que apresenta uma linha mais horizontal. Para um detalhamento mais preciso, foi calculado também para a ferramenta Rad-WS o coeficiente de correlação de Pearson (*r*) para cada métrica da cobertura de código em relação à TPR e também o declive (*m*) da linha ajustada da regressão linear. A Figura 29 e a Tabela 10 apresentam os resultados dessa análise.

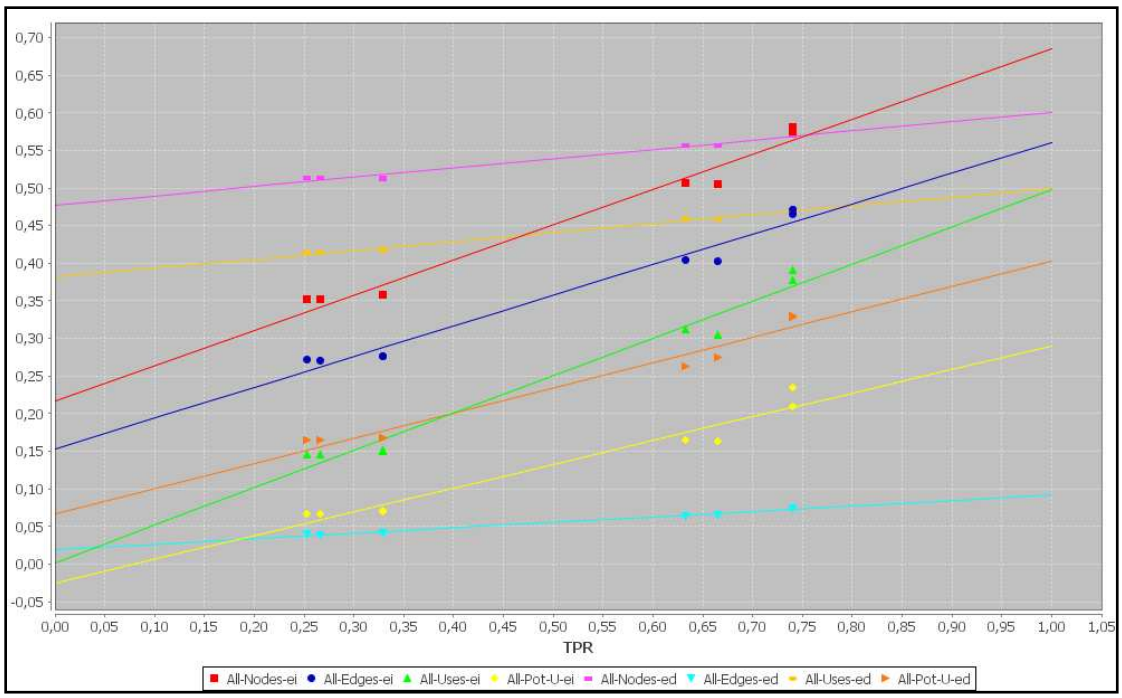

Figura 29 Gráfico de dispersão das métricas de cobertura de código – Rad-WS

Analisando a Figura 29 observamos que os pontos de todas as métricas estão bastante próximos às retas correspondentes, indicando uma alta correlação das métricas de cobertura de código e à TPR. Também verificamos que as métricas *All-Nodes-ed*, *All-Edges-ed* e *All-Uses-ed* apresentam um declive baixo com relação às outras métricas (constatado também nos experimentos da primeira ferramenta), o que significa que esses valores crescem de forma muito diferente em termos absolutos indicando uma baixa correlação à TPR.

Tabela 10 Correlação entra a Cobertura de Código e a TPR - Rad-WS

	All-Nodes		***All-Edges***		***All-Uses***		***All-Pot-Uses***	
	-ei	***-ed***	***-ei***	***-ed***	***-ei***	***-ed***	***-ei***	***-ed***
r	0.**98905**	0.98990	0.98775	0.99424	0.**98839**	0.99441	0.97793	0.98308
m	0.46913	0.**12393**	0.40910	0.**07297**	0.**49720**	0.**11863**	0.31609	0.33726

A Tabela 10 apresenta os resultados das correlações de maneira mais detalhada. A análise mostra que todas as métricas apresentam um coeficiente de correlação bastante alto com relação à TPR, somente a métrica *All-Pot-Uses-ei* fica abaixo de 0.98.

Apesar das métricas *All-Nodes-ed*, *All-Edges-ed* e *All-Uses-ed* apresentarem uma alta correlação à TPR, de acordo com a Tabela 10, elas apresentam um declive (*m*) baixo confirmando o obervado na Figura 29.

Nas métricas *–ei,* a que apresenta a melhor correlação com a qualidade dos testes de detecção de vulnerabilidades é a métrica *All-Nodes-ei*, seguida da métrica *All-Uses-ei*, esta última também apresenta o maior valor de (*m*) indicando maior correlação com a TPR. Reforçamos assim, uma resposta afirmativa para a questão **Q1** do Capítulo 1.

- **Resultado da Cobertura *hotspot***

A cobertura *hotspot* foi dividida pela TPR. É importante notar que na **Erro! Fonte de referência não encontrada.** a escala do eixo vertical foi "ampliada" quando em comparação com os gráficos anteriores.

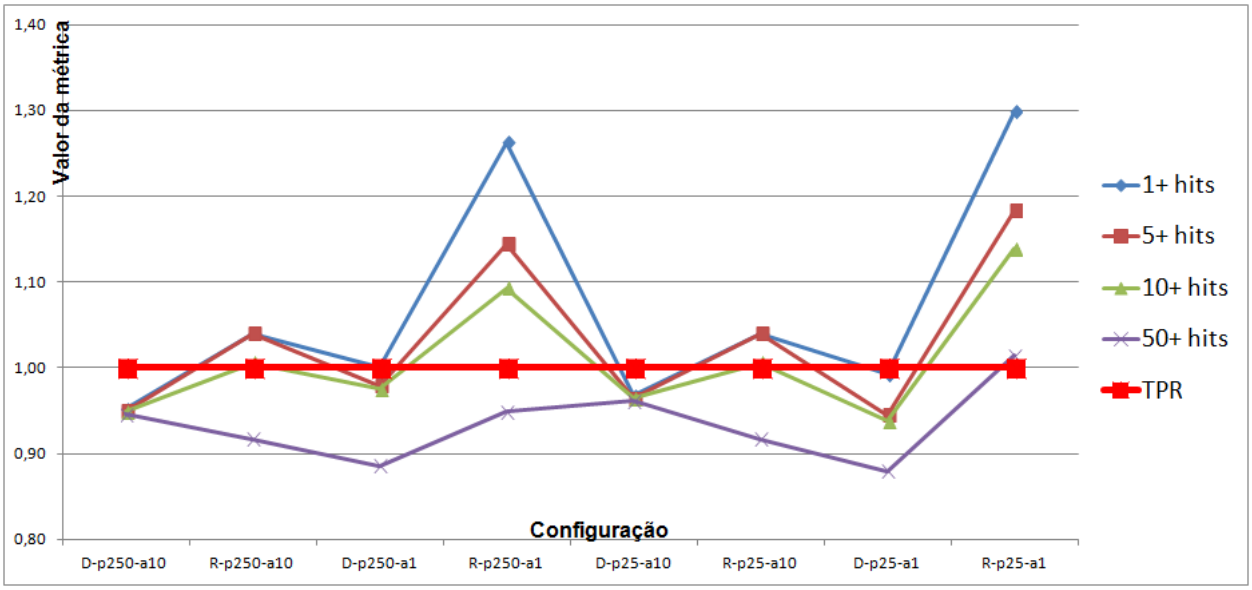

Figura 30 Resultado da métrica de cobertura *hotspot* divida pela respectiva TPR – Rad-WS

