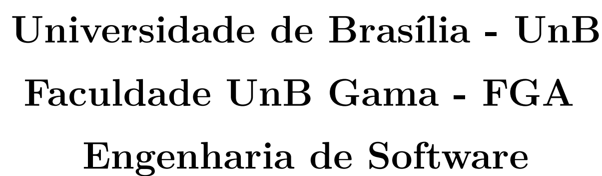
Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software



Modelos de Predição para Monitoramento de Métricas de Ameaças de Vulnerabilidade de Código-fonte

Autor: Lucas Kanashiro Duarte
Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF
2015

Lucas Kanashiro Duarte

Modelos de Predição para Monitoramento de Métricas de Ameaças de Vulnerabilidade de Código-fonte

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2015

Lucas Kanashiro Duarte

Modelos de Predição para Monitoramento de Métricas de Ameaças de Vulnerabilidade de Código-fonte/ Lucas Kanashiro Duarte. – Brasília, DF, 2015-

102 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. Métrica, Vulnerabilidade. 2. Código-fonte, Predição. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Modelos de Predição para Monitoramento de Métricas de Ameaças de Vulnerabilidade de Código-fonte

CDU 02:141:005.6

Lucas Kanashiro Duarte

Modelos de Predição para Monitoramento de Métricas de Ameaças de Vulnerabilidade de Código-fonte

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 15 de Julho de 2015:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Prof. Dr. Nilton Correia da Silva
Convidado 1

Prof. Dr. Tiago Alves Fonseca
Convidado 2

Brasília, DF
2015

Resumo

Devido a constante evolução da Engenharia de Software, a cada dia surgem novas linguagens de programação, paradigmas de desenvolvimento, formas de avaliar processos, entre outras coisas. Com as métricas de código-fonte não é diferente, com o passar do tempo surgem outras classes de métricas e para a utilização das mesmas vem a necessidade de se saber como utiliza-las. Para a utilização de uma métrica de software qual for, é necesário ter conhecimento sobre como realizar a coleta, cálculo, interpretação e análise para tomada de decisões. No contexto das métricas de código-fonte, a coleta e cálculo na maioria das vezes são automatizadas por ferramentas, mas como acompanhá-las e monitorá-las de maneira correta no decorrer do ciclo de desenvolvimento de software? Este trabalho visa auxiliar o Engenheiro de Software a monitorar e acompanhar métricas de ameaças de vulnerabilidade de código-fonte através de um modelo de predição de referência, tendo em vista que cada vez mais os softwares possuem requisitos não funcionais de segurança, o que leva a necessidade de saber como monitorar esses requisitos durante o ciclo de desenvolvimento de software.

Palavras-chaves: Métricas; Vulnerabilidade; Código-fonte; Predição.

Abstract

Due to constant evolution of software engineering, each day brings new programming languages, development paradigms, ways of evaluating processes, among other things. The source code metrics does not be different, with the passage of time arise other metrics classes and the use thereof has the need to know how to use them. For the use of any software metric is necessary to have knowledge about how to perform the collection, calculation, interpretation and analysis for decision. In the context of source code metrics, collecting and calculating most often are automated by tools, but how should we monitor them during the software development cycle? This work aims to assist the Software Engineer to monitor metrics of vulnerability threats of source code through a reference prediction model, considering that increasingly softwares have security non functional requirements, which leads to the need to know how monitor these requirements during the software development cycle.

Key-words: Metrics; Vulnerability; Source code; Prediction.

Lista de ilustrações

Figura 1 – Diagrama de blocos de uma ferramenta genérica (CHESS; WEST, 2007) 32
Figura 2 – Exemplo de *Parse Tree* (CHESS; WEST, 2007) 34
Figura 3 – Exemplo de *AST* (CHESS; WEST, 2007) 34
Figura 4 – Sequência de atividades dos métodos de análise (WASIAK, 2012) . . . 37
Figura 5 – Exemplo *4-Plot* 39
Figura 6 – Explicação sobre identificação de *outliers* com *boxplot* 40
Figura 7 – Processo contendo atividades executadas durante a pesquisa 46
Figura 8 – *Plot* das taxas de CWE pelo número total de módulos 50
Figura 9 – CWE476 - 4 *plots* 51
Figura 10 – Histograma da taxa da CWE476 52
Figura 11 – *Q-Q plot* da taxa da CWE476 52
Figura 12 – *Run Sequence* da taxa da CWE476 53
Figura 13 – *Lag Plot* da taxa da CWE476 54
Figura 14 – CWE457 - 4 *plots* 54
Figura 15 – Histograma da taxa da CWE457 55
Figura 16 – *Q-Q Plot* da taxa CWE457 56
Figura 17 – *Run Sequence* da taxa CWE457 56
Figura 18 – *Lag Plot* da taxa CWE457 57
Figura 19 – *Boxplot* da taxa da CWE476 por módulo 60
Figura 20 – Curva gerada pelo método *LOESS* sobre os dados da taxa da CWE476 61
Figura 21 – Curva de predição do modelo quadrático dos dados da CWE476 62
Figura 22 – Curva de predição do modelo cúbico dos dados da CWE476 62
Figura 23 – *Boxplot* da taxa da CWE457 por módulo 63
Figura 24 – Curva gerada pelo método *LOESS* sobre os dados da taxa da CWE457 64
Figura 25 – Curva de predição do modelo quadrático dos dados da CWE457 65
Figura 26 – Curva de predição do modelo cúbico dos dados da CWE457 65
Figura 27 – Curva de predição dos modelos dos dados da CWE476 66
Figura 28 – Validação cruzada *Ten-fold* do modelo quadrático da taxa da CWE476 67
Figura 29 – Validação cruzada *Ten-fold* do modelo cúbico da taxa da CWE476 . . 68
Figura 30 – Curva de predição dos modelos dos dados da CWE457 69
Figura 31 – Validação cruzada *Ten-fold* do modelo quadrático da taxa da CWE457 70
Figura 32 – Validação cruzada *Ten-fold* do modelo cúbico da taxa da CWE457 . . 70
Figura 33 – Erro de predição de modelos não lineares para a CWE476 74
Figura 34 – Erro de predição de modelos não lineares para a CWE457 75
Figura 35 – Gráfico de Percentis da métrica AN 87
Figura 36 – Gráfico de Percentis da métrica ASOM 88

Figura 37 – Gráfico de Percentis da métrica AUV 88
Figura 38 – Gráfico de Percentis da métrica BD 89
Figura 39 – Gráfico de Percentis da métrica BF 89
Figura 40 – Gráfico de Percentis da métrica DBZ 90
Figura 41 – Gráfico de Percentis da métrica DF 90
Figura 42 – Gráfico de Percentis da métrica DNP 91
Figura 43 – Gráfico de Percentis da métrica DUPV 91
Figura 44 – Gráfico de Percentis da métrica FGBO 92
Figura 45 – Gráfico de Percentis da métrica MLK 92
Figura 46 – Gráfico de Percentis da métrica OBAA 93
Figura 47 – Gráfico de Percentis da métrica OSF 93
Figura 48 – Gráfico de Percentis da métrica PITFC 94
Figura 49 – Gráfico de Percentis da métrica ROGU 94
Figura 50 – Gráfico de Percentis da métrica RSVA 95
Figura 51 – Gráfico de Percentis da métrica SAIGV 95
Figura 52 – Gráfico de Percentis da métrica UAF 96
Figura 53 – Gráfico de Percentis da métrica UA 96
Figura 54 – Gráfico de Percentis da métrica UAV 97

Lista de tabelas

Tabela 1 – Estrutura do arquivo *CSV* gerado. 48
Tabela 2 – Estrutura de dados após Engenharia de Características. 49
Tabela 3 – Matriz de correlação. 49
Tabela 4 – Resumo do ERP dos modelos contruídos para as taxas da CWE476 . . 67
Tabela 5 – EMQ dos modelos das taxas da CWE476 através da validação cruzada. 68
Tabela 6 – Resumo do ERP dos modelos contruídos para as taxas da CWE457 . . 69
Tabela 7 – EMQ dos modelos das taxas da CWE457 através da validação cruzada. 71
Tabela 8 – Alguns valores da taxa da CWE476 mais predição realizada. 72
Tabela 9 – Alguns valores da taxa da CWE457 mais predição realizada. 72
Tabela 10 – Predição de métricas em projetos de software livre maiores. 73
Tabela 11 – Projeto Bash 99
Tabela 12 – Projeto Blender 99
Tabela 13 – Projeto FFmpeg 100
Tabela 14 – Projeto Firefox 100
Tabela 15 – Projeto Gstreamer 100
Tabela 16 – Projeto Inetutils 101
Tabela 17 – Projeto Linux Kernel 101
Tabela 18 – Projeto OpenSSH 101
Tabela 19 – Projeto OpenSSL 101
Tabela 20 – Projeto Python2.7 102
Tabela 21 – Projeto Ruby-2.1 102

Lista de Códigos

2.1 Código exemplo CWE476 27
2.2 Evitando ameaça de vulnerabilidade CWE476 27
2.3 Código exemplo CWE457 28
2.4 Evitando ameaça de vulnerabilidade CWE457 29
2.5 Código exemplo CWE401 29
3.1 Análise léxica simples com a ferramenta *Flex* 33
3.2 Definição de uma *CFG* simples com a ferramenta *GNU Bison* 33

Lista de abreviaturas e siglas

AN Argument with 'nonnull' attribute passed null - Argumento com atributo não nulo passado nulo

ASOM Allocator sizeof operand mismatch - Operador incompatível com alocador sizeof

AST Abstract Syntaxe Tree - Árvore de sintaxe abstrata

AUV Assigned value is garbage or undefined - Valor atribuído é lixo ou indefinido

BD Bad deallocator - Desalocador de memória ruim

BF Bad free - Liberação ruim de memória

CSV Comma Separeted Values - Valores separados por vírgula

CWE Common Weakness Enumeration - Enumeração de Vulnerabilidades Comuns

DBZ Divisions by zero - Divisão por zero

DF Double free - Liberação dupla de memória

DNP Dereference of null pointer - Desreferência de ponteiro nulo

DUPV Dereference of undefined pointer value - Desreferência de valor de ponteiro indefinido

ERP Erro residual padrão

EMQ Erro médio quadrático

FGBO Potential buffer overflow in call to 'gets' - Potencial buffer overflow na chamada de 'gets'

MLK Memory leak - Vazamento de memória

NIST National Institute of Standards and Technology - Instituto Nacional de Padrões e Tecnologia

OBAA Out-of-bound array access - Acesso de array fora do limite

OSF Offset free - Liberação de memória com deslocamento

PITFC Potential insecure temporary file in call ’mktemp’ - Potencial insegurança de arquivo temporário na chamada ’mktemp’

ROGU Result of operation is garbage or undefined - Resultado da operação é lixo ou indefinido

RSVA Return of stack variable address - Retorno de endereço de uma variável da pilha

SAIGV Stack address stored into global variable - Endereço da pilha armazendo em variável global

UA Undefined allocation of 0 bytes - Alocação indefinida de 0 bytes

UAF Use-after-free - Uso de memória após liberação

UAV Uninitialized argument value - Valor do argumento não inicializado

Sumário

1 INTRODUÇÃO . . . 19
1.1 Justificativa . . . 19
1.2 Objetivos . . . 20
1.3 Organização do Trabalho . . . 21

2 ANÁLISE ESTÁTICA DE CÓDIGO-FONTE . . . 23
2.1 Classificação e Taxonomia das Vulnerabilidade de Software . . . 24
2.2 Métricas de Ameaças de Vulnerabilidade de Código-fonte . . . 26
2.2.1 CWE476 - Referência a Ponteiros Nulos . . . 26
2.2.2 CWE457 - Variáveis não Inicializadas . . . 28
2.2.3 CWE401 - Vazamento de Memória . . . 29

3 FERRAMENTAS DE ANÁLISE ESTÁTICA DE CÓDIGO-FONTE . 31
3.1 Ferramentas de Análise Estática de Segurança de Código-fonte . . 31
3.1.1 Construção do Modelo . . . 32
3.1.2 Realização da Análise . . . 35
3.1.3 Apresentação dos Resultados . . . 35
3.1.4 Classificação das Ferramentas . . . 36

4 ANÁLISE EXPLORATÓRIA DE DADOS . . . 37
4.1 Análise dos dados . . . 38
4.2 Técnica de Visualização dos Dados . . . 38
4.3 Identificação de *Outliers* . . . 39
4.4 Definição de Modelos . . . 40
4.5 Validação de Modelos . . . 41

5 METODOLOGIA . . . 43
5.1 Trabalhos Relacionados . . . 44
5.2 Projeto da Pesquisa . . . 44
5.3 Teste das Hipóteses . . . 46
5.3.1 Descrição dos dados e Engenharia de Características . . . 48
5.3.2 Análise Exploratória dos Dados . . . 49
5.3.2.1 CWE476 - Referência de Ponteiros Nulos . . . 51
5.3.2.2 CWE457 - Variável não Inicializada . . . 54
5.3.2.3 Resultados Obtidos . . . 56

6 DEFINIÇÃO DOS MODELOS DE PREDIÇÃO . . . 59

6.1 Definição de modelo para CWE476 - Referência a Ponteiros Nulos . 60
6.2 Definição de modelo para CWE457 - Variável não Inicializada . . . 62
6.3 Comparação e Escolha dos Modelos 65
6.3.1 CWE476 - Referência de Ponteiros Nulos 66
6.3.2 CWE457 - Variável não Inicializada 69
6.4 Consolidação dos Resultados 71
6.5 Exemplos de Uso 73
6.6 Evolução dos Modelos de Predição 74

7 CONCLUSÃO 77
7.1 Limitações do Trabalho 78
7.2 Trabalhos Futuros 78

Referências 81

APÊNDICES 85

APÊNDICE A – ANÁLISE DOS PERCENTIS 87

APÊNDICE B – ANÁLISE QUALITATIVA 99

1 Introdução

A medição é um processo auxiliar essencial para o desenvolvimento de software com qualidade. É importante medir para entender e controlar processos, produtos e projetos (ROCHA; SOUZA; BARCELLOS, 2012). Medidas fornecem informações sobre objetos (processos, produtos e projetos) e eventos, tornando-os compreensíveis e controláveis (FENTON; PFLEENGER, 1998). Entretanto, muitas vezes, mede-se simplesmente por medir (ROCHA; SOUZA; BARCELLOS, 2012), não justificando todo o esforço gasto durante o processo.

O cenário apresentado acima é o que geralmente ocorre quando as pessoas não estão acostumadas a trabalhar no dado contexto. Quando as pessoas não sabem trabalhar com determinadas métricas de software existem duas opções: utilizam de maneira inadequada, não agregando valor ao processo de desenvolvimento de software; ou apenas não utilizam, mesmo que sejam importantes para o processo de desenvolvimento.

Muitas vezes, as métricas de código-fonte se encaixam bem nessa situação, não são utilizadas dentro do processo de desenvolvimento porque não se sabe o que deve ser feito com as mesmas e que decisões tomar. Quando utilizadas, são utilizadas apenas as métricas de *design* de código-fonte mais conhecidas, como as métricas orientadas a objetos de Chidamber e Kemerer (2002), isso ocorre porque são métricas já consolidadas e possuem várias diretrizes de como utilizá-las de maneira adequada.

Neste trabalho, nós argumentamos que outras classes de métricas de código-fonte, como as métricas de ameaças de vulnerabilidade, por exemplo, não são tão utilizadas no desenvolvimento de software, devido a falta de conhecimento de como interpretá-las e monitorá-las. Nesse contexto ao investigar como monitorar métricas de ameaças de vulnerabilidade de código-fonte, de modo que ela possa ser utilizada e interpretada de maneira que agregue valor ao produto/negócio, estamos propondo modelos preditivos para essa classe de métrica.

1.1 Justificativa

Sabendo que existe o problema dentro da Engenharia de Software de interpretação e monitoramento de métricas de código-fonte além das mais conhecidas, que são as métricas de *design*, espera-se conseguir auxiliar os Engenheiros de Software com este trabalho. Tentando apresentar uma forma de acompanhar e monitorar essas diferentes métricas que estão emergindo nos últimos tempos. Muitos não as utilizam devido a falta de conhecimento sobre o que fazer com essas informações advindas dessas classes de métricas.

Relacionado às métricas de ameaças de vulnerabilidade de código-fonte, existe a questão da segurança dos softwares, que está em voga nos últimos tempos, o que vem aumentando cada vez mais a busca por requisitos não funcionais relacionados a segurança do software desenvolvido. Isso reforça a necessidade de inserção de métricas de ameaças de vulnerabilidade de código-fonte dentro do ciclo de desenvolvimento de software, a fim de quantificar alguns aspectos do software até então não mensurados. Como foi mencionado anteriormente, não se consegue controlar um aspecto do desenvolvimento do software enquanto não se pode medi-lo. A identificação dessas ameaças de vulnerabilidades ainda dentro do ciclo de desenvolvimento facilita a correção das mesmas, além de evitar custos de possíveis futuras manutenções corretivas. Mas para isso, é necessário saber como monitorá-las e acompanhá-las no decorrer do ciclo de desenvolvimento.

Existem algumas iniciativas de pesquisas relacionadas a como associar características de *design* do código-fonte com vulnerabilidades, como nos trabalhos de Esposte e Bezerra (2014) e Alshammari, Fidge e Corney (2009). Entretanto, isso não se mostra suficiente para a inserção dessas métricas no ciclo de desenvolvimento de software. Além disso, é necessário encontrar uma forma mais objetiva para auxiliar Engenheiros de Software no monitoramento das métricas de ameaças de vulnerabilidade de código fonte, sendo esse um dos possíveis motivos para a sua não utilização.

Segundo um artigo divulgado pelo *NIST* (*National Institute of Standards and Technology*), escrito por Jansen (2009), a coleta de dados de projetos existentes e a análise dos mesmos podem indicar padrões e informações importantes para a medição da segurança de software, sendo essa apontada como uma possível área de pesquisa. Isso corrobora com a ideia deste trabalho, onde foram definidos modelos preditivos de referência para algumas métricas de ameaças de vulnerabilidade de código-fonte baseado em um software de referência, analisando temporalmente as suas versões.

1.2 Objetivos

Levando em consideração o contexto apresentado anteriormente, o objetivo deste trabalho é encontrar um modelo preditivo para métricas de ameaças de vulnerabilidade de código-fonte que sirva de referência para outros projetos de software.

Para isso, outros objetivos paralelos precisam ser atingidos, como o entendimento do processo de análise estática, principalmente o que diz respeito a segurança de código-fonte, como as ferramentas que realizam essa análise funcionam, além de algum processo de análise estatística que auxilie no desenvolvimento da pesquisa.

Em resumo, o principal objetivo de pesquisa deste trabalho é responder a seguinte questão-problema:

É possível o desenvolvimento de modelos preditivos de referência, para acompanhamento e monitoramente de métricas de ameaças de vulnerabilidade de código-fonte em projetos de software?

Para responder essa questão-problema foram levantadas as seguintes hipóteses a serem respondidas:

- *H1*: As métricas de ameaças de vulnerabilidade de código-fonte podem ser observadas de maneira similar às métricas de *design* de código fonte.
- *H2*: Os valores das métricas de ameaças de vulnerabilidade de código-fonte se comportam como distribuições estatísticas de cauda longa, e não distribuições estatísticas normalizáveis, assim como acontece com as métricas de *design*.
- *H3*: A definição de um modelo baseado em uma função polinomial simples possibilita o monitoramento e acompanhamento das métricas de ameaças de vulnerabilidade de código-fonte.

1.3 Organização do Trabalho

Este trabalho está dividido em seis capítulos subsequentes. No capítulo 2, são apresentados alguns conceitos como o de análise estática e métricas de software, além de apresentar algumas peculiaridades sobre a classificação e taxonomia de vulnerabilidades de software para em seguida serem apresentadas as métricas de ameaças de vulnerabilidade de código-fonte utilizadas neste trabalho. No capítulo 3, é apresentada a importância de uma ferramenta que automatize o processo de análise estática, como essas ferramentas funcionam em geral e em seguida é aprofundado sobre cada uma das etapas realizadas pelas ferramentas de análise estática de segurança de código-fonte. No capítulo 4, é apresentado a diferença da análise exploratória de dados para as abordagens estatísticas tradicionais, em seguida é detalhado cada uma das etapas realizadas durante esse processo. No capítulo 5, contém toda a metodologia desenvolvida nesta pesquisa, seguindo uma abordagem de análise exploratória de dados, para poder se definir modelos preditivos para as métricas em questão. No capítulo 6, são apresentados as definições e validações dos modelos preditivos das métricas de ameaças de vulnerabilidade de código-fonte, continuando a abordagem estatística apresentada anteriormente, chegando ao final com fórmulas matemáticas que representem-os. No capítulo 7, sendo esse o último capítulo do trabalho, serão feitas algumas conclusões, respondendo a questão de pesquisa levantada, além de apresentar algumas limitações e riscos que podem ameaçar este trabalho e uma lista de trabalhos futuros.

2 Análise Estática de Código-fonte

A análise estática examina o código-fonte do programa e as razões sobre todos os comportamentos possíveis que possam surgir em tempo de execução (ERNST, 2005). Ou seja, é uma análise realizada sem a necessidade de execução do código-fonte desenvolvido, não necessitando de suas dependências e outras peculiaridades para realizá-la. Além disso, análise estática pode nos apresentar características inerentes ao código-fonte, podendo as mesmas serem relacionadas ao *design* do código, apresentando-nos informações como complexidade dos métodos e tamanho por exemplo.

Em geral, ao final de um processo de análise estática se tem alguma métrica de software, sendo métrica de software uma função cujas entradas são os dados de software e cuja saída é um único valor numérico, que pode ser interpretada como o grau em que o software possui um determinado atributo que afeta sua qualidade (IEEE, 1998). Entretanto, também existem análises estáticas que tem como objetivo a identificação de não conformidades com o estilo do código por exemplo, que podem não ter uma métrica de software como saída do processo.

Usualmente, análise estática de código-fonte remete à uma análise do *design* do código, da complexidade ou tamanho do mesmo, entretanto, essa é uma visão que vem mudando devido aos requisitos não funcionais relacionados à segurança do software ganharem cada vez mais importância em virtude do contexto atual. Onde cada vez mais são desenvolvidas pesquisas acerca da análise estática voltada para a segurança do código, tentando tornar esse tipo de requisito quantificável e facilitar a inserção dessa classe de métricas no ciclo de desenvolvimento de software.

Vários estudos vêm sendo realizados acerca da análise de código-fonte e definição de métricas, além da sua relação com ameaças ou falhas em programas. Sendo que um conjunto de ameças de vulnerabilidade pode ou não se tornar uma vulnerabilidade dependendo do estado do sistema e abrir brechas para atacantes. Como por exemplo em Ferzund, Ahsan e Wotawa (2009), Misra e Bhavsar (2003), Nagappan, Ball e Zeller (2006), Esposte e Bezerra (2014). Isso demostra que cada vez mais existe a preocupação em volta da segurança e ameaças de vulnerabilidade de software, e pensando em desenvolver e manter software de uma maneira mais segura, necessita-se um melhor entendimento das ameaças de vulnerabilidade e as vulnerabilidades conhecidas. Lembrando que a análise estática de código-fonte é um forte aliado para garantir a segurança de software, permitindo que equipes de desenvolvimento encontrem possíveis vulnerabilidades durante o ciclo de desenvolvimento, reduzindo o custo de futuras manutenções corretivas.

Serão apresentadas na Seção 2.1 a taxonomia e classificação das vulnerabilidades

já conhecidas, e na Seção 2.2 serão explicadas algumas das métricas de ameaças de vulnerabilidade mais comuns em projetos de software, sendo apresentadas as selecionadas para a realização deste trabalho.

2.1 Classificação e Taxonomia das Vulnerabilidade de Software

Segundo Seacord e Householder (2005), compreender as vulnerabilidades é crucial para entender as ameaças que elas representam. Visando isso, alguns trabalhos foram desenvolvidos acerca da classsificação das vulnerabilidades existentes, tentou-se definir uma taxonomia para as mesmas. Taxonomia é o processo científico de categorizar entidades, ou seja, organizá-las em grupos (GRéGIO et al., 2005). Dividindo as vulnerabilidades em grupos passa a ser mais fácil o entendimento de cada um delas e seus conteúdos. As características dos grupos, chamadas características taxonômicas ou atributos, devem satisfazer às seguinte propriedades segundo Krsul (1998):

- **Objetividade**: A característica deve ser identificada a partir do conhecimento objetivo e não do conhecimento subjetivo. O atributo mensurado deve ser claramente observado.
- **Determinismo**: Deve haver um processo claro que pode ser seguido para se extrair a característica.
- **Repetibilidade**: Várias pessoas, independentemente, extraindo a mesma característica do objeto devem concordar com o valor observado.
- **Mutuamente exclusiva**: A categorização em um grupo exclui a categorização em qualquer outro grupo.
- **Exaustiva**: Juntos, os grupos incluem todas as possibilidades.
- **Aceitável**: Lógica e intuitiva, de forma que as categorias possam ser aceitas pela comunidade.
- **Útil**: Pode ser utilizada para a obtenção de conhecimento no campo de pesquisa.

Portanto, se as propriedades apresentadas não forem satisfeitas não existe taxonomia, sendo apenas uma simples classificação de vulnerabilidades. Vários trabalhos foram desenvolvido com o intuito de desenvolver uma taxonomia para as vulnerabilidades de software, como em Krsul (1998), Aslam, Krsul e Spafford (1996), Landwehr et al. (1994). A primeira proposta de taxonomia para vulnerabilidades de software foi proposta em Abbott et al. (1976), esse estudo ficou conhecido como *RISOS* (*Research Into Secure*

Operating Systems), que tinha como intuito auxiliar administradores de sistemas a entender aspectos de segurança dos sistemas operacionais para prover uma melhor segurança para os mesmos. Nesse caso, as vulnerabilidades foram agrupadas em sete classes, sendo elas:

1. Validação incompleta dos parâmetros
2. Validação inconsistente dos parâmetros
3. Compartilhamento implícito de privilégios ou dados confidenciais
4. Validação assíncrona ou serialização inadequada
5. Autorização ou autenticação ou identificação inadequadas
6. Violação de proibição ou limite
7. Erro explorável de lógica

Vários outros estudos foram derivados a partir do *RISOS*, entretanto, o que acontece é que essas taxonomias estão em desuso e a tendência atual é a adoção de esquemas de classificação de vulnerabilidades, ou seja, o que acontece na prática não repeita as propriedades de uma taxonomia.

Apesar da classificação ser o método mais utilizado nos dias de hoje, ainda não existe um consenso de uma forma correta de se classificar vulnerabilidades. Uma das mais conhecidas é a CVE (*Common Vulnerabilities and Exposures*), que é mantida pelo *MITRE*[1], sendo essa uma lista de nomes padronizados para vulnerabilidades e fraquezas de sistemas e softwares, que tem por objetivo padronizar as vulnerabilidades e fraquezas já conhecidas, facilitando o compartilhamento de informações encontradas de ano a ano (GRéGIO et al., 2005). Esse é um trabalho de extrema importância vide que algumas organizações nomeavam a mesma vulnerabilidade com nomes diferentes no passado, dificultando o entendimento das mesmas.

Mesmo com o uso das CVEs, viu-se a necessidade das empresas e organizações de utilizarem uma terminologia padrão para listar e classificar vulnerabilidades de software, gerando uma base de dados unificada e uma base para ferramentas e serviços de medição dessas vulnerabilidades, sendo assim foram desenvolvidas as CWEs (ESPOSTE; BEZERRA, 2014). A CWE[2] (*Common Weakness Enumeration*) é uma lista formal de vulnerabilidades comuns de software, tendo como objetivo estabelecer uma linguagem comum para descrever ameaças de vulnerabilidade de software no *design*, arquitetura ou no código (ESPOSTE; BEZERRA, 2014). A principal diferença entre as CVEs e CWEs é que as

[1] <http://www.mitre.org/>
[2] <https://cwe.mitre.org/>

CVEs são vulnerabilidades que já ocorreram e são conhecidas pela comunidade, já as CWEs estão relacionadas a fraquezas de software, apresentando códigos e práticas de programação que podem se tornar futuras vulnerabilidades do software.

As métricas apresentadas na Seção 2.2 foram definidas baseadas nas CWEs mais recorrentes em projetos de software livre, sendo esse um indicativo que devem ser monitoradas em um projeto se software, conforme discutido no Apêndice B, referente a primeira etapa deste trabalho.

2.2 Métricas de Ameaças de Vulnerabilidade de Código-fonte

As métricas de ameaças de vulneabilidade de código-fonte apresentadas nesta seção foram baseadas no estudo realizado na primeira parte desta pesquisa, detalhes sobre essas decisões são discutidos no Capítulo 5.

As principais ameaças de vulnerabilidade encontradas em projetos de software livre foram: referência a ponteiros nulos, variáveis não inicializadas e vazamento de memória. Sendo a CWE476[3] referente a referência a ponteiros nulos, a CWE457[4] referente a variáveis não inicializadas e a CWE401[5] referente ao vazamento de memória. A definição das métricas relacionadas a cada uma das ameaças de vulneabilidade de código-fonte foi bem simples, sendo obtida a partir do somatório da quantidade de ameaças encontradas no projeto dividido pelo número total de módulos, dando uma taxa de cada uma das ameaças de vulnerabilidade por módulo, podendo o valor da métrica variar entre 0.0 e 1.0.

Aprofundando mais em cada uma das ameaças de vulnerabilidade a fim de compreendê-las, serão apresentadas uma a uma a seguir, nas Seções 2.2.1, 2.2.2 e 2.2.3.

2.2.1 CWE476 - Referência a Ponteiros Nulos

A ameaça de vulnerabilidade de referência a ponteiros nulos acontece quando a aplicação tenta acessar o conteúdo da região de memória a que o ponteiro deveria estar apontando, entretanto, o ponteiro está nulo, devido a algum estado inesperado do programa. Tipicamente, esse tipo de ocorrência aborta a execução do programa. Esse tipo de ameaça de vulnerabilidade pode ocorrer desde um pequeno erro de programação, esquecendo de validar algum ponteiro antes de acessá-lo, até complexas condições de corrida, onde existam dois processos paralelos que acessam o mesmo ponteiro, e um libere a região de memória antes do outro finalizar a utilização da mesma em um estado do programa específico.

[3] <https://cwe.mitre.org/data/definitions/476.html>
[4] <https://cwe.mitre.org/data/definitions/457.html>
[5] <https://cwe.mitre.org/data/definitions/401.html>

No Código 2.1 pode-se ver um exemplo simples onde é chamada uma função que deveria retornar um ponteiro para uma estrutura de dados e um campo dessa estrutura é passado para uma chamada de sistema, entretanto, pode haver um caso específico onde a função chamada não retorne o ponteiro esperado, e nesse caso, faça referência a um ponteiro nulo.

Lista de Códigos 2.1 Código exemplo CWE476

```
1 typedef struct command_ {
2    char* program;
3    char* input;
4 } command;
5
6 void main() {
7     command *p = get_command();
8     system(p->program, p->input);
9 }
```

O exemplo do Código 2.1 apresenta uma ameaça de vulnerabildade, pois o conteúdo para onde o ponteiro aponta pode aparentemente sempre vir certo, entretanto, em algum momento, com algumas entradas específicas o ponteiro retornado pode ser nulo, caracterizando uma ameaça de vulnerabilidade de código fonte. A eliminação da ameaça de vulnerabilidade é bastante simples nesse caso, apenas é necesária a validação do ponteiro antes de sua chamada, bastando fazer algo similar ao que é apresentado no Código 2.2.

Lista de Códigos 2.2 Evitando ameaça de vulnerabilidade CWE476

```
1 typedef struct command_ {
2    char* program;
3    char* input;
4 } command;
5
6 void main() {
7     command *p = get_command();
8
9     if(p != NULL) {
10        system(p->program, p->input);
11    }
12 }
```

Lembrando que essa ameaça de vulnerabilidade só pode ocorrer em linguagens de programação que permitem a manipulação de ponteiros, dando liberdade para gerenciar a memória utilizada pelo programa, sendo possível em linguagens tais como *C* e *C++*.

Nesse caso específico apresentado ainda abre brechas para outras ameaças de vulnerabilidade, onde um atacante poderia executar algum comando no sistema com o usuário que está executando o processo do programa, já que o mesmo faz uma chamada de sistema sem validar os seus parâmetros.

2.2.2 CWE457 - Variáveis não Inicializadas

A ameaça de vulnerabilidade de código-fonte relacionada a variáveis não inicializadas ocorre quando o programa faz uso de uma variável que ainda não foi inicializada pelo mesmo, tendo que lidar com certas coisas que pode levar o pragrama para um estado imprevisível. Em algumas linguagens como *C*, as variáveis quando são declaradas não são inicializadas, e quando isso aconte o conteúdo daquela variável passa a ser algo que estava anteriormente naquela região de memória alocada, nesse caso, um atacante pode de alguma forma conseguir escrever algo nessa região de memória inicializando a variável com algum valor malicioso. Em outras linguagens onde as variáveis quando declaradas são inicializadas com algum valor *default*, dependendo da implementação do programa, pode indicar um erro de tipo de variável em tempo de execução, abortando o programa.

No Código 2.3 é exemplificado essa ameaça de vulnerabilidade de código-fonte, onde se inicializa a variável apenas se o retorno de uma função é verdadeiro. Nessa situação, quando o retorno for falso a variável não será inicializada, entretanto, ainda será acessada, ocasionando um erro no programa.