As métricas de cobertura *hotspot* parecem ser mais correlacionadas à TPR do que as métricas de cobertura de código. As configurações **D**, como era previsto, são as mais correlacionadas à TPR que as configurações **R**.

Os resultados foram novamente submetidos à correlação de Pearson (*r*) e o declive (*m*) como é possível observar na Figura 31 e Tabela 11.

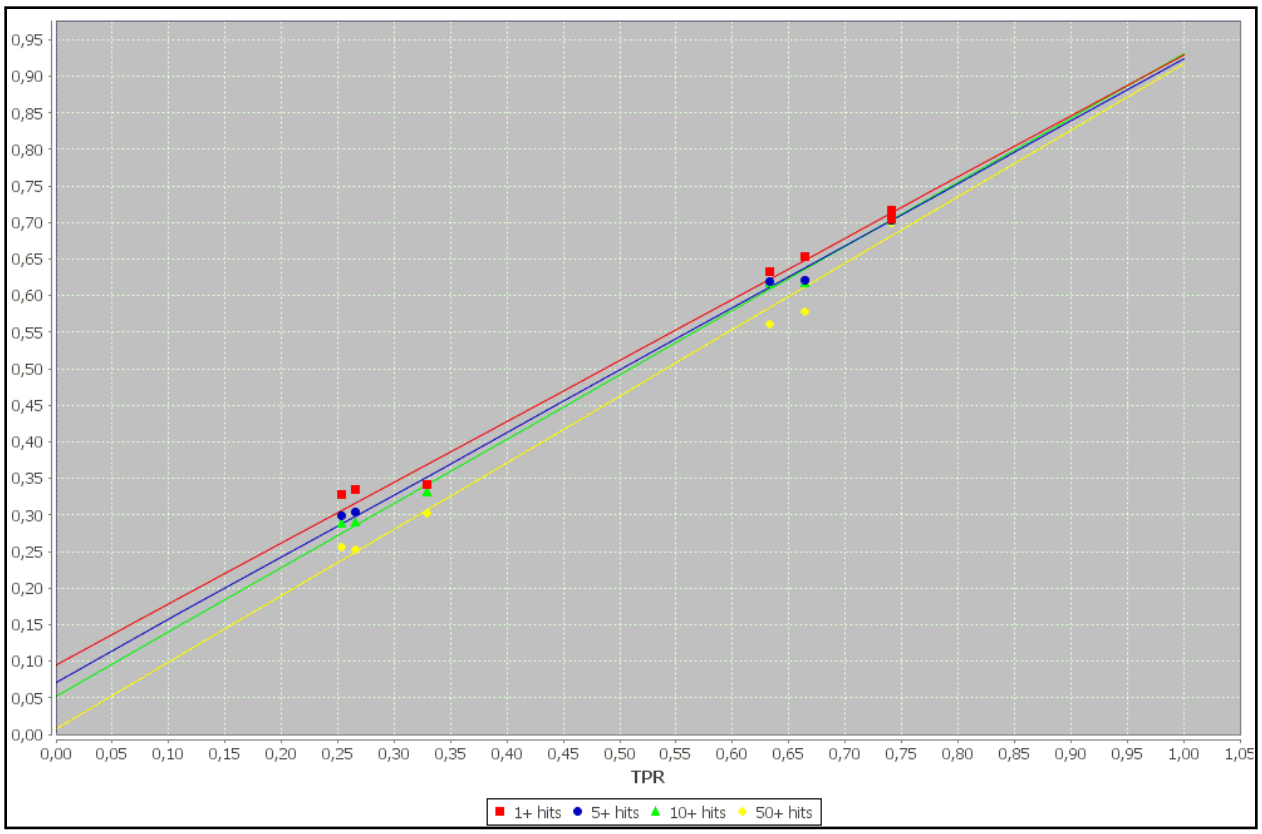

Figura 31 Gráfico de dispersão das métricas de cobertura *hotspot* – Rad-WS

Analisando a Figura 31 e a Tabela 11 confirmamos a nossa observação. Pelo gráfico de dispersão das quatro métricas de cobertura *hotspot* é notável a alta correlação à TPR. É possível observar também pela Tabela 11 que o coeficiente de correlação (*r*) de todas as métricas de cobertura *hotspot* são bastante altos assim como o declive (*m*). Novamente, damos uma resposta afirmativa para a questão **Q2** do Capítulo 1.

Tabela 11 Correlação entra a Cobertura *Hotspot* e a TPR - Rad-WS

	1+hits	***5+hits***	***10+hits***	***50+hits***
r	0.99447	**0.99860**	**0.99845**	**0.99469**
m	0.83729	0.85548	0.88088	**0.91306**

As métricas *5+hits* e *10+hits* apresentam a maior correlação, e a *50+hits* apresenta uma boa correlação e declive. Além de serem as melhores métricas *hotspot*, também são as métricas

mais correlacionadas à TPR de todas as métricas analisadas até aqui. A questão **Q4** do Capítulo 1 é novamente reforçada com os resultados das métricas de cobertura *hotspot*.

- **Serviços agrupados por lógica de negócio**

Os resultados da ferramenta RAD-WS também foram agrupados pela lógica de negócio. A Tabela 12 apresenta os dados de todas as métricas de cobertura *hotspot*, as mesmas métricas analisadas para a ferramenta Sign-WS e também os cálculos da análise de correlação (*r*) e (*m*) para cada agrupamento. Os dados foram agrupados pela configuração e pelos serviços: TPC-App, TPC-C e TPC-W.