Lista de Códigos 2.3 Código exemplo CWE457

```
1 void main() {
2     int x;
3
4     if(isTrue()) {
5         x = 10;
6     }
7
8     manipulate(x);
9 }
```

No melhor caso, se a variável não for inicializada o programa irá abortar, entretanto, um atacante pode ter escrito algum código ou dado malicioso na região de memória onde se localiza a pilha de variáveis do programa, podendo ter inicializado a variável com

algo malicioso, e essa variável será chamada levando o programa a um estado imprevisível. Uma possível solução para esse problema pode ser inicializar a variável com algum valor *default* que a variável não possa assumir, e ao chamar o método que manipula a variável verificar se o conteúdo da variável ainda é o valor *default*, como pode-se ver no Código 2.4.

Lista de Códigos 2.4 Evitando ameaça de vulnerabilidade CWE457

```
1 void main() {
2     int x = -1;
3
4     if(isTrue()) {
5         x = 10;
6     }
7
8     if(x != -1) {
9         manipulate(x);
10    }
11 }
```

2.2.3 CWE401 - Vazamento de Memória

A ameaça de vulnerabilidade de vazamento de memória ocorre quando o programa não consegue gerenciar a liberação de memória após a sua utilização de maneira aceitável, consumindo toda a memória alocada para o *HEAP* do programa aos poucos, até não restar mais. Esse tipo de erro acaba encerrando o programa em algum momento inesperado, o que pode acabar gerando algum tipo de perda de dados por exemplo.

No Código 2.5 é exemplificado a ameaça de vulnerabilidade de código-fonte, onde uma função aloca memória para as variáveis e quando a função de liberação de memória é chamada, a mesma esquece de desalocar a região de memória de uma das variáveis. Isso não seria um problema muito grande se não houvesse várias requisições a essas funções, onde em algum ponto não restará mais memória para o programa alocar novas variável e o mesmo será encerrado.

Lista de Códigos 2.5 Código exemplo CWE401

```
1 foo connect() {
2     foo foo = malloc(2048);
3     foo bar = malloc(1024);
4
```

```
5     return foo;
6 }
7 
8 void destroy(foo) {
9     free(foo);
10 }
11 
12 void main()
13 {
14     while(get_request_for_connection()) {
15         foo var = connect();
16         destroy(var);
17     }
18 }
```

Esse pequeno erro de programação apresentado pode levar o programa a ser encerrado no meio da sua execução, nesse caso a solução é bastante simple, basta remover o segundo *malloc()* da função *connect()*. Porém, podem haver casos mais críticos de vazamento de memória, onde possa haver algum tipo de vulnerabilidade no programa que permita um atacante puxar algum gatilho para que haja vazamento de memória, dessa forma podendo caracterizar um ataque de negação de serviço (*Denial of Service Attack*).

Esse tipo de ameaça de vulnerabilidade assim como a apresentada na Seção 2.2.1, só é possível em programas escritos em linguagens de programação que dê liberdade para o seu próprio gerenciamento de memória, como as linguagens *C* e *C++*.

Com base nas métricas de ameaças de vulnerabilidade de código-fonte aqui apresentadas, foi realizada uma análise ao decorrer do tempo das versões do projeto *Linux Kernel*, a fim de definir modelos de predição de referência para essas métricas que auxiliem a monitoramento das mesmas no ciclo de desenvolvimento de software.

3 Ferramentas de Análise Estática de Código-fonte

A realização da análise estática de código-fonte de maneira manual, ou seja, sem o auxílio de uma ferramenta que automatize esse processo, é bastante custosa e impraticável, já que seria necessário um especialista passar em cada arquivo fonte realizando as análises necessárias. Sendo assim, seria praticamente impossível a inserção dessa prática no ciclo de desenvolvimento de qualquer software, tendo em vista que o custo seria gigantesco, salvo alguns casos específicos onde esse tipo de atividade se faz necessária. Com isso em mente, as ferramentas de análise estática de código-fonte são essenciais para que se torne viável a utilização da mesma no desenvolvimento de software.

Visando automatizar o processo de análise estática, as ferramentas varrem arquivos e diretórios em busca do código-fonte, analisam-os e apresentam métricas para o usuário, de maneira quantitativa ou qualitativa. Muitas dessas ferramentas são bem complexas e para realizar esse processo de análise do código-fonte realizam várias etapas intermediárias. Entretanto, existem ferramentas que visam analisar o código-fonte de maneira mais simples, menos complexa, onde será menos custoso computacionalmente, para que possa dar uma resposta mais rápida para o usuário.

As ferramentas de análise estática de código-fonte, em geral, se assemelham bastante com a estrutura de um compilador, onde se faz uma análise do código fonte, se controi um modelo/estrutura desse código, realiza uma análise sobre essa estrutura e apresenta os resultados para o usuário. Segundo Chess e West (2007), os problemas enfrentados por grande parte das ferramentas de análise estática de código-fonte se assemelham aos enfrentados na contrução de compiladores e na otimização dos mesmos. Tendo em vista que este trabalho tem um foco em segurança de código-fonte, na Seção 3.1 é apresentada a estrutura da maioria das ferramentas que realizam análise estática voltada para a segurança do código fonte.

3.1 Ferramentas de Análise Estática de Segurança de Código-fonte

Para se entender o funcionamento de uma ferramenta de análise estática de segurança de código-fonte genérica é apresentado um diagrama de blocos na Figura 1, onde Chess e West (2007) resumem essa sistemática.

Como apresentado na Figura 1, essas ferramentas possuem um código-fonte como entrada, constroem um modelo baseado nesse código-fonte, realizam uma análise nesse

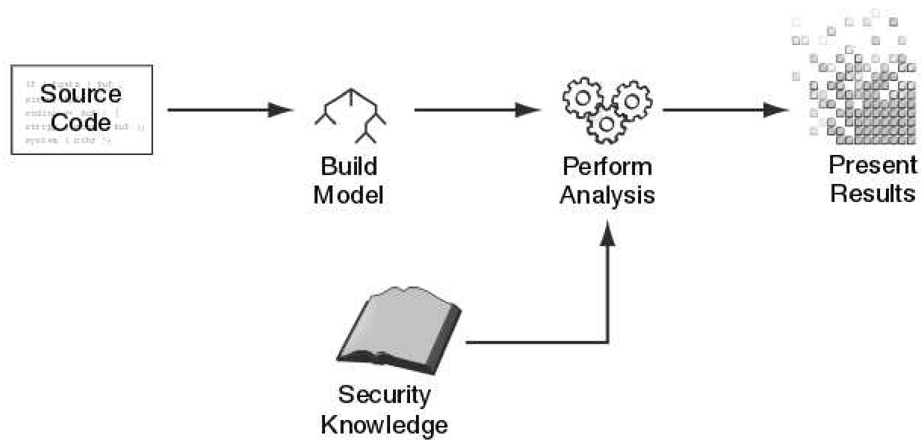


Figura 1 – Diagrama de blocos de uma ferramenta genérica (CHESS; WEST, 2007)

modelo construído baseado em um conhecimento prévio em segurança de código e finalmente apresentam os resultados para o usuário. Esse fluxo é similar ao que foi apresentado no início deste capítulo, logo, o funcionamento das ferramentas de análise de código-fonte em geral seguem a mesma estrutura. Simplesmente alterando a forma que é realizada a análise em si, onde vai variar dependendo do objetivo da análise estática.

Nas seções a seguir serão apresentados em mais detalhes cada um dos blocos representados na Figura 1.

3.1.1 Construção do Modelo

A fase de construção de um modelos baseado no código-fonte de entrada é praticamente idêntica ao que é realizado por um compilador. São elaboradas as seguintes atividades sequencialmente:

- Análise léxica
- *Parsing* ou Análise sintática
- Análise semântica

Na análise léxica, todo o código-fonte é transformado em uma série de *tokens*, descartando partes não importantes do código, como espaços em branco e comentários (CHESS; WEST, 2007). Existem algumas ferramentas, como o *Flex*[1], que auxiliam nessa tranformação do código-fonte em *tokens*, sendo essa bastante utilizada na construção de compiladores. Para a execução dessa atividade utiliza-se bastante de técnicas como expressões regulares para a identificação dos *tokens* desejados. O Código 3.1 exemplifica como pode ser feita uma análise léxica bastante simples com a ferramenta *Flex*.

[1] <http://flex.sourceforge.net/>

Lista de Códigos 3.1 Análise léxica simples com a ferramenta *Flex*

```
1  [ \t\n]+            { //ignorando espaços em branco }
2  \/\/.*              { //ignorando comentários }
3  if                  { return IF; }
4  for                 { return FOR; }
5  (                   { return LEFT_PARETHESIS; }
6  )                   { return RIGHT_PARENTHESIS; }
```

Na fase de *parsing* ou análise sintática, o *stream* de *tokens* recebidos da análise léxica é manipulado tentando enquadrá-lo a gramática de contexto livre (*context free-grammar* ou simplesmente *CFG*) que é definida. A gramática de contexto livre nada mais é do que um conjunto de *productions* (regras) que descrevem os símbolos daquela linguagem (CHESS; WEST, 2007). A ferramenta *GNU Bison*[2] é bastante utilizada para a definição dessa gramática de contexto livre, sendo essa facilmente integrada com o *Flex*. O Código 3.2 exemplifica a definição de uma gramática de contexto livre simples.

Lista de Códigos 3.2 Definição de uma *CFG* simples com a ferramenta *GNU Bison*

```
1  entrada
2          : /* vazio */
3          | entrada linha
4          ;
5  linha
6          : '\n'
7          | expressao '\n'
8          ;
9  expressao
10         : IF LEFT_PARENTHESIS expressao RIGHT_PARENTHESIS bloco
11         ;
```

Com esse tipo de definição de gramática de contexto livre e o *stream* de *tokens* pode-se chegar a uma *Parse Tree*, onde facilita a visualização da estrutura do código, como pode ser visto na Figura 2 apresentada por Chess e West (2007).

Muitas ferramentas de análise estática de segurança de código-fonte realizam as suas análises diretamente na sua *Parse Tree* já que essa está bem próximo ao código que foi escrito, mas para realizar análises mais complexas pode ser inconveniente, sendo necessário o desenvolvimento de uma *Abstract Syntax Tree* (*AST*) (CHESS; WEST, 2007).

[2] <https://www.gnu.org/software/bison/>

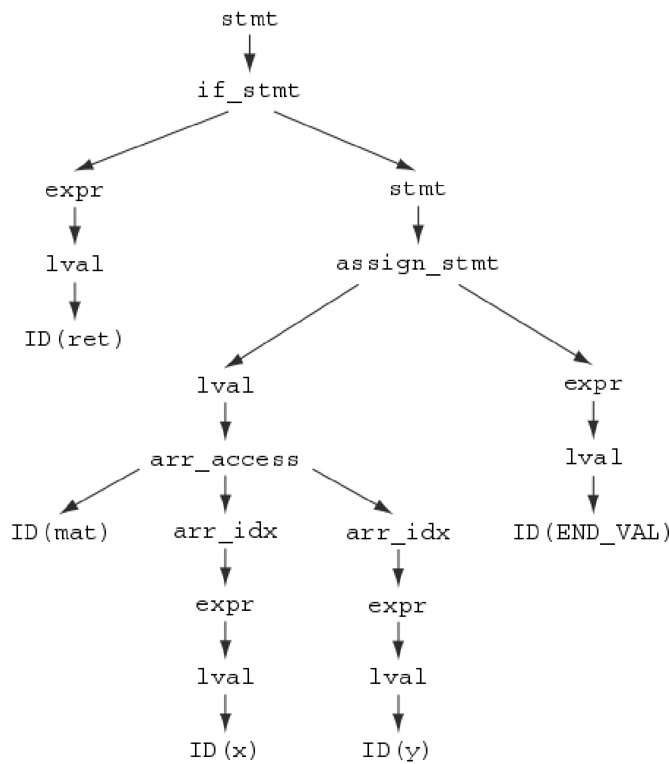


Figura 2 – Exemplo de *Parse Tree* (CHESS; WEST, 2007)

Na *AST* vários detelhes relacionado a gramática são abstraídos a fim de facilitar a análise da mesma, a Figura 3 nos mostra um exemplo dessa estrutura de dados.

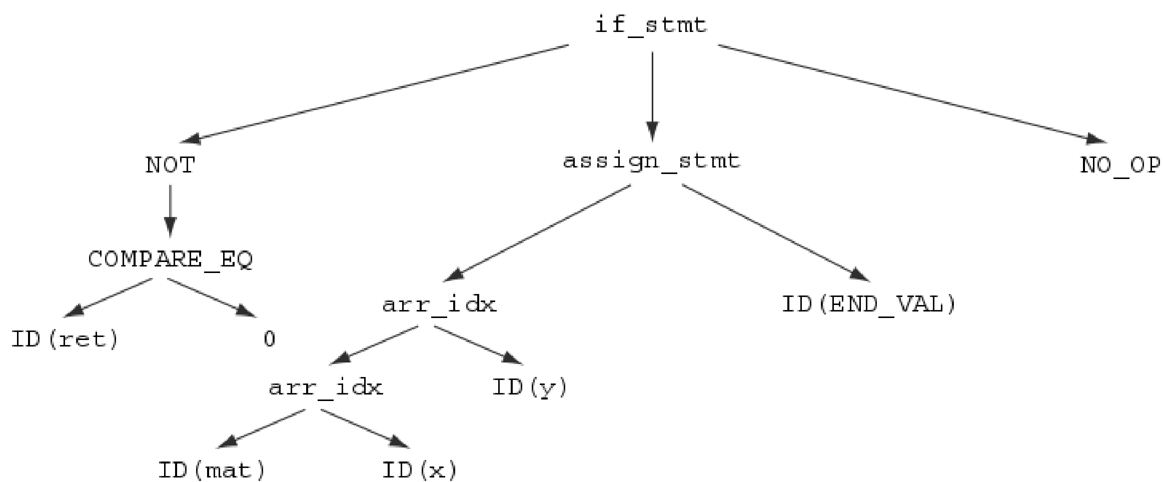


Figura 3 – Exemplo de *AST* (CHESS; WEST, 2007)

Em paralelo a construção dessas estruturas de dados a ferramenta de análise estática constroi também a chamada tabela de símbolos, onde nela são associados os identificadores, os seus respectivos tipos e um ponteiro para onde os mesmos foram declarados ou definidos (CHESS; WEST, 2007). Com isso feito, as ferramentas realizam uma análise semântica do código, como por exemplo *type checking*. É após a análise semântica que as

ferramentas de análise estática e os compiladores começam a se diferenciar.

Para algumas ferramentas de análise estática são necessárias algumas formas de representação intermediária do código, como gráficos de fluxo do programa, fluxo dos dados do programa e chamadas de métodos ou funções. Dependo do que a ferramenta se propõe a fazer, a elaboração desse tipo de representação é essencial.

3.1.2 Realização da Análise

Após a contrução do modelo apresentada anteriormente, inicia-se a fase de análise desse modelo, que como foi dito no início desta seção depende de conhecimentos relacionados à segurança de código-fonte. Devido a essa análise ser diferente dependendo do objetivo da ferramenta, e sendo o intuito deste capítulo apenas uma visão geral do funcionamento das mesmas, não foi aprofundado esse assunto, apenas serão apresentados os diferentes tipos de análise que podem ser realizadas.

Existem basicamente dois tipos de análise que podem ser realizadas, sendo elas análise intraprocedural (local) ou interprocedural (global). Como o próprio nome já diz, a análise intraprocedural ou local é a análise realizada apenas no contexto daquele procedimento ou função, ou seja, não será levado em considerção nada que não esteja presente naquele contexto, por exemplo, chamadas a outras funções dentro da função analisada serão desconsideradas. A análise interprocedural ou global leva em consideração a interação entre as funções, sendo essa mais completa, entretanto, bem mais complexa. Algumas ferramentas utilizam a análise local para a realização da análise global, onde a análise local gera um sumário de todas as funções e durante a análise global esse sumário é consultado, facilitando o processo.

Segundo Chess e West (2007), a flexibilidade das ferramentas para a definição de regras para a realização da análise do código-fonte é essencial para que o usuário tenha o controle e a liberdade para definir seu próprio processo de análise quando for necessário.

3.1.3 Apresentação dos Resultados

Com a realização da análise do código-fonte concluída os resultados devem ser aresentados para o usuário de maneira adequada para que o mesmo possa conseguir utilizar aquela informação para a tomada de alguma decisão. Em geral, ferramentas que desejam chegar ao mercado, e não apenas o público acadêmico, devem permitir que os usuários agrupem e ordenem os resultados, eliminem resultados indesejados e a ferramenta deve explicar a significância dos resultados (CHESS; WEST, 2007).

As ferramentas mais utilizadas no mercado para a realização de análise estática de segurança de código-fonte são integradas a alguma *IDE* (*Integrated Development Environment*), o que facilita para o desenvolvedor utilizá-la, sendo os *reports* das ferramentas

apresentados na maioria das vezes diretamente no código e sendo autoexplicativos, inclusive apresentando métodos para a validação da possível vulnerabilidade e qual seria uma possível correção.

3.1.4 Classificação das Ferramentas

Além de entender sobre o funcionamento das ferramentas de análise estática de segurança de código-fonte, é importante saber como as mesmas são classificadas. Segundo Black (2001), as ferramentas podem ser classificadas baseado em como elas realizam as suas análises, podendo ser uma análise *sound*, heurística ou completa.