Tabela 12 Resultado dividido pela lógica de negócio - RAD-WS

BM	Cfg	*TPR*	*1+ hits*	*5+ hits*	*10+ hits*	*50+ hits*	*AU-ei*	*APU-ed*	*AE-ed*	*APU-ei*
TPC-App	**D-p250-a10**	1,000	1,000	1,000	1,000	1,000	0,781	0,782	0,728	0,667
	R-p250-a10	0,079	0,455	0,455	0,386	0,273	0,369	0,443	0,377	0,310
	D-p250-a1	1,000	0,955	0,955	0,955	0,955	0,753	0,698	0,695	0,636
	R-p250-a1	0,079	0,455	0,273	0,227	0,227	0,361	0,443	0,377	0,304
	D-p25-a10	0,079	0,455	0,455	0,386	0,273	0,371	0,443	0,377	0,310
	R-p25-a10	0,079	0,455	0,455	0,386	0,273	0,371	0,443	0,377	0,310
	D-p25-a1	0,947	0,955	0,955	0,955	0,955	0,747	0,710	0,687	0,627
	R-p25-a1	0,079	0,432	0,273	0,227	0,227	0,361	0,443	0,377	0,304
r		1,0	0,999	0,970	0,980	0,998	0,999	0,989	0,999	0,999
m		1,0	0,575	0,650	0,716	0,791	0,435	0,318	0,362	0,372
TPC-C	**D-p250-a10**	0,506	0,612	0,612	0,612	0,612	0,267	0,297	0,350	0,162
	R-p250-a10	0,325	0,221	0,221	0,221	0,212	0,071	0,135	0,181	0,039
	D-p250-a1	0,301	0,522	0,510	0,510	0,433	0,191	0,225	0,269	0,115
	R-p250-a1	0,205	0,215	0,215	0,205	0,160	0,065	0,132	0,173	0,035
	D-p25-a10	0,494	0,628	0,628	0,628	0,628	0,284	0,300	0,359	0,190
	R-p25-a10	0,313	0,224	0,224	0,224	0,215	0,072	0,135	0,181	0,039
	D-p25-a1	0,373	0,522	0,516	0,513	0,462	0,186	0,238	0,270	0,114
	R-p25-a1	0,241	0,208	0,208	0,202	0,167	0,066	0,132	0,175	0,036
r		1,0	0,795	0,807	0,813	0,890	0,863	0,867	0,888	0,863
m		1,0	1,410	1,419	1,440	1,603	0,750	0,604	0,652	0,503
TPC-W	**D-p250-a10**	1,000	0,907	0,895	0,895	0,884	0,774	0,552	0,182	0,711
	R-p250-a10	0,595	0,721	0,721	0,698	0,640	0,546	0,435	0,580	0,514
	D-p250-a1	1,000	0,884	0,860	0,849	0,837	0,754	0,552	0,674	0,695
	R-p250-a1	0,595	0,721	0,651	0,640	0,605	0,546	0,435	0,580	0,514
	D-p25-a10	1,000	0,907	0,895	0,895	0,884	0,774	0,552	0,683	0,711
	R-p25-a10	0,595	0,721	0,721	0,698	0,640	0,546	0,435	0,580	0,514
	D-p25-a1	1,000	0,872	0,849	0,837	0,826	0,724	0,552	0,665	0,676
	R-p25-a1	0,459	0,721	0,651	0,640	0,605	0,546	0,435	0,580	0,514
r		1,0	0,975	0,967	0,968	0,976	0,973	0,983	-0,095	0,977
m		1,0	0,377	0,427	0,451	0,524	0,463	0,256	-0,065	0,406

Analisando a Tabela 12, observa-se que há uma relação entre as métricas de cobertura de código e a TPR. Observa-se ainda, uma forte relação entre as métricas de cobertura *hotspot* e a TPR.

Para esta análise nota-se que, assim como ocorreu para o estudo de caso anterior: 1) as configurações D-p250-a1 e D-p25-a1 do TPC-C apresentaram TPR e métricas de cobertura

baixas em relação às configurações com domínios pré-definidos dos outros *benchmarks*, isso se deve ao serviço *Order Status* (ver Tabela 5 de serviços); 2) algumas configurações apresentaram valores muito diferentes das configurações com domínios pré-definidos em TPC-App do que em TPC-C e TPC-W, isto ocorre dado a necessidade de dados mais rigorosos nas requisições em TPC-App.

- **Serviços agrupados pela complexidade ciclomática**

Outra análise que será repetida para os resultados da ferramenta RAD-WS é pela complexidade ciclomática, onde os serviços foram agrupados pelas CC entre 2-3; 4-6 e 13-33, conforme a Tabela 13. Foi calculado a correlação de Pearson *(r)* e o declive *(m)* da linha ajustada para cada agrupamento.

Tabela 13 Resultados agrupados pela complexidade ciclomática - RAD-WS

CC	Cfg	*TPR*	*1+ hits*	*5+ hits*	*10+ hits*	*50+ hits*	*AU-ei*	*APU-ed*	*AE-ed*	*APU-ei*
13-33	**D-p250-a10**	0,494	0,602	0,602	0,602	0,602	0,264	0,295	0,343	0,161
	R-p250-a10	0,299	0,207	0,207	0,207	0,197	0,067	0,131	0,173	0,038
	D-p250-a1	0,273	0,510	0,497	0,497	0,418	0,186	0,222	0,261	0,114
	R-p250-a1	0,169	0,201	0,201	0,191	0,148	0,061	0,129	0,165	0,034
	D-p25-a10	0,481	0,618	0,618	0,618	0,618	0,281	0,298	0,353	0,189
	R-p25-a10	0,286	0,211	0,211	0,211	0,201	0,068	0,131	0,173	0,038
	D-p25-a1	0,351	0,510	0,503	0,500	0,447	0,181	0,235	0,262	0,113
	R-p25-a1	0,208	0,194	0,194	0,188	0,155	0,062	0,129	0,167	0,035
r		1,0	0,827	0,838	0,843	0,908	0,883	0,886	0,904	0,882
m		1,0	1,374	1,379	1,401	1,534	0,708	0,571	0,614	0,471
4-6	**D-p250-a10**	0,958	0,909	0,909	0,909	0,894	0,695	0,499	0,676	0,609
	R-p250-a10	0,229	0,515	0,515	0,470	0,379	0,351	0,368	0,391	0,297
	D-p250-a1	0,958	0,879	0,864	0,848	0,848	0,673	0,471	0,650	0,582
	R-p250-a1	0,229	0,515	0,379	0,333	0,318	0,345	0,368	0,391	0,293
	D-p25-a10	0,958	0,909	0,909	0,909	0,894	0,692	0,499	0,676	0,604
	R-p25-a10	0,229	0,515	0,515	0,470	0,379	0,352	0,368	0,391	0,297
	D-p25-a1	0,917	0,879	0,864	0,848	0,848	0,668	0,475	0,643	0,575
	R-p25-a1	0,229	0,500	0,379	0,333	0,318	0,345	0,368	0,391	0,293
r		1,0	0,999	0,975	0,978	0,995	0,999	0,991	0,998	0,999
m		1,0	0,532	0,611	0,664	0,727	0,464	0,165	0,376	0,414
2-3	**D-p250-a10**	1,000	0,972	0,958	0,958	0,958	0,894	0,615	0,753	0,854
	R-p250-a10	0,545	0,750	0,750	0,722	0,667	0,610	0,470	0,625	0,600
	D-p250-a1	1,000	0,944	0,931	0,931	0,917	0,868	0,615	0,742	0,834
	R-p250-a1	0,545	0,750	0,681	0,681	0,639	0,610	0,470	0,625	0,600
	D-p25-a10	1,000	0,972	0,958	0,958	0,958	0,894	0,615	0,753	0,854
	R-p25-a10	0,545	0,750	0,750	0,722	0,667	0,610	0,470	0,625	0,600
	D-p25-a1	1,000	0,931	0,917	0,917	0,903	0,831	0,615	0,731	0,809
	R-p25-a1	0,394	0,750	0,681	0,681	0,639	0,610	0,470	0,625	0,600
r		1,0	0,975	0,977	0,983	0,980	0,973	0,983	0,977	0,977
m		1,0	0,402	0,453	0,476	0,556	0,513	0,285	0,234	0,466

Novamente, observa-se que os resultados são similares aos resultados da ferramenta analisada anteriormente: 1) o agrupamento 13-33 das configurações com domínios pré-definidos, possuem TPR e métricas de cobertura baixas em relação aos outros agrupamentos, o mesmo ocorreu para a análise da lógica de negócio do *benchmark* TPC-C, uma vez que o agrupamento

TPC-C possui os quatro serviços mais complexos; 2) os serviços mais complexos possuem as piores métricas de cobertura; 3) os resultados das configurações randômicas do grupo 4-6 também são baixos, isto se deve ao tipo de requisições de entrada.

Como podemos observar nos resultados das duas ferramentas analisadas até o momento, tanto o agrupamento pela lógica de negócio quanto pela complexidade ciclomática, obtiveram tendências semelhantes, o que reforça a eficácia da abordagem proposta.

- **Cobertura *hotspot* nas fases de aprendizagem e ataque**

Como a ferramenta RAD-WS tem duas diferentes fases (aprendizagem e ataque), fez-se necessária a realização da análise da cobertura *hotspot* de uma maneira diferente com o intuito de confirmar a validade dos resultados obtidos.

Por ter uma abordagem de detecção de vulnerabilidades através da fase de aprendizagem e fase de ataque, onde os conjuntos de teste usados durante a aprendizagem e detecção são baseados em conjuntos de dados diferentes (tal como recomendado nas melhores práticas de *data mining*, *intrusion detection systems*, entre outras), pode ocorrer que na fase de ataques as partes do código executadas não tenham sido executadas na fase de aprendizagem.

A Tabela 14 apresenta o total de comandos *hotspots*, ou comandos SQL, que foram encontrados durante os testes dos *Web Services* para as diferentes configurações. Esta tabela contém o total de comandos *hotspots*, os comandos que foram executados nas duas fases da ferramenta RAD-WS, os comandos que não foram executados em nenhuma das fases, os comandos que foram executados somente na fase de Aprendizagem (Fase WL) e os comandos que foram executados somente na fase de Ataque (Fase AL). A última coluna representa os comandos SQL que não foram aprendidos na primeira fase, esses comandos são nomeados *unlearned,* e como não é possível comparar esse comando com o da fase de aprendizagem, também não é possível afirmar que representem uma vulnerabilidade.

Tabela 14 Quantidade de comandos *hotspots* nas diferentes fases da ferramenta RAD-WS

Configuração	Total	Intersecção	Não executados	Fase WL	Fase AL (*unlearned*)
D-p250-a10	442	**308**	109	20	5
D-p250-a1	442	279	113	48	2
D-p25-a10	442	299	110	14	19
D-p25-a1	442	270	118	42	12
R-p250-a10	442	118	291	0	33
R-p250-a1	442	147	113	**180**	2
R-p25-a10	442	110	280	8	**44**
R-p25-a1	442	110	288	8	36

Observando a Tabela 14 nota-se que a configuração D-p250-a10 obteve o maior número de comandos SQL executados nas duas fases (308 comandos).

A configuração R-p25-a10 apresentou o maior número de *unlearned*. Como a fase de aprendizagem obteve resultados de cobertura de comandos *hotspot* baixos (118 comandos pelos dados da intersecção mais os dados da fase de *workload*), isso se refletiu no número de comandos não aprendidos na fase de ataque. A configuração R-p250-a1 obteve um alto valor de comandos aprendidos somente na fase de aprendizagem (180 comandos) devido a configuração executar um valor muito alto de requisições na fase de aprendizado e um valor muito baixo de requisições na fase de ataque.

A análise desta forma permite providenciar a pessoa responsável pelos testes um grau de confiança acerca da qualidade de cada parte do processo de testes:

1) Os números de comandos não executados mostram que é necessário realizar testes melhores. Para isso, é deve-se usar melhor as informações das requisições com domínios pré-definidos e também combinar os dados de entrada de forma que aumente a cobertura de comandos *hotspots*;
2) Os comandos executados somente na fase de aprendizagem ficaram por testar num contexto de ataques. Isso mostra ao utilizador que a fase de ataque deve ser melhorada;
3) Os comandos *unlearned*, isto é, comandos executados somente na fase de ataques, mostram que a fase de aprendizagem foi incompleta. Isso pode levar a falsos positivos e à redução da confiança nos resultados das ferramentas.

A Figura 32 ilustra a análise da intersecção dos comandos *hotspot* encontrados em duas configurações: D-p250-a10 e R-p250-a10.

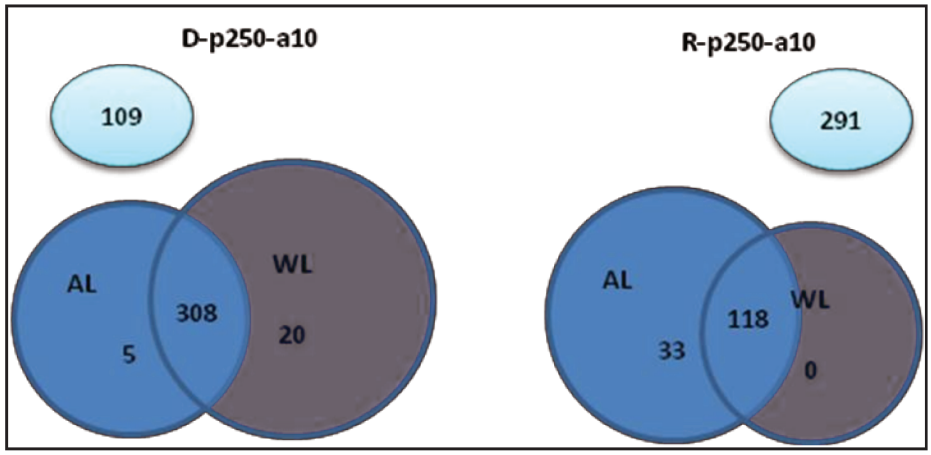

Figura 32 Intersecção dos comandos *hotspot* para as configurações D-p250-a10 e R-p250-a10

Analisando a Figura 32 nota-se que para a configuração que usa domínio pré-definido a fase de aprendizado obteve resultados melhores por possui alto número de requisições e pela qualidade dos inputs usados, o que justifica o número menor de comandos *unlearned* (5 comandos) na fase de ataque. Por outro lado, a configuração randômica executou poucos comandos *hotspot* na fase de aprendizado.