Uma ferramenta classificada como *sound* ela é 100% correta em todos os seus julgamentos (BLACK, 2001), ou seja, todas as ameaças de vulnerabilidades que a ferramenta aponta realmente são vulnerabilidades. Isso não quer dizer que esse tipo de ferramenta é a ferramenta ideal, pois sempre garantir que o que ela está dizendo é verdade não assegura que todas as vulnerabilidades do código foram encontradas. A principal característica desse tipo de ferramenta é a baixa taxa de falsos positivos, entretanto, a taxa de falsos negativos pode ser grande, que seria o caso de haver vulnerabilidade no código-fonte e a mesma não indicar-la.

Agora quando a análise é feita baseada nas regras que podem ser automaticamente derivadas do código através de *machine learning* do código existente a ferramenta é classificada como heurística (BLACK, 2001). Por exemplo, quando se encontra uma função *open()* durante a análise do código, espera-se que exista em alguma lugar uma função *close()*, esse tipo de análise pode trazer vários falsos positivos, assim como falsos negativos.

Tendo em vista esses dois tipos de análise, o ideal seria a união dos dois tipos de análise apresentadas, as ferramentas que tentam implementar isso são chamadas de ferramentas completas. Esses ferramentas geralmente possuem resultados bem melhores do que ferramentas apenas heurísticas ou *sound*, e se utilizam de técnicas já mencionadas neste capítulo como acompanhamento do fluxo de dados e controle de fluxo de execução (BLACK, 2001).

Na Seção 5.3 foram análisadas algumas ferramentas de análise estática de código-fonte, sendo selecionada a ferramenta *Cppcheck*, que apesar de tentar se apresentar como uma ferramenta *sound* foi a que melhor se adaptou ao contexto da pesquisa. Em situações reais percebe-se que o desenvolvimento de uma ferramenta completa é algo ainda em evolução na área.

4 Análise Exploratória de Dados

A análise exploratória de dados, mais conhecida como *Exploratory Data Analysis* (*EDA*), são procedimentos para a análise de dados, técnicas para a interpretação dos resultados de tais procedimentos, formas de planejamento da coleta de dados para fazer a sua análise mais fácil, mais precisa e mais acurada, e todo o maquinário e resultados estatísticos que se aplicam aos dados (TUKEY, 1961). A análise exploratória de dados traz uma variedade de técnicas para, por exemplo, encontrar características escondidas dos dados, extrair variáveis importantes e detectar anomalias e *outliers*, sendo *outlier* um número que é muito maior ou menor que o restante dos números em uma dada série de números (HAWKINS, 1980). Segundo Wasiak (2012), o principal objetivo da abordagem de análise exploratória de dados é adiar as premissas habituais sobre que tipo de modelo os dados seguem com uma abordagem mais direta de permitir que os dados revelem a sua estrutura e o modelo.

Para apresentar a análise de dados exploratória foi feita uma rápida comparação com outros métodos de análise de dados, sendo eles as análises clássica e bayesiana. A Figura 4 apresenta um diagrama de sequência de atividades dos métodos de análises de dados que aqui serão comparados.

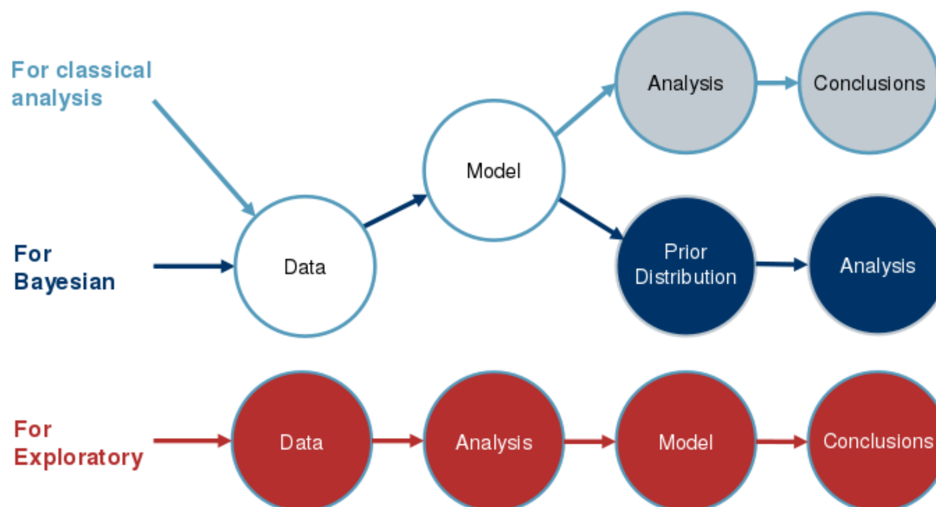


Figura 4 – Sequência de atividades dos métodos de análise (WASIAK, 2012)

Como pode-se ver na sequência de atividades apresentadas na Figura 4, os métodos clássico e bayesiano tendem a definir um modelo para os dados antes de realizar qualquer tipo de análise, sendo essa feita apenas após a definição do modelo. Esse tipo de abordagem pode levar à elaboração de modelos não satisfatórios e isso só será identificado após a análise desse modelo construído. Esse tipo de erro torna o processo de definição de um

modelos bastante custoso, onde deverá ser realizado o processo de análise várias vezes até atingir um resultado satisfatório. Já a abordagem de análise exploratória de dados, como foi dito anteriormente no início do capítulo, visa a exploração dos dados antes da definição do modelo, deixando os dados apresentarem as suas estruturas e qual possível modelo melhor se adequa àqueles dados.

4.1 Análise dos dados

No início da análise exploratória de qualquer tipo de dados é importante entendê-los, verificar possíveis correlações entre os mesmos e se necessário derivar novas características dos dados que auxiliem na exploração dos mesmos.

Para isso pode-se utilizar de algumas técnicas como o coeficiente de correlação de Pearson, sendo essa uma técnica da estatística descritiva, que visa medir a direção e grau com que duas variáveis, de tipo quantitativo, se associam linearmente (ROSSMAN, 1996). Uma técnica que se pode utilizar do método de correlação de Pearson é a construção de uma matriz de correlação, onde todos os dados são confrontados para tentar encontrar algum tipo de dependência linear entre os mesmos.

Com essas informações em mãos pode-se realizar por exemplo uma engenharia de características, sendo engenharia de características a utilização de características existentes para gerar novas características que aumentem o valor da base de dados original, utilizando conhecimento dos dados ou do domínio em questão (BRINK; RICHARDS, 2014). Além disso, pode-se remover características dispensáveis do conjunto de dados, que possam ser redundantes com outras características por exemplo. Em suma, engenharia de característica é o processo de transformar dados nunca trabalhados em características que melhor representam o problema atacado para o modelo preditivo, resultando em uma precisão de modelo melhorada nos dados escondidos (BROWNLEE, 2014).

4.2 Técnica de Visualização dos Dados

Para a realização dessa análise exploratória dos dados antes da definição de um modelo são utilizados vários tipos de técnicas gráficas, tornando-as muito importantes dentro do processo. Uma técnica simples e que reune um grupo de gráficos para trazer uma visão geral sobre os dados trabalhados é a *4-Plot* trazido pelo *Handbook*[1] sobre análise exploratória desenvolvido pelo *NIST* (*National Institute of Standards and Technology*). Essa técnica nos possibilita entender desde se a distribuição dos dados segue uma distribuição normal até a sua aleatoriedade. Identificar esses tipos de características revelam informações importantes para a definição de um modelo que satisfaça determinado

[1] <http://www.itl.nist.gov/div898/handbook/eda/section3/eda3332.htm>

conjunto de dados, ou pelo menos eliminar opção indesejadas. Os tipos de representação gráfica apresentados por essa abordagem *4-Plot* são histograma, *Q-Q Plot*, *Run Sequence* e *Lag Plot*, como pode ser visto na Figura 5, sendo o gráfico inferior esquerdo o *Run Sequence*, inferior direito o *Lag Plot*, superior esquerdo o histograma e o superior direito o *Q-Q Plot*.

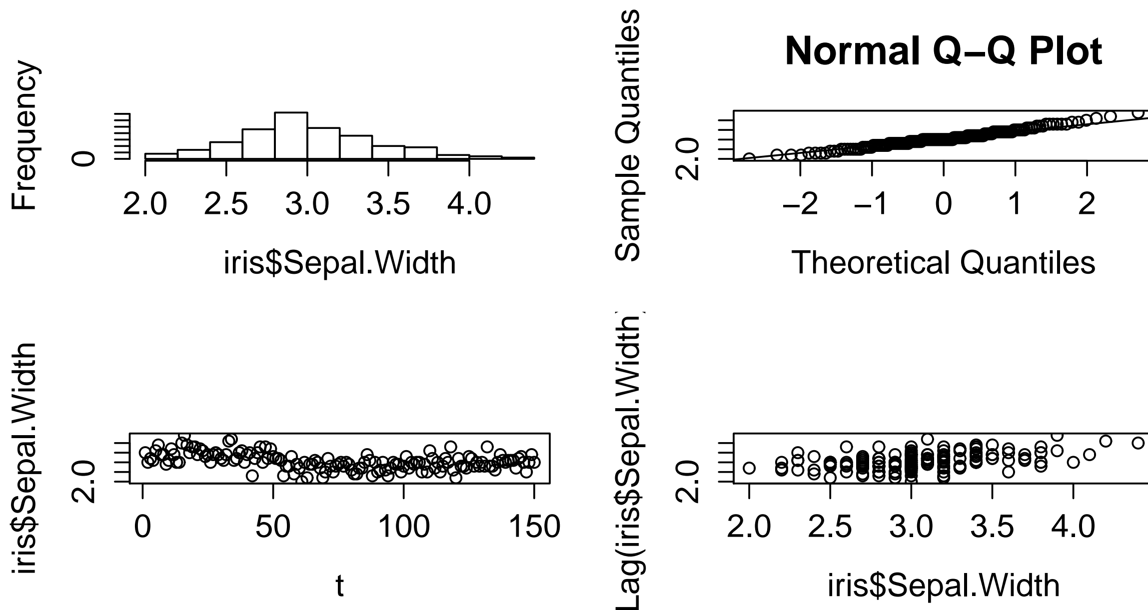


Figura 5 – Exemplo *4-Plot*

Através do histograma consegue-se identificar a distribuição dos dados, a partir dele juntamente com o *Q-Q Plot* pode-se identificar se a distribuição dos dados segue uma distribuição normal ou não, sendo o *Q-Q Plot* um gráfico que confronta os quantis de uma distribuição normal e os quantis da distribuição dos dados em questão. se ambos forem similares, tendem a traçar uma reta diagonal. O gráfico *Run Sequence* visa apresentar uma forma de verificar se a variância da distribuição dos dados é constante ou não, apresentando possíveis deslocamentos, para isso é plotado um índice incremental pelo valor variável dependente. Já o gráfico *Lag Plot* tenta mostrar a aleatoriedade dos dados, onde são plotados seguindo a ordem do conjunto de dados o seu valor diretamente antecessor pelo valor atual, dessa forma, se os dados da distribuição forem realmente aleatórios os pontos deverem ficar espalhados pelo gráfico, caso siga algum tipo de padrão, a hipótese é negada.

4.3 Identificação de *Outliers*

Um passo importante para atingir um modelo que se adeque bem aos seus dados é a identificação e remoção dos possíveis *outliers*. Um método bastante simples para identificação de *outliers* apresentado por Tukey (1977) é utilizar o gráfico *boxplot*. Como

pode ser visto na Figura 6, o gráfico possui uma linha central da caixa representanado a mediana do conjunto de dados, a superior o 3º quartil e a inferior o 1º quartil, a diferença entre os quartis é chamada *IQR* (*Interquantile Range*). A técnica trazido por Tukey (1977) consiste em limitar os valores válidos, ou seja, não *outliers*, até 150% do valor do respectivo *IQR* a mais e a menos dos quartis que limitam a caixa, valores que extrapolem isso são considerados *outliers*.

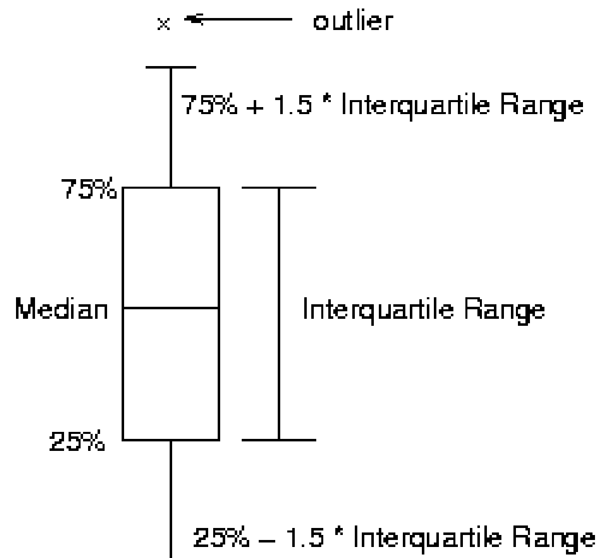


Figura 6 – Explicação sobre identificação de *outliers* com *boxplot*

4.4 Definição de Modelos

Após o entendimento de algumas características inerentes aos dados e identificação e remoção de *outliers*, necessita-se definir um modelo que se adapte ao conjunto de dados com uma certa flexibilidade, ou seja, evitando o *overfitting*, que seria o modelo se ajustar demais aos dados ao ponto de o mesmo não conseguir inferir outros valores. Para a definição de um modelo pode-se utilizar métodos paramétricos ou não paramétricos dependendo das informações que se tem em mão e que foram extraídas do conjunto de dados. Um modelo não paramétrico bastante robusto e simples de utilizar é o *Locally Weighted Regression* (muito conhecido como *LOESS*) apresentado por Cleveland e Devlin (1988), esse método possui três objetivos principais: exploração dos dados, diagnosticar métodos não paramétricos e prover uma superfície suave utilizando um regressão não paramétrica. Com isso, esse método pode nortear a definição de um possível modelo paramétrico. Basicamente esse modelo realiza várias pequenas regressões no conjunto de dados, conhecido como janelamento. Ele define pesos para cada uma dos pontos vizinhos dependendo da função que se utiliza, em geral, se utiliza uma função tricúbica (CLEVELAND; DEVLIN, 1988). Essas várias regressões acabam resultando em um modelo que apresenta uma curva de predição bastante suave, sendo um ponto a se considerar o fato do método ser "caixa

preta”, não permitindo a obtenção de uma fórmula matemática que represente o modelo. Outra forma de se definir um modelo é a partir da utilização de um método paramétrico, sendo a regressão polinomial um dos mais conhecidos. A regressão polinomial implementada por boa parte das ferramentas geralmente é feita através do método dos mínimos quadrados. O método dos mínimos quadrados tem como objetivo encontrar um modelo cuja “energia do erro” seja a menor possível. Tenta-se, dessa forma, minimizar o valor médio quadrático do erro entre valores obtidos pelo modelo e o dado coletado na observação.

4.5 Validação de Modelos

Para a validação de modelos construídos pode-se utilizar várias técnicas, aqui serão apresentadas a *ANOVA* e validação cruzada *K-Fold*. A Análise de Variância, mais conhecida como *ANOVA*, visa analisar a variância entre o resultado obtido pelo modelo e o valor real, também sendo utilizada para comparação de modelos, onde é possível verificar a variância quando se adiciona algum elemento novo ao modelo. Para a utilização da *ANOVA* algumas suposições acerca do modelo em questão devem ser respeitadas, como independência entre os valores ou aleatoriedade, a normalidade dos dados e a igualdade das variâncas do conjunto de dados (SNEDECOR; COCHRAN, 1967). Como pode-se ver a abordagem *4-Plot*, apresentada na Seção 4.2, consegue responder todas essas suposições listadas.

Outra técnica que pode auxiliar na validação de um modelo é a técnica de validação cruzada *K-Fold*, que consiste em dividir o seu conjunto de dados em K grupos, onde um deles é usado para o teste do modelo e os outros para o treinamento do mesmo. A acurácia dos resultados pode então ser medida por meio da comparação dos resultados obtidos com os esperados (ARAúJO, 2011). O passo-a-passo dessa técnica, segundo Araújo (2011), pode ser visto a seguir:

1. O conjunto de dados original é particionado aleatoriamente em k subconjuntos
2. Em cada uma das K rodadas:
    - Um dos subconjuntos é reservado para testar o modelo
    - Os demais subconjuntos são passados ao modelo como dados de treinamento
    - Uma predição é gerada e avaliada por meio de métricas pertinentes
3. Ao final dos testes, os k resultados são combinados para produzir uma estimativa única

Esse tipo de validação cruzada consegue nos mostrar como que o modelo se comportaria em uma situação real de predição, onde os valores que devem ser preditos não

fazem parte do conjunto de treinamento. Como são conhecidos os valores reais desses pontos, o erro de predição do modelo pode ser calculado, usando por exemplo o erro médio quadrático, que nada mais é do que a diferença entre o valor estimado e o valor real dos dados, ponderados pelo número de termos (CAETANO, 2012), como pode ser visto na equação a seguir:

$$EMQ = \sum (y - \hat{y})^2/n$$

Na equação acima, o valor de y representa o valor estimado, y chapéu o valor real e n o número total de elementos dentro do conjunto de dados. Outra métrica que também pode ser utilizada é o erro residual padrão que nada mais é do que o desvio padrão do erro residual, que é a diferença entre o valor real e o predito.