Esta análise realizada pode ser importante para definir a configuração da ferramenta RAD-WS. Caso a fase de aprendizado execute poucos comandos *hotspot* a configuração pode ser alterada antes mesmo de iniciar a fase de ataque, pois os resultados da fase de ataque podem ser comprometidos pela pouca cobertura da fase de aprendizagem, gerando assim muitos *unlearned*.

A próxima seção é dedicada as limitações no que diz respeito à abordagem e os estudos de caso utilizados neste trabalho.

6.3.LIMITAÇÕES

Após a análise dos resultados das ferramentas Sign-WS e RAD-WS, nesta seção iremos discutir alguns pontos que ameaçam a validade de nossos experimentos:

T1. *Workload* dos Serviços: o grupo de serviços que foi usado nos dois estudos de caso vem de outros estudos, onde foram discutidas a sua representatividade (ANTUNES e VIEIRA, 2015). Os serviços são divididos em três implementações independentes. Os resultados podem possuir características específicas devido aos serviços utilizados, limitando assim a validade em outros estudos. Na análise realizada na Seção 6.1 são apresentados os resultados para diversos grupos de serviços, diminuindo assim a ameaça.

T2. Desprezando os falsos positivos: durante nosso estudo descartamos os falsos positivos, que no contexto de detecção de vulnerabilidades é bem conhecido. No entanto, ferramentas de detecção de vulnerabilidade têm vários módulos, alguns responsáveis pela geração de testes e outros responsáveis por identificar vulnerabilidades. A questão de falsos positivos depende quase que exclusivamente do módulo de identificação de vulnerabilidade e, neste estudo focamos apenas na qualidade dos testes executados, e, assim, sobre o módulo de geração de teste.

T3. Ferramenta de detecção de vulnerabilidade: os resultados podem ser influenciados pelas características das ferramentas utilizadas. As ferramentas Sign-WS e RAD-WS são consideradas como estado da arte no seu domínio e os autores argumentam que os módulos de identificação de vulnerabilidade implementados são "praticamente perfeitos". Eles não discutem o mesmo sobre as técnicas de geração de teste, portanto, esta questão relaciona-se com T2. Outro ponto a ressaltar é que as duas ferramentas utilizadas no estudo de caso (Sign-WS e RAD-WS) foram desenvolvidas pelo grupo de pesquisa em Tolerância a Falhas da Universidade de Coimbra. Acreditamos que essa ameaça deve ser objeto de um estudo futuro que engloba a avaliação e comparação de várias ferramentas de outros grupos de pesquisa ou comerciais.

T4. Ferramenta e critérios de cobertura: como utilizamos uma ferramenta que é produto de investigação, estamos sujeitos a erros de medição que podem afetar os resultados. Os resultados também podem ser válidos somente para os critérios considerados pela ferramenta de cobertura de código selecionada. Porém se a ferramenta utilizada possui pouca confiança no que diz respeito a sua maturidade, por outro lado, aumenta a nossa confiança por usar o estado da arte dos critérios. Em relação a outros critérios, este deve ser objeto de uma análise mais aprofundada, os critérios de cobertura devem ser avaliados de forma sensata e também este tipo de experiência devem ser executados antes de outras hipóteses sobre ouros critérios.

7. CONCLUSÃO

Diante dos desafios de manter seus sistemas computacionais protegidos, cada vez mais, as empresas buscam por ferramentas que mantenham seus dados em segurança. Este trabalho visa proporcionar outros meios para que a segurança e dependabilidade sejam alcançadas pelos produtos de software.

O objetivo principal deste trabalho é confirmar a aplicabilidade da análise de cobertura de código como um meio para compreender a qualidade de ferramentas de detecção de vulnerabilidade. Esta necessidade surgiu após realizar o levantamento e estudo desses dois tipos de ferramentas, as que realizam a detecção de vulnerabilidades e as que realizam a cobertura de código, e identificar que essas funções podem ser realizadas em uma única abordagem.

Para isso, foi projetada uma abordagem que se resume em seis fases: (1) **Carga de trabalho -** define a carga de trabalho que um conjunto de serviços deve executar; (2) **Instrumentação –** "sondas" são inseridas no código dos serviços e são coletadas informações para a medição da cobertura *hotspot*; (3) **Configuração -** definição das configurações dos testes que serão executados; (4) **Testes -** ferramenta de detecção de vulnerabilidade que executa os serviços; (5) **Monitoramento** - as "sondas" inseridas na etapa (2) geram arquivos *traces* que contêm informações do código executado e, por fim, (6) **Análise -** os arquivos *traces* e *hotspots* são analisados para se verificar a cobertura alcançada pelas ferramentas de detecção de vulnerabilidades.

Para realizar a cobertura de código foi selecionada a ferramenta JaBUTi (VINCENZI *et al.*, 2003) que apresenta os critérios de teste de fluxo de controle: *All-Nodes-ei, All-Edges-ed, All-Edges-ei, All-Nodes-ed*; e os critérios de fluxo de dados: *All-Uses-ei, All-Uses-ed, All-Pot-Uses-ei, All-Pot-Uses-ed*. A cobertura *hotspot*, inspirada na cobertura de instrução destino (SMITH *et al.,* 2008), é definida como a porcentagem das possíveis vulnerabilidades existentes que a ferramenta detecta e foi separada nos seguintes critérios: *1+hits, 5+hits, 10+hits e 50+hits*.

A partir da abordagem proposta, foram realizados dois estudos de caso com duas diferentes ferramentas de detecção de vulnerabilidades consideradas estado da arte em detecção de vulnerabilidades. São elas: Sign-WS e RAD-WS. Para a ferramenta Sign-WS foram executados testes para quatro diferentes configurações e para ferramenta RAD-WS foram

executados oito configurações. Um conjunto de 21 serviços que contêm 158 vulnerabilidades de injeção SQL conhecidas foram submetidos aos testes das ferramentas de detecção de vulnerabilidades.

Os resultados para as duas ferramentas mostraram que as configurações com requisições de domínio pré-definidos obtiveram resultados melhores do que as configurações randômicas, alcançando altos valores das métricas de cobertura de código e cobertura *hotspot* e também altos valores da TPR.

Através dos resultados foi possível identificar uma relação entre as métricas de cobertura e o número de vulnerabilidades relatadas, trazendo fortes indícios de que os indicadores de cobertura podem ser úteis para comparar a eficácia de diferentes conjuntos de testes de detecção de vulnerabilidades e também que, sob as condições corretas, métricas de cobertura *hotspot* podem servir como uma estimativa para as vulnerabilidades menos detectadas. Sendo assim, os resultados sugerem uma certa correlação ou mesmo alguma proporcionalidade entre as métricas de cobertura de código e coberta *hotspot* e a qualidade dos testes de detecção de vulnerabilidade.

Este trabalho propôs responder quatro questões que fazem parte dos objetivos, que são:

Q1. Existe uma relação entre os valores de cobertura de código e o número de vulnerabilidades reportado?

Os resultados apresentados pelas ferramentas Sign-Ws e RAD-WS mostraram que existe uma relação entre a cobertura de código e número de vulnerabilidades detectadas. Pela análise de correlação de Pearson *(r)* e o declive *(m)* foi possível verificar que as métricas *–ed* são as menos correlacionadas à TPR, pois códigos que contém vulnerabilidades geralmente não estão localizados nos blocos dependentes de exceções. Nas duas ferramentas a métrica *All-Uses-ei* obteve bons resultados, o que levam a uma resposta afirmativa a esta questão.

Q2. Existe uma relação entre os valores da cobertura hotspot e o número de vulnerabilidades reportado?

Os estudos de caso indicaram que a relação entre a cobertura hotspot e o número de vulnerabilidades reportadas é ainda maior que a cobertura de código. Pela análise de correlação de Pearson a métrica *1+hit* apresentou o menor valor para as duas ferramentas, sendo 0,99634 para a ferramenta Sign-WS e 0.99447 para a RAD-WS. Mesmo sendo a métrica com o valor mais baixo de todas as métricas de cobertura *hotspot*, é um valor alto, o que indica alta correlação à TPR.

Q3. Métricas de cobertura de código e cobertura *hotspot* são eficazes para comparar a qualidade de diferentes testes de detecção de vulnerabilidades?

Pela análise geral dos resultados das duas ferramentas obtivemos uma resposta afirmativa para esta questão, pois verificamos que para as diferentes configurações das ferramentas de detecção de vulnerabilidades obtivemos diferentes resultados. As configurações com domínios pré-definidos, como era esperado, obtiveram resultados melhores para ambas as coberturas e as configurações randômicas obtiveram coberturas inferiores. Sendo assim, as métricas de cobertura de código e cobertura *hotspot* corroboram para avaliar a qualidade de diferentes testes de detecção de vulnerabilidades.

Q4. Métricas de cobertura *hotspot* são eficazes para estimar a eficácia de uma ferramenta de detecção de vulnerabilidades?

As métricas de cobertura de código se mostraram bastantes promissoras por apresentarem alta correlação à TPR. Tais observações nos levam a argumentar que é importante definir métricas de cobertura que são específicas para o domínio de testes de detecção de vulnerabilidade. Essas métricas podem constituir de fato uma forma de estimar a eficácia de uma ferramenta de detecção de vulnerabilidades.

Trabalhos futuros podem propor uma abordagem para avaliar a confiabilidade de ferramentas de detecção de vulnerabilidade, eventualmente sob a forma de novos *benchmarks* para o domínio. Inclui também utilizar estas informações para melhorar as abordagens de detecção de vulnerabilidade, principalmente através da melhoria contínua dos testes realizados.

Diante as limitações apresentadas, é proposto também que a abordagem seja executada utilizando um conjunto de *Web Services* que representem outros segmentos, permitindo assim que os testes verifiquem diferentes tipos de serviços. E também que os testes sejam executados com outras ferramentas de detecção de vulnerabilidades de outros grupos de pesquisa ou comerciais, o que corrobora para a ideia de independência da abordagem em relação à ferramenta DV.

Outro trabalho futuro relacionado as métricas de cobertura *hotspot* que demonstraram ser bastante promissoras é propor métricas melhores que levem em conta o número de testes realizados, o número de entradas testadas, entre outros. Assim como a automatização do procedimento proposto para a abordagem na combinação de *hotspots* que é realizada manualmente. Este processo pode ser automatizado utilizando processamento de texto.

REFERÊNCIAS

ACUNETIX. ***Acunetix Web Application Security***. Disponível em: <http://www.acunetix.com>. Acesso em: 5 de novembro de 2013.

AHO, A. V.; LAM, M. S.; ULLMAN J. D. ***Compilers: Principles, Techniques, and Tools***, Pearson Education, Inc, 2006.

ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. ***Web services***. Springer Berlin Heidelberg, p. 123-149. 2004.

ANTUNES, N.; VIEIRA, M. ***Benchmarking Vulnerability Detection Tools for Web Services***. *IEEE Eighth International Conference on Web Services* (ICWS 2010), Miami, Florida, Estados Unidos. 2010.

ANTUNES, N.; VIEIRA, M. ***Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services***. *IEEE Int'l Conf. on Services Computing*, 104-111. 2011.

ANTUNES, N.; VIEIRA, M., ***Attack Signatures and Interface Monitoring to Detect Injection Vulnerabilities in Web Services***. Disponível em: <http://eden.dei.uc.pt/~mvieira/signws.zip>. Acesso em: 20 de julho de 2013.

ANTUNES, N.; VIEIRA, M. "***Penetration Testing for Web Services***," Computer, vol. 47, no. 2, pp. 30–36, Feb. 2014.

ANTUNES, N.; VIEIRA, M. "***Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples***," IEEE Trans. Serv. Comput., vol. 8, no. 2, pp. 269–283, 2015.

ANTUNES, N; LARANJEIRO, N.; VIEIRA, M.; MADEIRA, H. ***Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services***. *IEEE International Conference on Services Computing* (SCC 2009), Bangalore, India. 2009.

ASM. ***OW2 Consortium***. Disponível em: <http://asm.ow2.org/> Acesso em: 05 de janeiro de 2014.

AVIZIENIS, A.; LAPRIE, J.C.; RANDELL, B. ***Fundamental concepts of dependability***. *In Third Information Survivability Workshop*, Boston, MA, 2000.

BASSO, T.; FERNANDES, P.C.S.; JINO, M.; MORAES, R.L.O. ***Analysis of the effect of Java software faults on security vulnerabilities and their detection by commercial Web vulnerability scanner tool***. *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on IEEE*, 2010.

BASSO, T. **Uma abordagem para avaliação da eficácia de *scanners* de vulnerabilidades em aplicações Web**. Dissertação de Mestrado. Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, Campinas, Brasil. 2010.

BASSO, T.; MORAES, R. L. O.; JINO, M. ***A Methodology for Effectiveness Analysis of Vulnerability Scanning Tools***. III EADCA - Terceiro Encontro dos Alunos e Docentes do Departamento de Engenharia de Computação e Automação Industrial. Universidade Estadual de Campinas. 2010.

BCEL. **Apache Commons**. Disponível em: <.http://commons.apache.org/proper/commons-bcel/> Acesso em: 05 de janeiro de 2014.

BONETT, D. G.; WRIGHT, T. A. "***Sample size requirements for estimating pearson, kendall and spearman correlations***" Psychometrika, vol. 65, no. 1, pp. 23–28, 2000.

BRUNETON, E.; LENGLET, R.; COUPAYE, T. ***ASM: a code manipulation tool to implement adaptable systems***. In *Journes Composants* 2002 (JC'02). 2002.