Neste capítulo foram apresentados de forma objetiva apenas as técnica e conceitos utilizados neste trabalho, não explorando os diferentes métodos e tipos de abordagem disponíveis, apesar desse ramo da estatística ser amplo e haver diferentes formas de abordar um mesmo problema. Na sequência deste trabalho temos a aplicação das técnicas e conceitos apresentados neste capítulo.

5 Metodologia

Neste capítulo será apresentada toda a metodologia utilizada durante a pesquisa realizada, podendo ser um norte para a reprodução da mesma em trabalhos futuros. Inicialmente, a questão de pesquisa a ser respodida com este trabalho era a seguinte:

Como podemos monitorar e acompanhar métricas de ameaças de vulnerabilidade de código-fonte dentro do ciclo de desenvolvimento de software?

Na primeira parte do trabalho foram levantadas algumas hipóteses com relação ao monitoramento das métricas de ameaças de vulnerabilidades de código-fonte, levando em consideração trabalhos já realizados sobre métricas de orientação a objetos e de tamanho, como o trabalho de Meirelles (2013). A hipótese foi a seguinte:

- *H1*: As métricas de ameaças de vulnerabilidade de código-fonte podem ser observadas de maneira similar às métricas de *design* de código fonte.

Ao final da primeira parte da pesquisa foram respondidas as hipóteses levantadas inicialmente. Através da análise das métricas de ameaças de vulnerabilidade de código-fonte extraídas do projeto *Linux Kernel* pode-se perceber que boa parte dos valores das métricas eram nulos (zero), o que faz sentido, já que espera-se que não exista ameaças de vulnerabilidades em todos os módulos do projeto. Dessa forma, as métricas de ameaças de vulnerabilidade não se assemelham com métricas de *design* de código, pois as métricas de *design* de código não costumam ter valoração nula na maioria dos casos devido a sua própria natureza, logo, ambas não podem ser observadas de maneira similar, o que nos faz negar a hipótese *H1*. Além disso, foi realizada uma análise dos percentis das métricas, que pode ser visto no Apêndice A, assim como foi realizada para as métricas de *design* em Meirelles (2013), e percebeu-se que ambas diferem, o que corrobora com a decisão tomada. Sendo que os percentis são medidas que dividem uma amostra ordenada, em ordem crescente, em cem partes, cada uma com uma porcentagem de dados aproximadamente igual (MARTINS, 2013).

Foi realizado um estudo qualitativo das métricas em questão, foram analisados outros dez projetos, que podem ser vistos no apêndice B. Após uma análise mais detalhada dos valores das métricas dos projetos chegou-se a um subconjunto mais frequente das mesmas em projetos de software livre, ou seja, as ameaças de vulnerabilidade mais encontradas nesses projetos, sendo elas:

- Referência a ponteiros nulos (CWE 476)

- Variáveis não inicializadas (CWE 457)

- Vazamento de memória (CWE 401)

Esses cenários de vulnerabilidades identificados na primeira etapa da pesquisa serviram de insumo para a continuação da mesma, que será apresentada no decorrer deste capítulo. Uma discussão mais detalhada sobre as CWEs (*Common Weaknesses Enumaration*) pode ser vista na Seção 2.1.

Apesar de obter algumas respostas nesse início de pesquisa outras questões foram levantadas, direcionando a pesquisa para a definição de modelos de predição explicados a seguir.

5.1 Trabalhos Relacionados

Durante a pesquisa realizada encontrou-se alguns trabalhos voltados para a avaliação de ferramentas de análise estática de ameaças de vulnerabilidade de código, como em Rutar e Foster (2004), não levando em conta o ciclo de desenvolvimento de software, mas apenas o desempenho das ferramentas.

Em Zheng et al. (2006), foi analisada a efetividade de ferramentas de análise estática desse tipo, tendo como parâmetro os testes e o número de falhas reportadas pelos clientes. Conclui-se que ferramentas de análise estática são efetivas para encontrar defeitos a nível de código, entretanto, não é apresentada uma solução para como monitorar os mesmos.

No trabalho realizado pelo *National Institute of Standards and Technology* (NIST) (OKUN et al., 2007) foi feito um estudo inicial tentando identificar se a inserção de uma ferramenta de análise estática de vulnerabilidades de código-fonte dentro do ciclo de desenvolvimento de um software aumenta a segurança do mesmo. E a conclusão desse trabalho foi que não necessariamente a utilização dessas ferramentas melhora a segurança do software. A definição de modelos para o monitoramento dessas ameaças de vulnerabilidade de código-fonte não foi abordado nesse trabalho.

Entretanto, não se encontrou um trabalho que tentasse definir algum modelo estatístico para monitorar e controlar métricas de ameaças de vulnerabilidade de código-fonte dentro do ciclo de desenvolvimento de um software, sendo modelos esses a principal contribuição deste trabalho.

5.2 Projeto da Pesquisa

Com o objetivo de definir modelos de predição para as métricas de ameaças de vulnerabilidade de código-fonte aqui trabalhadas, a questão de pesquisa redefinida é:

É possível o desenvolvimento de modelos preditivos de referência, de baixa complexidade, para acompanhamento e monitoramente de métricas de ameaças de vulnerabilidade de código-fonte em projetos de software?

Tendo essa questão de pesquisa em mente o objetivo deste trabalho é encontrar uma função matemática que viabilize o monitoramento e acompanhamento dessa classe de métricas. E para encontrar esse meio de realizar esse monitoramente e acompanhamento das métricas de ameaças de vulnerabilidade de código-fonte foram levantadas seguintes as hipóteses:

- *H2*: Os valores das métricas de ameaças de vulnerabilidade de código-fonte se comportam como distribuições estatísticas de cauda longa, e não distribuições estatísticas normalizáveis.
- *H3*: A definição de um modelo baseado em uma função polinomial possibilita o monitoramento e acompanhamento das métricas de ameaças de vulnerabilidade de código-fonte.

Através de uma análise exploratória dos dados espera-se responder a hipótese *H2*, onde será possível ter uma visão geral dos dados. E através da definição e validação de modelos estatísticos será possível responder a hipótese *H3*.

Com o intuito de responder as hipóteses levantas, foi definido um fluxo de atividades para sistematizar esta pesquisa. A Figura 7 ilustra o processo com o fluxo de atividades que foram desenvolvidas, a fim de uma melhor compreensão do trabalho realizado.

Detalhamento das atividades apresentadas na Figura 7:

1. **Selecionar projeto**: Selecionar projeto de software livre representativo no contexto de segurança, para que possa ser feita a análise sobre o mesmo. O projeto deve ser um projeto já consolidado e com várias versões para que possa ser feita uma análise através do tempo.
2. **Obter código-fonte**: Obter código-fonte das versões disponíveis do projeto selecionado, para que possa ser realizada a análise estática. A obtenção do código-fonte deve ser feita de forma automatizada.

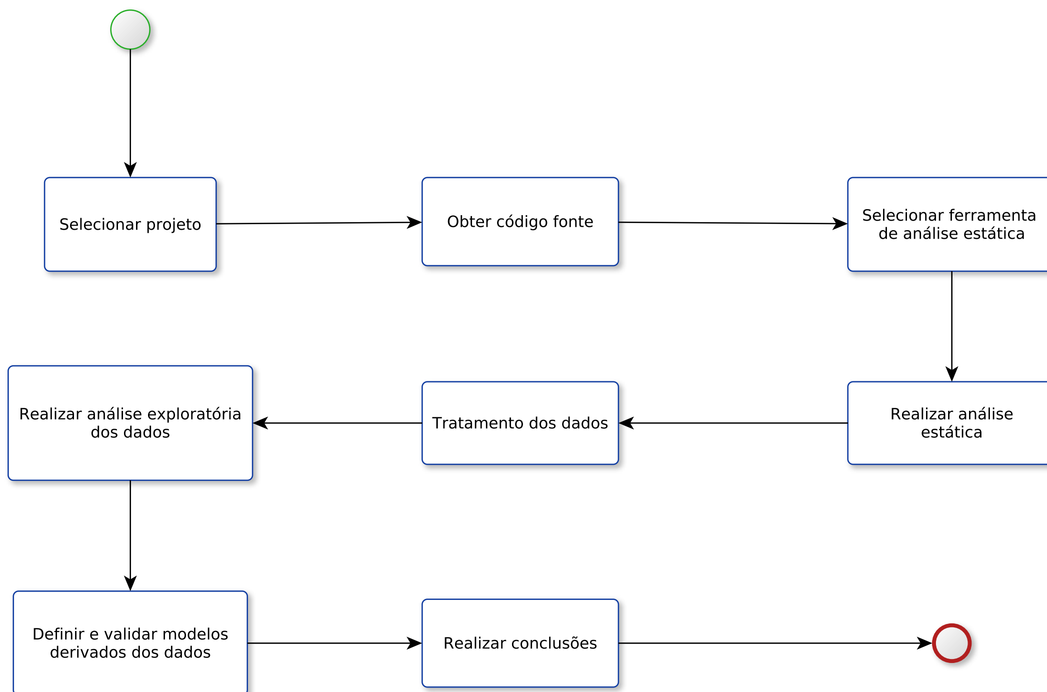


Figura 7 – Processo contendo atividades executadas durante a pesquisa

3. **Selecionar ferramenta de análise estática**: Analisando os pontos fortes e fracos das ferramentas disponíveis, selecionar a ferramenta que melhor se adeque ao contexto em questão.

4. **Realizar análise estática**: Realizar a análise estática sobre o código-fonte obtido com a ferramenta selecionado de maneira automatizada.

5. **Tratamento dos dados**: Ajustar o formato de saída da ferramenta utilizada para um arquivo CSV (*Common Separated Values*) se a ferramenta não prover, além de tratar os dados descartando o que não for preciso e compondo os dados para gerar novos dados se necessário. Também deve ser feito de forma automatizada.

6. **Realizar análise exploratória dos dados**: Utilizando uma ferramenta estatística, explorar os dados a fim de entender o comportamento do mesmo, para que facilite a dedução de um modelo próximo do real.

7. **Definir e validar modelos derivados dos dados**: Encontrar modelos estatísticos (funções matemáticas) para monitoramento, acompanhamento e previsão das métricas de ameaças de vulnerabilidade de código-fonte. Validar os modelos com dados que não foram empregados na construção do mesmo.

8. **Realizar conclusões**: Comparar os modelos definidos e validados e indicar qual seria o melhor modelo para o monitoramento, acompanhamento e previsão das métricas em questão.

5.3 Teste das Hipóteses

O projeto selecionado para testar as hipóteses foi o *Linux Kernel*, assim como o mesmo foi selecionado em Raymond (1997) para se entender o ecossistema e as práticas, inclusive de engenharia de software, pela comunidade liderado por *Linus Torvalds*. Devido o *Linux Kernel* seguir um modelo *Bazaar* de desenvolvimento (RAYMOND, 1997), a correção de *bugs*, possíveis ameaças de vulnerabilidade, se dá de maneira muito mais rápida, o que torna interessante a análise dessa nova classe de métricas em um software bastante consolidado. Segundo *Linus Torvalds*, com uma grande quantidade de testadores beta e co-desenvolvedores, qualquer problema vai ser identificado rápido e a solução será óbvia para alguém. Sendo esse o primeiro projeto a explorar a rede de colaboração de software livre nos moldes atuais (RAYMOND, 1997), tornando-se uma referência. Partiu-se do *Linux Kernel* para se estudar sobre práticas de desenvolvimento de software e a própria engenharia de software, e agora será dado o primeiro passo com relação a definição de um modelo estatístico que auxilie no monitoramento de métricas de ameaças de vulnerabilidade de código fonte.

Obtenção do código-fonte O código-fonte do projeto *Linux Kernel* foi obtido no espelho (*mirror*) do Github[1] do repositório oficial Git do projeto[2]. Fez-se uma cópia local do repositório e foram utilizados alguns comandos *bash* (via terminal) para que a obtenção do código de todas as *tags* disponíveis fosse feita de uma única vez. Nesse repositório não havia as *tags* de todas as versões, pois o projeto passou a utilizar o Git[3] apenas a partir da versão 2.6.11, entretanto, foi possível obter o código de 391 *tags* (da versão 2.6.11 até 3.9, contando com todas as *releases candidates* ou *releases* intermediárias), sendo considerado um número representativo e suficiente para a realização do estudo.

Seleção da ferramenta de análise estática A ferramenta de análise estática de código-fonte selecionada para a realização deste estudo foi o *Cppcheck*[4]. A parte inicial da pesquisa se deu com a ferramenta *Clang Static Analyzer*[5](um submódulo do compilador *Clang*), esta ferramenta é bastante robusta e consegue capturar bem as ameaças de vulnerabilidade de código-fonte que a mesma se propõe, sendo essa uma ferramenta que está focada em se tornar uma ferramenta completa. Entretanto, ela faz uma análise inter-procedural do código, o que necessita da compilação do mesmo. Infelizmente, o projeto *Linux Kernel* ainda não suporta a sua total compilação utilizando outro compilador a não ser o GCC[6]. Existe um projeto chamado *LLVMLinux*

[1] <https://github.com/torvalds/linux>
[2] <https://git.kernel.org/cgit/>
[3] Ferramenta de controle de versão descentralizado
[4] <http://cppcheck.sourceforge.net/>
[5] <http://clang-analyzer.llvm.org/>
[6] <https://gcc.gnu.org/>

Project[7] que está tentando fazer com que o *Linux Kernel* possa ser compilado com o *Clang*, mas esse trabalho ainda está em andamento. Logo, decidiu-se abandonar o *Clang Static Analyzer* e encontrar uma analisador estático cujo qual não fosse necessária a compilação do código-fonte, sendo esse o *Cppcheck*, que apesar de não realizar uma análise inter-procedural, ele se propõe a não emitir uma grande quantidade de falsos positivos, chamada ferramenta *sound*, podendo ser entendido mais na Seção 3.1.4.

Análise estática do código-fonte Após a seleção da ferramenta a ser utilizada, extrair do código-fonte as ameaças de vulnerabilidade foi relativamente simples. O código-fonte referente a todas as *tags* foi armazenado em um mesmo diretório, sendo o *Cppcheck* capaz de percorrer recursivamente o diretório a procura de arquivos de código-fonte C e C++ para realizar a análise, foi necessário apenas executar a ferramenta nesse diretório. Foram feitas algumas configurações para a geração de um arquivo de saída para cada uma das *tags*. O arquivo de saída de todas as análises realizadas estão disponíveis no repositório.[8]

5.3.1 Descrição dos dados e Engenharia de Características

Levando em consideração as principais ameaças de vulnerabilidades levantadas na primeira parte da pesquisa, a saída da ferramenta foi tratada a fim de filtrar essas ameaças de vulnerabilidades, além de criar um arquivo *CSV* com esses dados. O tratamento dos dados e geração do arquivo *CSV* foram feitos através dos scripts disponíveis no repositório[9], assim como o próprio arquivo *CSV* gerado. A Tabela 1 apresenta a estrutura do arquivo *CSV* gerado.

CWE476	CWE457	CWE401	Módulos
XXX	XXX	XXX	XXX

Tabela 1 – Estrutura do arquivo *CSV* gerado.

As colunas “CWE476”, “CWE457” e “CWE401” referem-se a quantidade total de cada uma das ameaças de vulnerabilidades em toda a versão em questão, e a coluna “Módulos” representa a quantidade total de módulos daquela versão. A seguir são apresentados os tipos de cada um dos dados:

- **CWE476**: Inteiro >= 0
- **CWE457**: Inteiro >= 0

[7] <http://llvm.linuxfoundation.org/index.php/Main_Page>
[8] <https://github.com/lucaskanashiro/linux-analysis/tree/master/data>
[9] <https://github.com/lucaskanashiro/linux-analysis>

- **CWE401**: Inteiro >= 0
- **Módulos**: Inteiro > 0

Uma análise levando em consideração apenas esses dados não seria interessante, já que utilizando a quantidade total absoluta de ameaças de vulnerabilidade corre-se o risco de versões com uma maior quantidade de módulos se sobressair em relação as outras. Pensando nisso, foi realizada uma Engenharia de Características, onde ao final se definiu uma taxa de ameaças de vulnerabilidades por módulo, que é facilmente calculada dividindo o total de cada uma das ameaças de vulnerabilidade pela quantidade total de módulos. Essa taxa claramente representa a porcentagem de vulnerabilidades por módulo se multiplicada por 100, nesse caso a mesma deve variar entre 0 e 1. Ao final, os dados trabalhados durante a pesquisa ficaram da forma apresentada na Tabela 2.