CAI, X.; LYU, M. R. ***The effect of code coverage on fault detection under different testing Profiles***. *In First International Workshop on Advances in Model-based Testing,* p. 1-7, St. Louis, Missouri, Estados Unidos, 2005.

CAMPWOOD. Campwood Software, "***SourceMonitor Version 2.5***". Disponível em: <http://www.campwoodsw.com/sourcemonitor.html>. Acesso em: 3 de março de 2015.

CLOVER. ***CLOVER***. Disponível em: <https://www.atlassian.com/software/clover/overview>. Acesso em: 07 de outubro de 2015.

COBERTURA. ***Cobertura: A code coverage utility for Java***. Disponível em: <cobertura.github.io/cobertura>. Acesso em: 07 de outubro de 2015.

CORNETT, Steve. ***Code Coverage Analysis. Bullseye Testing Technology***. Disponível em: <http://www.bullseye.com/coverage.html>. Acesso em: 30 de outubro de 2013.

COVERITY. ***Software Development Testing and Static Analysis Tools***. Disponível em: <http://www.coverity.com/products/security-advisor.html>. Acesso em: 9 de novembro de 2013.

CRESPO, A. N.; PASQUINI A.; JINO, M.; MALDONADO, J. C. **Cobertura dos critérios potenciais-usos e a confiabilidade do software**. 9th SBES, p. 379-394, 1997.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. Ed. Campus, 2007.

DOUPÉ, A.; CAVEDON, L.; KRUEGEL, C.; VIGNA, G. "***Enemy of the State:A State-Aware Black-Box Web Vulnerability Scanner***," in 21st USENIX Security Symposium (USENIX Security '12). Bellevue, WA: USENIX, 2012, pp. 523–538.

DOUPÉ, A.; COVA M.; VIGNA, G. "***Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners***" in Detection of Intrusions and Malware, and Vulnerability Assessment, Springer Berlin Heidelberg, 2010, pp. 111–131.

DURÃES, J.; MADEIRA, H. "***Emulation of Software Faults: A Field Data Study and Practical Approach***". *IEEE Trans. on Software Engineering*, vol. 32, n. 11, Nov. 2006, pp.849-867.

ECLIPSE. ***Getting Started with Eclipse.*** Disponível em: <https://eclipse.org/>. Acesso em: 13 de dezembro de 2015.

EMMA. **EMMA**. Disponível em: <http://emma.sourceforge.net/>. Acesso em: 30 de outubro de 2013.

EMMA. ***EMMA: a free Java code coverage tool.*** Disponível em: <http://emma.sourceforge.net/>. Acesso em: 07 de outubro de 2015.

EXAME. **EXAME.com**. Disponível em: <http://exame.abril.com.br/tecnologia/noticias/adobe-estima-38-milhoes-de-afetados-por-ataque-de-hackers>. Acesso em: 5 de novembro de 2013.

FERNANDES, P. C. S.; BASSO, T.; MORAES, R. **J-Attack - Injetor de Ataques para Avaliação de Segurança de Aplicações Web**. In: Workshop de Testes e Tolerância a Falhas, 2011, Campo Grande - MT. Proceedings of WTF 2011, 2011.

FINDBUGS. ***FindBugs™ - Find Bugs in Java Programs***. Disponível em: <http://findbugs.sourceforge.net/>. Acesso em: 14 de jabeiro de 2016.

FONSECA, J.; VIEIRA, M. ***Mapping software faults with Web security vulnerability***. IEEE/IFIP *Int. Conf. on Dependable Systems and Networks* (DSN 2008), Anchorage, Estados Unidos. 2008.

FONSECA, J.; VIEIRA, M.; MADEIRA, H. ***Testing and comparing Web vulnerability scanning tools for SQL injection and XSS attacks***. *Pacific Rim Int'l Symp. Dependable Computing*, IEEE, vol. 0, p. 365–372. 2007.

FRANKL, P. G.; WEYUKER, E. J. ***An applicable family of data flow testing criteria***. *Software Engineering, IEEE Transactions on,* v. 14, n. 10, p. 1483-1498, 1988.

HALFOND, W.; ORSO, A. "***Command-Form Coverage for Testing Database Applications***," in 21st IEEE/ACM International Conf. on Automated Software Engineering. ASE'06. IEEE, 2006, pp. 69–80.

HORGAN, J. R.; LONDON, S.; LYU, M. R. ***Achieving software quality with testing coverage measures***. IEEE Computer, v. 27, n. 9, p. 60-69, 1994.

HP. **HP Official Site**. Disponível em: <http://www8.hp.com/br/pt/software-solutions/software.html?compURI=1341991#.UoER2BBu5cE>. Acesso em: 8 de novembro de 2013.

HP. **HP Official Site**. Disponível em: <http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#tab=TAB1>. Acesso em: 8 de novembro de 2013.

HSUEH, M.C.; TSAI, T.; IYER. R.; ***Fault Injection Techniques and Tools***. In *IEEE Computer*, p. 75-82, 1997.

IBM. ***International Business Machines***. Disponível em: <http://www-03.ibm.com/software/products/en/appscan-source>. Acesso em: 9 de novembro de 2013.

IBM. ***International Business Machines***. Disponível em: <http://www.ibm.com/developerworks/rational/library/10/introtocodecoveragetoolinrationalapplicationdeveloper/>. Acesso em: 5 de novembro de 2013.

ISO/IEC 17799. ***Information technology - code of practice for Information security management***. Switzerland: International Organization for Standardization (ISO); 2000.

JAVASSIST. **Javassist**. Disponível em: <.http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/> Acesso em: 05 de janeiro de 2014.

JMETER. **Apache JMeter**. Disponível em: <. http://jmeter.apache.org/> Acesso em: 20 de janeiro de 2014.

LAPRIE, J. C. ***Dependability – Its Attributes, Impairments and Means***. *In Predictability Dependable Computing Systems*, B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, Editors, Springer, Berlim, Alemanha, 1995.

LEITE, J.; ORLANDO, G. **Software**. In II SCTF, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, Brasil, 1987.

LEME, N. G. M. **Um sistema de padrões para injeção de falhas por software**. Dissertação de Mestrado - Universidade Estadual de Campinas, Campinas, Brasil. 2001.

LINDHOLM, T.; YELLIN, F.; BRACHA, G.; BUCKLEY, A. ***The Java virtual machine specification***. Addison-Wesley. 2013.

LYU, Ed. M. R. ***Handbook of software reliability engineering***. Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

MALDONADO, J. C; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. R. S.; DELAMARO, M. E. **Aspectos teóricos e empíricos de teste de cobertura de software**. Notas Didáticas do Instituto de Ciências Matemáticas de São Carlos, n° 31, São Carlos. 1998.

MCCABE. "***Combining McCabe IQ with Fuzz Testing***" McCabe Software, Inc., Tech. Rep., 2009. Disponível em: <http://www.mccabe.com/pdf/McCabeIQ-FuzzTesting.pdf>. Acesso em: 20 de julho de 2015.