Módulos	tax_CWE476	tax_CWE457	tax_CWE401
XXX	XXX	XXX	XXX

Tabela 2 – Estrutura de dados após Engenharia de Características.

5.3.2 Análise Exploratória dos Dados

Tendo todos os dados tratados e uma estrutura de dados bem definida, iniciou-se a fase de análise dos dados. Utilizou-se análise exploratória dos dados, diferente das técnicas estatísticas convencionais.

O primeiro passo foi tentar entender os dados e como os mesmos se relacionam entre si, para isso se utilizou de uma matriz de correlação, onde é dado o índice de correlação entre a permutação de todas as variáveis em questão. Esse índice varia entre -1 e 1, sendo próximo de 1 diretamente relacionada e próximo de -1 inversamente relacionada. A Tabela 3 apresenta a matriz de correlação do dados coletados.

	Referência a ponteiros nulos	Variáveis não inicializadas	Vazamento de memória
Referência a ponteiros nulos	1		
Variáveis não inicializadas	0.01902783	1	
Vazamento de memória	- 0.1286109	0.5671033	1

Tabela 3 – Matriz de correlação.

Através da matriz de correlação pode-se extrair algumas informações. Pode-se ver que, em geral, as taxas de ameaças de vulnerabilidades são inversamante proporcionais a quantidade de módulos. Ou seja, quanto menor a quantidade de módulos, provavelmente um software ainda imaturo, maior a quantidade de ameaças de vulnerabilidade, assim como um software mais maduro tende a ter um maior número de módulos com uma menor quantidade de ameaças de vulnerabilidade de código-fonte. As ameaças de vulnerabilidade em geral não possuem correlação, a exceção seria entre a CWE457(variáveis

não inicializadas) e a CWE401(vazamento de memória), que segundo a matriz podem se correlacionar de maneira direta. Isso pode ser levado em consideração tendo em vista que quando se declara uma variável e não a utiliza, existe uma boa chance da mesma cair em esquecimento e aquela região de memória não ser mais desalocada.

Após entender um pouco mais sobre a corelação entre os dados, foi feito um *plot* (Figura 8) de todas as taxas de ameaças de vulnerabilidade pelo número de módulos para a melhor visualização dos dados.

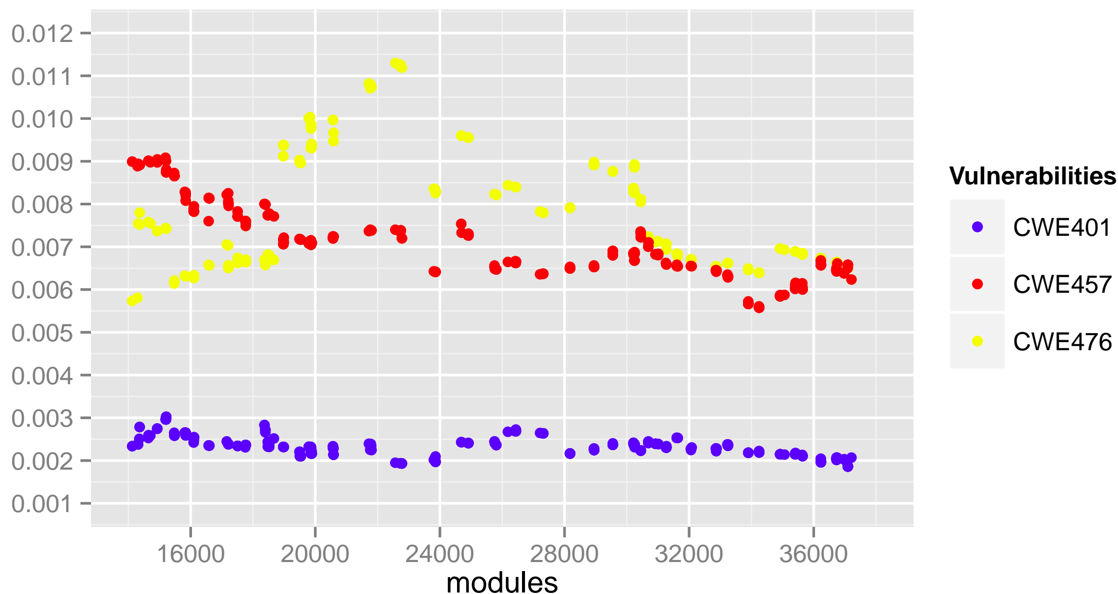


Figura 8 – *Plot* das taxas de CWE pelo número total de módulos

As taxas, como esperado são números bem pequenos, já que o número total de ameaças de vulnerabilidade de código-fonte são bem menores do que o número total de módulos.

Analisando o gráfico pode-se observar que, como foi percebido com a análise da matriz de correlação, a taxa das ameaças de vulnerabilidade por módulo tendem a diminuir quando se aumenta o número de módulos, provavelmente o projeto de software está crescendo e tendo um processo de desenvolvimento mais maduro. No início, o projeto tende a crescer e aumentar a sua complexidade, aumentando as ameaças de vulnerabilidades, depois de certo ponto o projeto amadurece e passa a melhorar o seu processo de desenvolvimento e *design* do código fonte, e as taxas de ameaças de vulnerabilidade tendem a cair.

Como pode-se ver, a CWE401 (vazamento de mémoria) se mantém sem muitas variações mesmo aumentando o número de módulos. Além disso ela possui valores bem abaixo em comparação com as outras taxas, o que nos mostra que um projeto de software em geral, independente da sua maturidade e tamanho, não consegue sobreviver se houver

várias possíveis ocorrências de vazamento de memória. Se isso ocorrer, facilmente o software pode estourar a região de memória alocada para o *HEAP* do programa. Devido a maior constância nas taxas dessa ameaça de vulnerabilidade, não foi elaborado um modelo específico para ela, já que em geral os valores se mantém.

Tendo isso em vista, as outras duas ameaças de vulnerabilidade de código-fonte (CWE476 e CWE457) foram estudadas separadamente a fim de ao final ter um modelo para monitoramento e predição de cada uma.

5.3.2.1 CWE476 - Referência de Ponteiros Nulos

Para analisar especificamente a ameaça de vulnerabilidade de código-fonte em questão, serão utilizados alguns gráficos para entender a taxa referente a CWE476, para que se possa definir um modelo próximo ao real. Foram escolhidos alguns gráficos para realizar esta análise, baseado no 4 *plots* apresentado pelo NIST (*National Institute of Standards and Technology*), como pode ser visto na Figura 9.

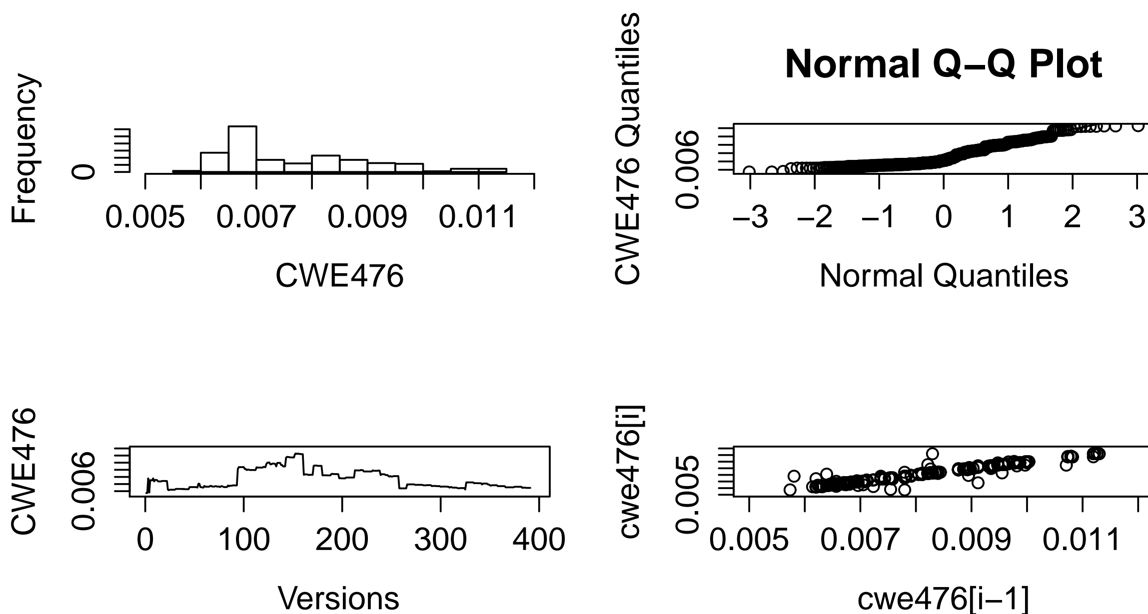


Figura 9 – CWE476 - 4 *plots*

Na sequência será apresentado cada um dos gráficos e feita uma discursão acerca dos mesmos.

O primeiro gráfico apresentado na Figura 9 é um histograma, onde pode-se observar a frequência da taxa desta ameaça de vulnerabilidade. Pode ser visto com mais detalhes na Figura 10.

Percebe-se que a distribuição referente a esta taxa não se assemelha a uma distribuição normal comum, já que a maior ocorrência está deslocada à esquerda, sendo essa similar a uma distribuição de cauda longa. A moda é o intervalo entre 0.0065 e 0.0070.

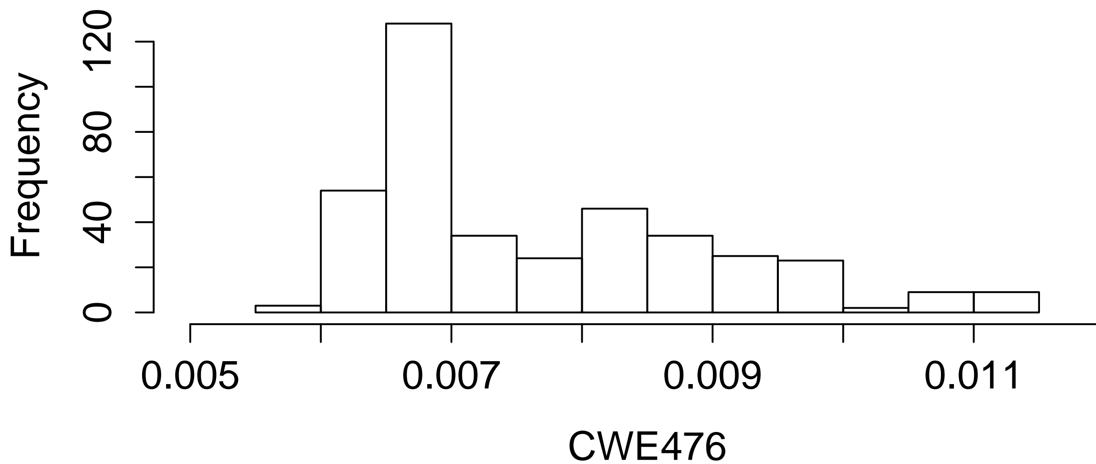


Figura 10 – Histograma da taxa da CWE476

Confirma também a hipótese respondida na primeira parte da pesquisa, onde a média dos valores não é representativa. O gráfico *Q-Q plot* apresentado na Figura 11 vem para corroborar com o que foi dito anteriormente, mostrando que o distribuição da taxa da CWE476 provavelmente não é uma distribuição normal, onde a média é representativa, mas sim uma de cauda longa.

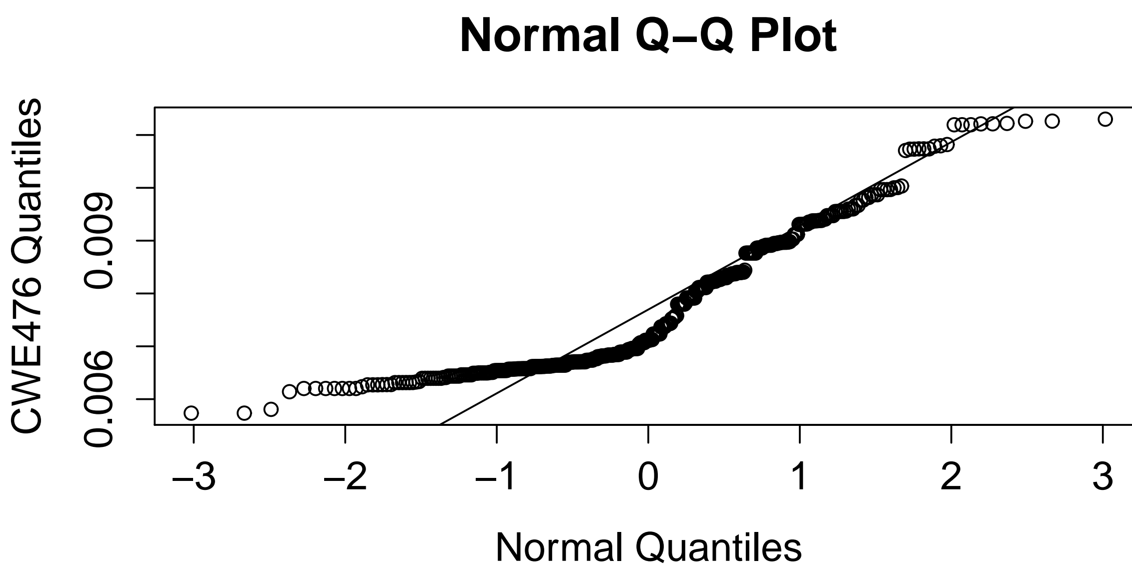


Figura 11 – *Q-Q plot* da taxa da CWE476

No *Q-Q plot* são confrontados os quantis da distribuição normal e os quantis do conjunto de dados da taxa da CWE476. Se a distribuição dos dados que representam

a taxa fosse uma distribuição normal os quantis iriam ser similares, assim os pontos se aproximariam da reta traçada. Quanto mais próximo da reta, mais a distribuição dos dados se assemelha a uma distribuição normal.

Na Figura 12 é apresentado o gráfico de *Run Sequence*, onde o eixo x é composto de uma variável sequencial e incremental, que começa em um e vai até o tamanho do conjunto da variável em questão (neste caso essa variável representa as versões do *Linux Kernel*), e o eixo y é composto pelos próprios dados analisados. Com este gráfico pode-se analisar variações locais e de escala.

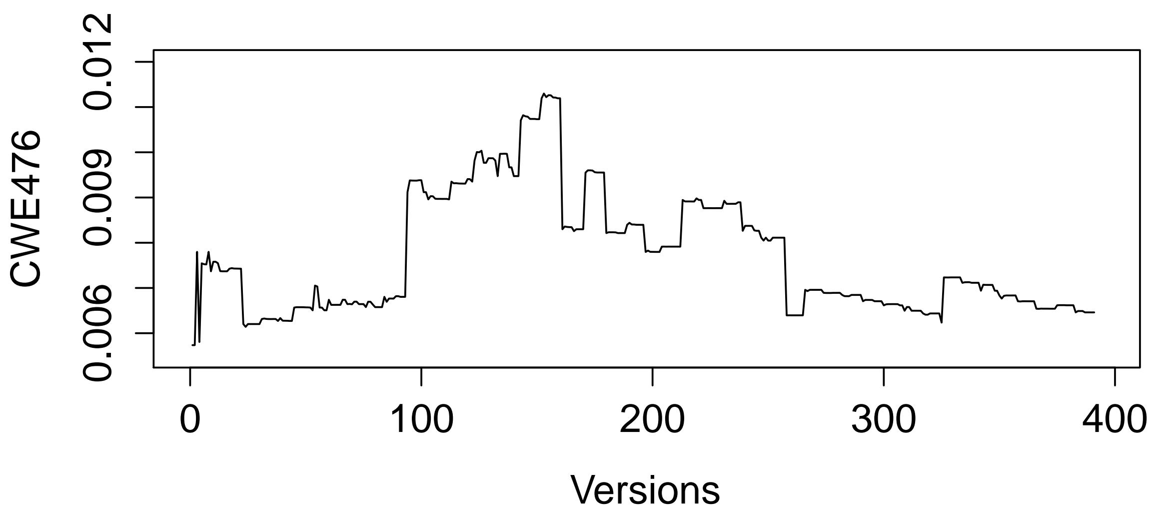


Figura 12 – *Run Sequence* da taxa da CWE476

Para um conjunto de dados que respeita a distribuição normal a variação é praticamente constante, neste caso apresentado na Figura 12, pode-se observar que a variação dos dados não é constante, e possui um pico próximo a versão 150 e depois volta a diminuir a taxa. Isso se deve a uma refatoração do código-fonte do *Linux Kernel* que se iniciou após a versão 2.6.29 (versão 150) até a versão 3.0 (versão 250), que possibilitou a redução de ameaças de vulnerabilidade de código-fonte.

Para encerrar a análise dos dados referentes a taxa da CWE476, é apresentado um *Lag Plot* na Figura 13. Esse gráfico nos auxilia a identificar a aleatoriedade dos dados e a identificar um modelo que se adeque aos mesmos. A construção se dá de maneira simples, onde no eixo y estão os dados, e no eixo x estão os antecessores dos dados do eixo y.

Pela figura pode-se perceber que há uma relação linear entre os dados e os seus antecessores, nos mostrando que este conjunto de dados não é aleatório. Sabendo que a variação dos dados não é constante como o que acontece em um polinômio de primeiro grau, foi realizada uma regressão para polinômios de diferentes graus maiores do que um

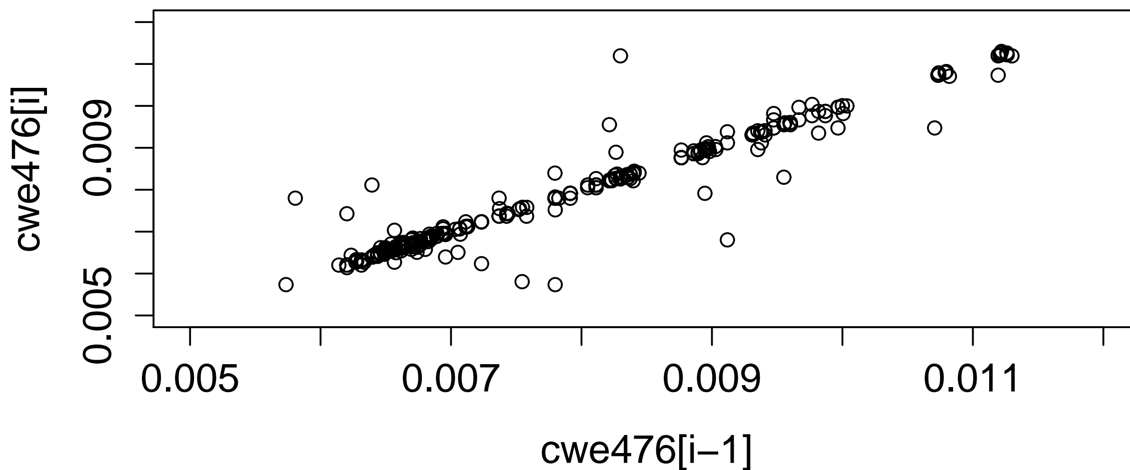


Figura 13 – *Lag Plot* da taxa da CWE476

para tentar encontrar um que melhor se adeque, conforme será apresentado no Capítulo 6.

5.3.2.2 CWE457 - Variável não Inicializada

A abordagem utilizada para o cenário de ameaça de vulnerabilidade anterior também será aplicada no contexto de variáveis não inicializadas. Primeiramente, será apresentado o gráfico *4 Plots*, visto na Figura 14. Em seguida serão destrinchados cada um dos referidos gráficos.

Iniciando a análise pelo histograma, que pode ser visto com mais detalhes na Figura 15. Assim como o histograma apresentado pela taxa da CWE476 (Figura 10), esse aparenta respeitar uma distribuição estatística de cauda longa ao invés de uma normal. Pode-se ver que a moda é o intervalo entre 0.0065 e 0.0070, da mesma forma que o cenário anterior.

O gráfico *Q-Q Plot* nos auxilia a ver de maneira mais clara a não conformidade com a distribuição normal através da Figura 16. Esse gráfico nos mostra que a distribuição dos dados da CWE457 está mais próxima de uma normal do que a CWE476, entretanto, pode-se dizer que a normal também não é representativa nesse caso, pois existe uma cauda longa próxima aos maiores quantis.

Analisando a Figura 17 que contém um gráfico *Run Sequence*, percebe-se que a variação dos valores também não são constantes, descartando uma regressão para um polinômio de primeiro grau. Entretanto, não existem picos que destoam totalmente das outras variações, não demonstrando efeito significativo da refatoção a partir da versão

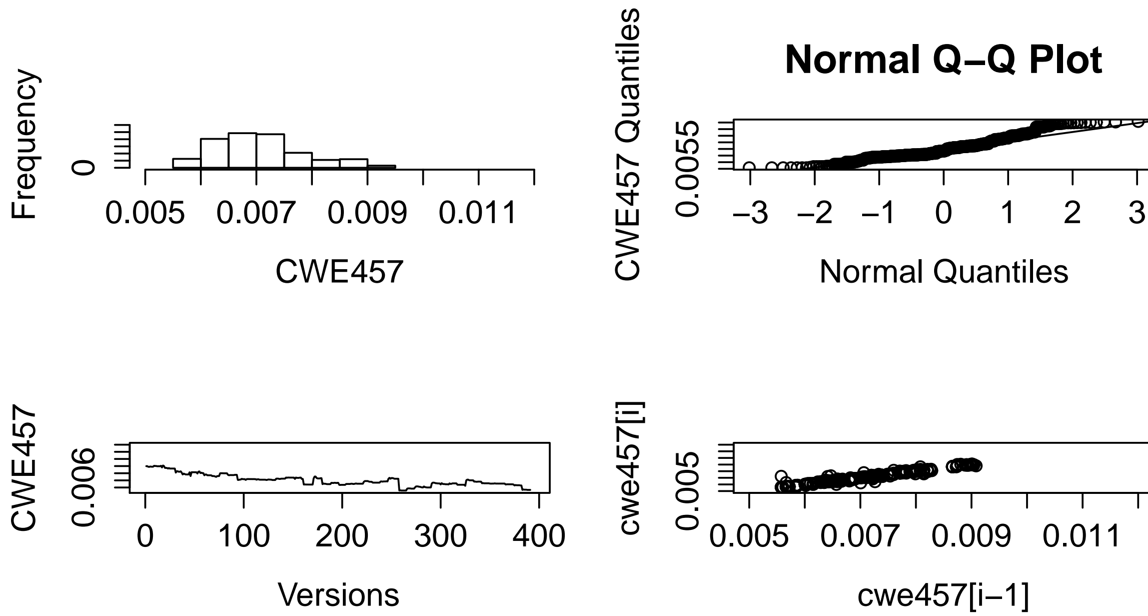


Figura 14 – CWE457 - 4 *plots*

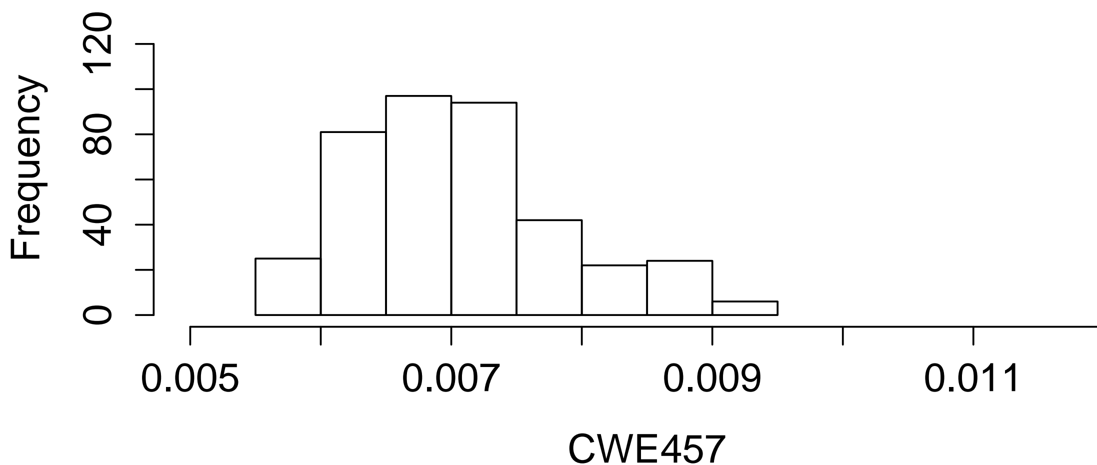


Figura 15 – Histograma da taxa da CWE457

150 (versão 2.6.29) com relação as variáveis não inicializadas.

Finalizando a análise do cenário de ameaça de vulnerabilidade de variáveis não inicializadas, é apresentado o *Lag Plot* na Figura 18. Assim como o apresentado pela CWE476 (Figura 13), existe uma relação linear entre os dados e seus antecessores diretos. Com isso, a decisão tomada também foi a mesma, realizar regressões para polinômios de diferentes graus diferentes de um e tentar confronta-los para identificar qual o melhor modelo.

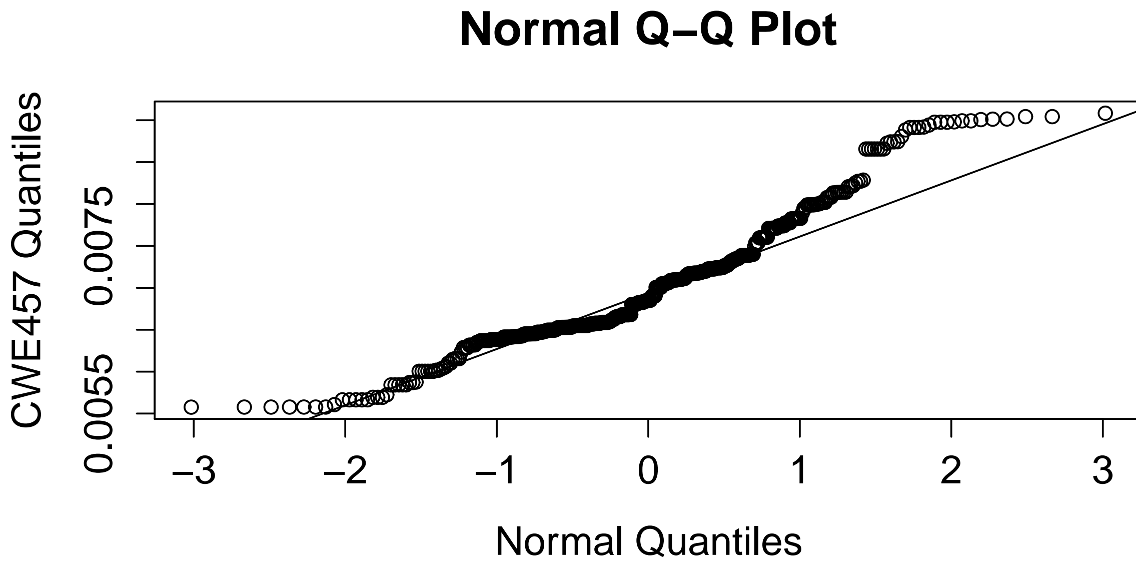


Figura 16 – *Q-Q Plot* da taxa CWE457

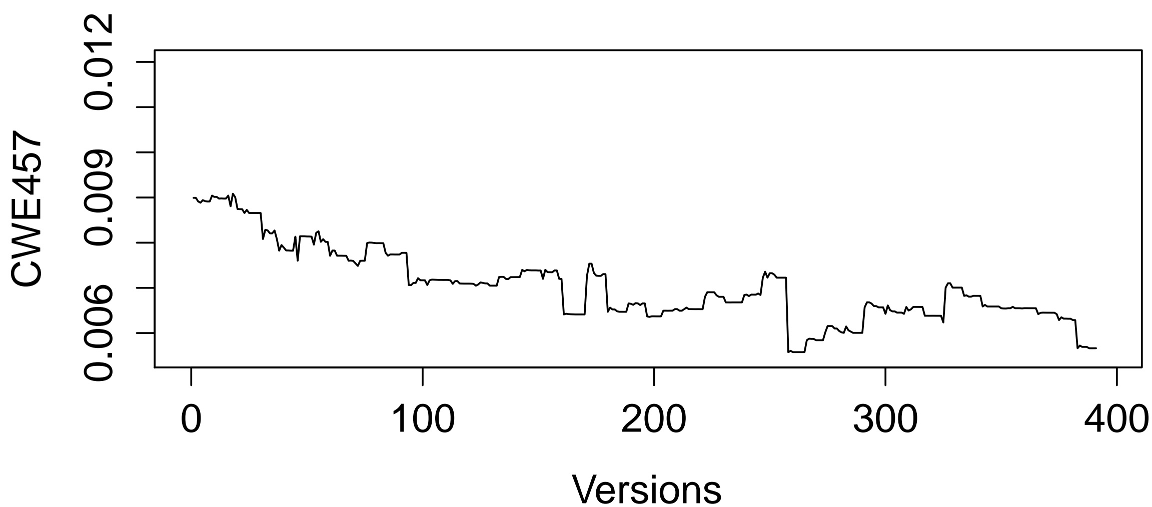


Figura 17 – *Run Sequence* da taxa CWE457

5.3.2.3 Resultados Obtidos

Após a realização do teste das hipóteses levantadas no início da pesquisa (lista 5.2) se tem insumo para chegar em algumas conclusões. A hipótese *H2* não foi negada, já que como pode-se ver nos histogramas e gráficos *Q-Q plot*, apresentados na Seção 5.3.2, que a distribuição das taxas de ameaças de vulnerabilidade de código-fonte trabalhadas seguem uma distribuição estatística de cauda longa, e não uma simples distribuição normal onde a média dos valores é representativa. Sendo esse um ponto que já havia sido discutido no

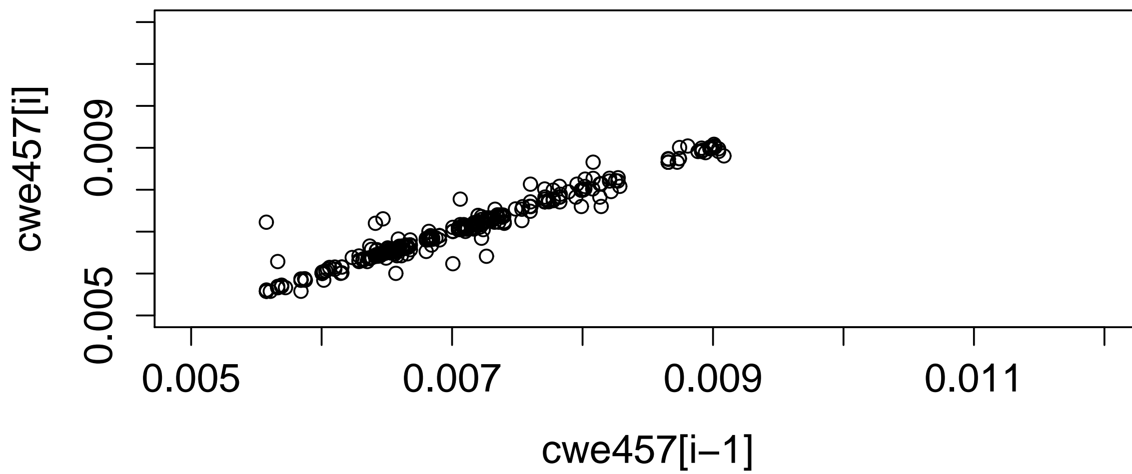


Figura 18 – *Lag Plot* da taxa CWE457

início deste capítulo e que foi corroborado após o trabalho realizado.

Tendo isso em vista, no Capítulo 6 é apresentado o resultado principal desse trabalho, que consistirá na definição de modelos polinomiais para testar a hipótese *H3*

6 Definição dos Modelos de Predição

Como pode ser visto na Seção 5.3.2, as duas métricas de ameaças de vulnerabilidade de código-fonte trabalhadas se comportaram de maneira similar. Mas como o intuito desta pesquisa é encontrar uma função matemática que possibilite o monitoramento e predição de cada uma das métricas, o modelo de ambas serão desenvolvidos separadamente, apesar de seguirem os mesmos métodos e critérios.

As atividades realizadas para definir cada um dos modelos que serão apresentados a seguir foram:

1. Identificar e remover possíveis *outliers*.
2. Dividir o conjuto de dados em conjunto de teste e de treinamento, sendo o conjunto de teste um terço e o de treinamento dois terços do total.
3. Definição de um modelo não paramétrico, que servirá de referência para a definição do modelo paramétrico.
4. Definição de um modelo paramétrico, tentando se aproximar da referência do modelo não paramétrico.

Foi utilizado o método de Tukey (1977) apresentado na Seção 4.3, para a identificação dos *outliers*. Utilizou-se de um método não paramétrico que traçasse uma curva suave baseado no *scatterplot* apenas como referência. Já que o mesmo não nos dá uma função matemática como saída, apesar dos resultados serem bastante satisfatórios. O método não paramétrico utilizado foi o *LOESS* (*Locally Weighted Regression*), como foi explicado na Seção 4.4, o mesmo tem como um dos seus pontos fortes a análise de uma série temporal, como a que estamos trabalhando, valores de métricas de uma série de versões do projeto *Linux Kernel*.

É importante salientar que pode-se representar qualquer conjunto de dados com um polinômio de algum grau, entretanto, para realizar predições baseado no modelo gerado precisa-se evitar o *overfitting*, que seria o modelo se ajustar extremamente ao conjunto de dados e não conseguir extrapolar para novas entradas. Utilizando essa abordagem de utilizar um modelo não paramétrico como referência ajuda a evitar um possível *overfitting* do modelo gerado, chegando a um modelo mais flexível.

Para a definição dos modelos foi utilizada a técnica de regressão polinomial, foram construídos modelos com polinômios de diferentes graus baseado na curva dada pelo método não paramétrico. Os modelos gerados serão comparados na Seção 6.3.