MEDINA, M.; FERTIG, C. **Algoritmos e Programação – teoria e prática – 2ª edição**. São Paulo: Novatec, 2006

MYERS, G. J.; SANDLER, C.; BADGETT, T.. ***The art of software testing***. John Wiley & Sons, 2011.

OW2. ***The open source community for infrastructure software - Bytecode Outline Plugin for Eclipse***. Disponível em: <http://asm.ow2.org/eclipse/>. Acesso em: 23 de setembro de 2013.

OWASP. **Owasp Top 10 Project - 2013**. Disponível em: <https://www.owasp.org/index.php/Top_10_2013>. Acesso em: 01 julho de 2013.

SILVA, P. A. L. D. **Princípios dos Métodos Estatísticos: conceitos, modelos e aplicações no Excel.** *Editora Universitária Santa Úrsula, RJ*. 1998.

PLAYSTATION. **PlayStation Blog**. Disponível em: <http://blog.us.playstation.com/2011/04/26/update-on-playstation-network-and-qriocity/>. Acesso em: 24 setembro de 2012.

PORTSWIGGER. ***Portswigger Web Security***. Disponível em: <http://portswigger.net/burp/>. Acesso em: 8 de novembro de 2013.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. 7ª Edição. Bookman, 2011.

RAPPS, S.; WEYUKER, E. J. ***Data flow analysis techniques for test data selection***. *In: Proceedings of the 6th international conference on Software engineering. IEEE Computer Society Press.* p. 272-278, 1982.

RAPPS, S.; WEYUKER, E. J**.** ***Selecting software test data using data flow information***. *Software Engineering, IEEE Transactions on Software Engineering*, p. 367-375, 1985.

SANCHES B.; BASSO, T.; MORAES, R. ***J-SWFIT: A Software Fault Injection Tool***. *IN: Proc. of The Fifth Latin-American Symposium on Dependable Computing* – LADC, 2011.

SKRUCH, P. ***A coverage metric to evaluate tests for continuous-time dynamic systems***. *Central European Journal of Engineering*, v. 1, n. 2, p. 174-180, 2011.

SMITH, B.; SHIN, Y.; WILLIAMS , L. "***Proposing SQL Statement Coverage Metrics***," in Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, ser. SESS '08. New York, NY, USA: ACM, 2008, pp. 49–56.

SPIEGEL, M. R. **ESTATÍSTICA**. 3. ED. São Paulo: Makron Books. Coleção Schaum 643 P. 1994.

STUTTARD, D.; PINTO, M. ***The web application hacker's handbook: discovering and exploiting security flaws***. Wiley Publishing, Inc., 2007.

TPC. "***Transaction Processing Performance Council***," 2009. Disponível em: <http://www.tpc.org/>. Acesso em: 3 de março de 2015.

VANOVERBERGHE, D.; HALLEUX, J. D.; TILLMANN, N.; PIESSENS, F. ***State coverage: software validation metrics beyond code coverage***. In: SOFSEM 2012: *Theory and Practice of Computer Science*. Springer Berlin Heidelberg, p. 542-553, 2012.

VIEIRA, M.; ANTUNES, N.; MADEIRA, H. ***Using Web Security Scanners to Detect Vulnerabilities in Web Services***. *In: Proceedings of the Conference on Dependable Systems and Networks*. 2009.

VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M.E.; MALDONADO, J. C. ***JaBUTi: A coverage analysis tool for Java programs***. XVII SBES–Simpósio Brasileiro de Engenharia de Software, p. 79-84, 2003.

VINCENZI, A. M. R.; DELAMARO, M. E.; MALDONADO, J. C.; WONG, W. E. "***Establishing structural testing criteria for Java bytecode***" Software: Practice and Experience, vol. 36, no. 14, pp. 1513–1541, 2006.

YANG, Q.; LI, J. J.; WEISS, D. M. ***A survey of coverage-based testing tools***. *The Computer Journal*, v. 52, n. 5, p. 589-597, 2009.

Anexo 1 – Código fonte de um serviço sob teste

```
package pt.uc.dei.wsvdbench.tpcw.versions;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
import pt.uc.dei.wsvdbench.tpcw.object.Book;
import pt.uc.dei.wsvdbench.util.Database;

public class DoSubjectSearch_Vx0
{
  public List<Book> doSubjectSearch(String search_key)
  {
    List<Book> vec = new ArrayList();
    Connection con = Database.pickConnection();
    try
    {
     PreparedStatement statement = con.prepareStatement("SELECT * FROM tpcw_item, tpcw_author WHERE tpcw_item.i_a_id = tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM <= 50 ORDER BY tpcw_item.i_title ");

      statement.setString(1, search_key);
      ResultSet rs = statement.executeQuery();
      while (rs.next()) {
        vec.add(new Book(rs));
      }
      rs.close();
      statement.close();
      con.commit();
    }
    catch (Exception ex) {}finally
    {
      Database.relaseConnection(con);
    }
    return vec;
  }
}
```

Anexo 2 – Código fonte de um serviço sob teste após a instrumentação pela ferramenta JaBUTi. Notar novos comandos com "sondas" que irão gerar o arquivo *trace* com dados da cobertura de código.

```
package pt.uc.dei.wsvdbench.tpcw.versions;

import br.jabuti.probe.DefaultProber;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
```

```
import pt.uc.dei.wsvdbench.tpcw.object.Book;
import pt.uc.dei.wsvdbench.util.Database;

public class DoSubjectSearch_Vx0
{
  public DoSubjectSearch_Vx0()
  {
    DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 0, 0L,
"0");DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 0, 0L, "4");
  }

  public List<Book> doSubjectSearch(String arg1)
  {
    long l = DefaultProber.getNest();DefaultProber.probe(this,
"pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "0");List vec = new ArrayList();
    Connection con = Database.pickConnection();
    try
    {
      DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l,
"12");PreparedStatement statement = con.prepareStatement("SELECT * FROM tpcw_item, tpcw_author WHERE
tpcw_item.i_a_id = tpcw_author.a_id AND tpcw_item.i_subject = ? AND ROWNUM <= 50 ORDER BY
tpcw_item.i_title ");

      statement.setString(1, search_key);
      ResultSet rs = statement.executeQuery();
      for (;;)
      {
        DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "40");
        if (!rs.next()) {
          break;
        }
        DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "50");vec.add(new
Book(rs));
      }
      DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "69");rs.close();
      statement.close();
      con.commit();


      DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l,
"89");Database.relaseConnection(con);
    }
    catch (Exception statement) {}finally
    {
      DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "105");localObject =
        finally;DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l,
"107");Database.relaseConnection(con);
    }
    DefaultProber.probe(this, "pt.uc.dei.wsvdbench.tpcw.versions.DoSubjectSearch_Vx0", 1, l, "114");return vec;
  }
}
```