6.1 Definição de modelo para CWE476 - Referência a Ponteiros Nulos

Na tentativa de identificar possíveis *outliers* utilizando o método de Tukey (1977), foi contruído o gráfico *boxplot* (Figura 19). Como pode-se ver não foram identificados *outliers* extremos, todos os pontos da distribuição dos valores da taxa de CWE476 por módulo estão dentro dos limites determinados pelo método, que extrapola para mais e para menos 150% do referido interquantil.

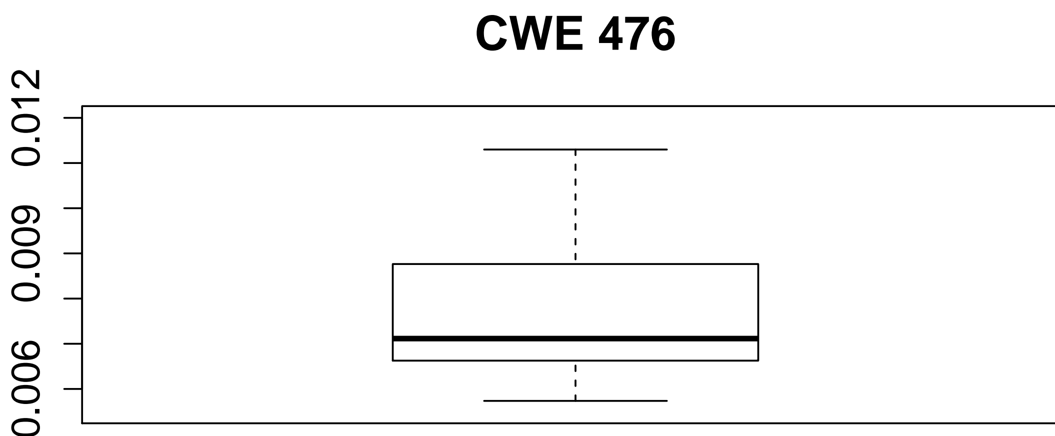


Figura 19 – *Boxplot* da taxa da CWE476 por módulo

Como não foram identificados possíveis *outliers*, o conjunto de dados permaneceu o mesmo, sem sofrer alterações. Com isso, passou-se a definir o modelo não paramétrico, como resultado final se teve uma curva suavizada bastante satisfatória, como se pode ver na Figura 20. Sendo a curva o resultado da predição do modelo definido sobre os dados de treinamento. O erro residual padrão desse modelo é 0.0008044, sendo o erro residual padrão o desvio padrão da diferença entre o valor real e o predito.

Analisando visualmente a curva gerada pelo método não paramétrico pode-se especular que provavelmente uma função quadrática poderia se adaptar bem ao conjunto de dados trabalhado e se aproximar da curva apresentada. Para validação do modelo acerca do conjunto de dados, foi desenvolvido em conjunto um modelo cúbico visando compará-los e chegar a um modelo que se aproxime do desejado e não seja tão custoso para calculá-lo.

O conjunto de dados obtidos a partir da análise do projeto *Linux Kernel* foi dividido entre conjunto de treinamento (65 % dos dados) e teste (35% dos dados). A variável

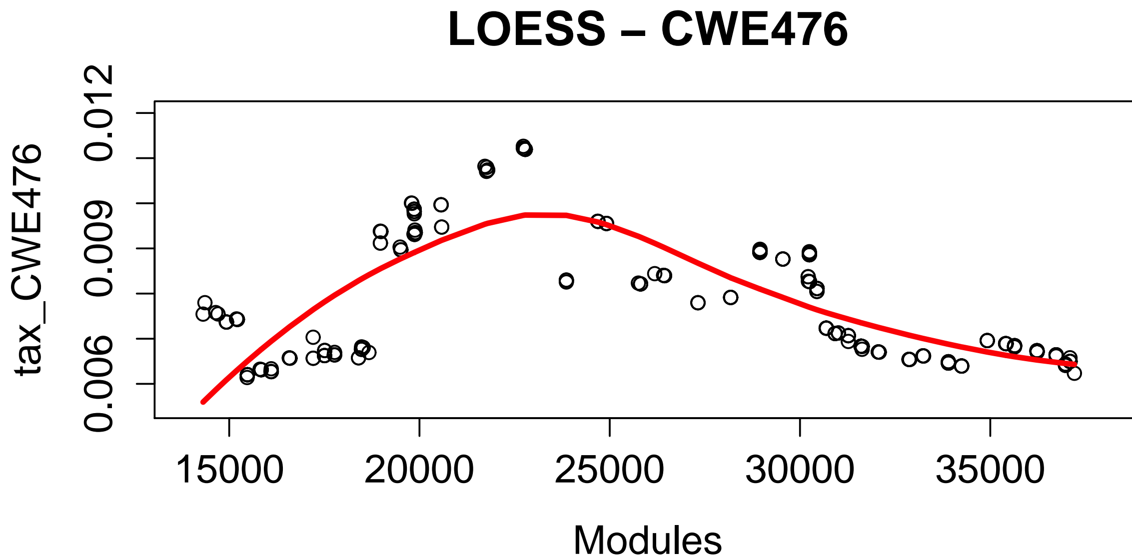


Figura 20 – Curva gerada pelo método *LOESS* sobre os dados da taxa da CWE476

dependente nesta regressão foi o número total de módulos. Geralmente o número de módulos cresce quanto maior a versão do software, até se atingir um ponto máximo, onde esse número começa a reduzir. A regressão para um polinômio quadrático do conjunto de treinamento nos deu a seguinte equação:

$$\begin{aligned} tax_CWE476(modules) = & \quad (-2.131399e^{-11}) * modules^2 \\ + & \quad (1.057282e^{-6}) * modules \\ - & \quad 0.004237619 \end{aligned}$$

Após realizada uma predição sobre o conjunto de teste, foi plotada a curva de predição juntamente com o *scatterplot* dos dados, como pode ser visto na Figura 21. O erro residual padrão para esse modelo foi 0.0009653.

A regressão para um polinômio cúbico do conjunto de treinamento nos deu a seguinte equação:

$$\begin{aligned} tax_CWE476(modules) = & \quad (1.911224e^{-15}) * modules^3 \\ - & \quad (1.72028e^{-10}) * modules^2 \\ + & \quad (4.857479e^{-6}) * modules \\ - & \quad 0.03460173 \end{aligned}$$

Após realizada uma predição sobre o conjunto de teste, foi plotada a curva de

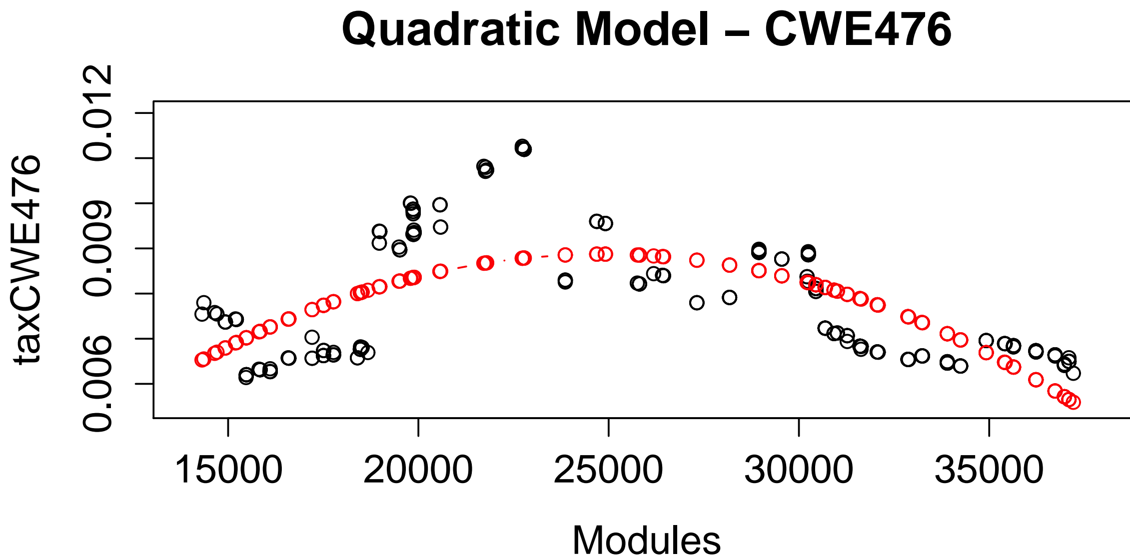


Figura 21 – Curva de predição do modelo quadrático dos dados da CWE476

predição juntamente com o *scatterplot* dos dados, como pode ser visto na Figura 22. O erro residual padrão para esse modelo foi 0.00084.

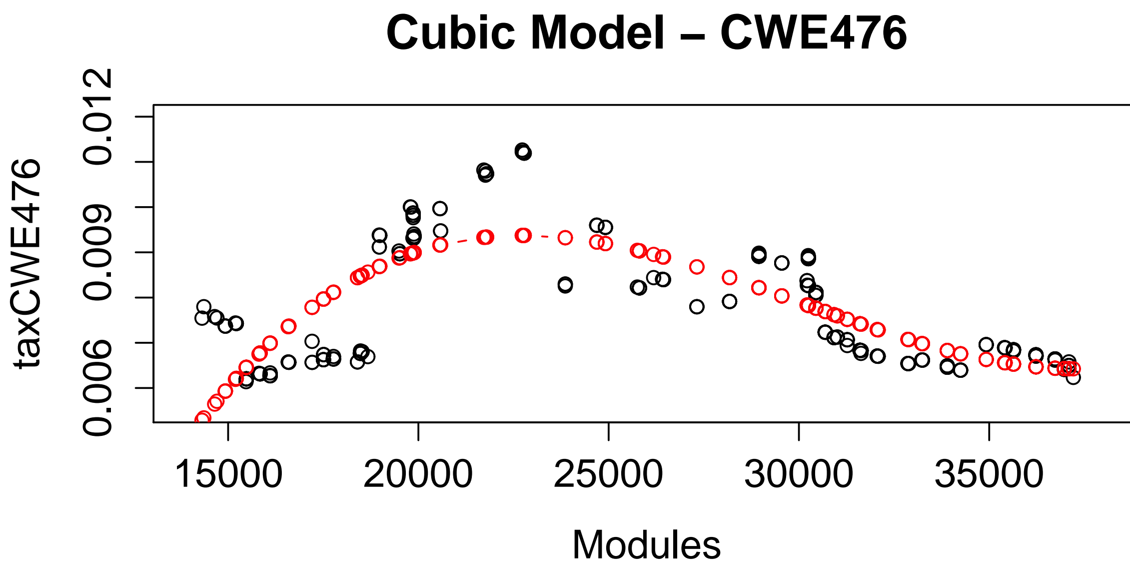


Figura 22 – Curva de predição do modelo cúbico dos dados da CWE476

6.2 Definição de modelo para CWE457 - Variável não Inicializada

Na busca por possíveis *outliers* foi contruído o gráfico *boxplot* que pode ser visto na Figura 23. No caso da CWE457 foram encontrados alguns *outliers*, como pode set visto

no gráfico, alguns pontos indo além do limite do *boxplot*. Foram removidos 19 pontos do conjunto de dados, não se encontrou uma justificativa plausível para a ocorrência desses *outliers*, essa decisão foi tomada apenas com o apoio do método estatístico.

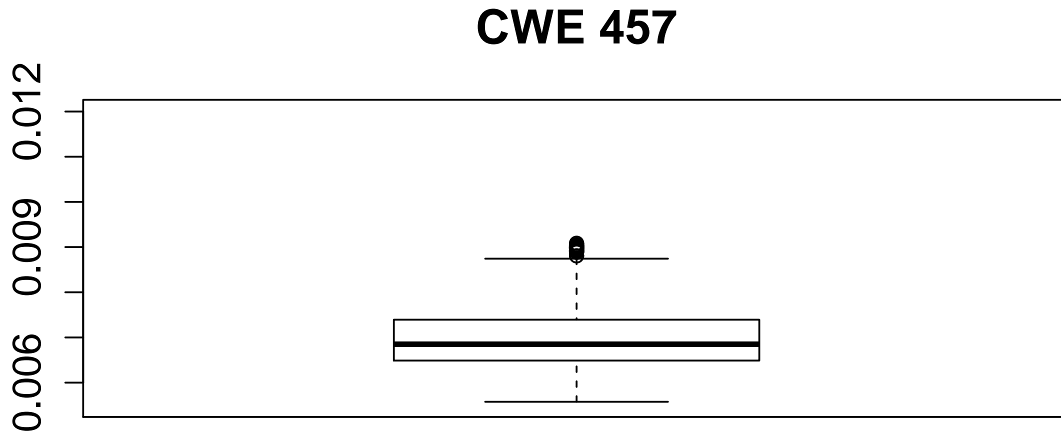


Figura 23 – *Boxplot* da taxa da CWE457 por módulo

Após a remoção dos *outliers* identificados, foi utilizado o método *LOESS* para a geração da curva de referência. Na Figura 24 está plotado a curva de predição do modelo gerado sobre o *scatterplot* dos dados das taxas da CWE457 por módulo. O erro residual padrão desse modelo é 0.0003126, sendo o erro residual padrão o desvio padrão da diferença entre o valor real e o predito.

Mais uma vez o modelo não paramétrico utilizado nos traz uma curva bastante suave que se adapta bem aos dados trabalhados. Apesar de aparentar ser um bom modelo, não é facilmente visível qual o grau do polinômio que melhor se assemelha a curva gerada. Com isso, se tomou como base o trabalho realizado sobre a CWE476, onde foram testados um modelo quadrático e um cúbico, que, provavelmente, vão conseguir se ajustar a curva de referência, já que a mesma não possuí muitas nuâncias.

O conjunto de dados obtidos a partir da análise do projeto *Linux Kernel* foi dividido entre conjunto de treinamento (65 % dos dados) e teste (35% dos dados). A variável dependente nesta regressão foi o número total de módulos. Geralmente o número de módulos cresce quanto maior a versão do software, até se atingir um ponto máximo, onde esse número começa a reduzir. A regressão para um polinômio quadrático do conjunto de treinamento nos deu a seguinte equação:

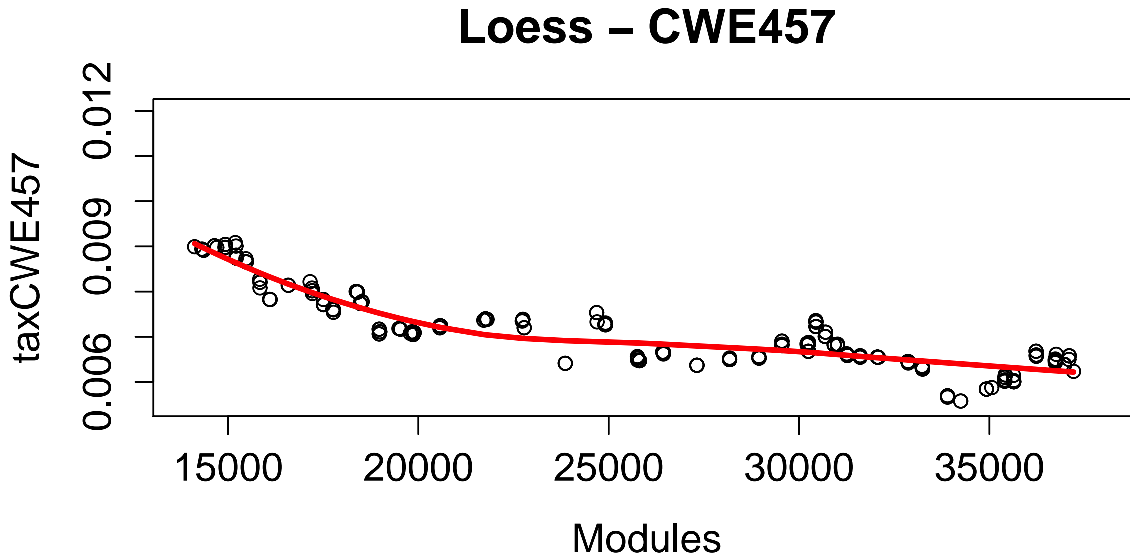


Figura 24 – Curva gerada pelo método *LOESS* sobre os dados da taxa da CWE457

$$
\begin{aligned}
tax_CWE457(modules) = \quad & (5.440273e^{-12}) * modules^2 \\
- \quad & (3.745102ei^{-07}) * modules \\
+ \quad & 0.01283622
\end{aligned}
$$

Após realizada uma predição sobre o conjunto de teste, foi plotada a curva de predição juntamente com o *scatterplot* dos dados, como pode ser visto na Figura 25. O erro residual padrão para esse modelo foi 0.0003629.

A regressão para um polinômio cúbico do conjunto de treinamento nos deu a seguinte equação:

$$
\begin{aligned}
tax_CWE476(modules) = \quad & (-6.466983e^{-16}) * modules^3 \\
+ \quad & (5.603787e^{-11}) * modules^2 \\
- \quad & (1.639652e^{-6}) * modules \\
+ \quad & 0.02287291
\end{aligned}
$$

Após realizada uma predição sobre o conjunto de teste, foi plotada a curva de predição juntamente com o *scatterplot* dos dados, como pode ser visto na Figura 26. O erro residual padrão para esse modelo foi 0.0003211.

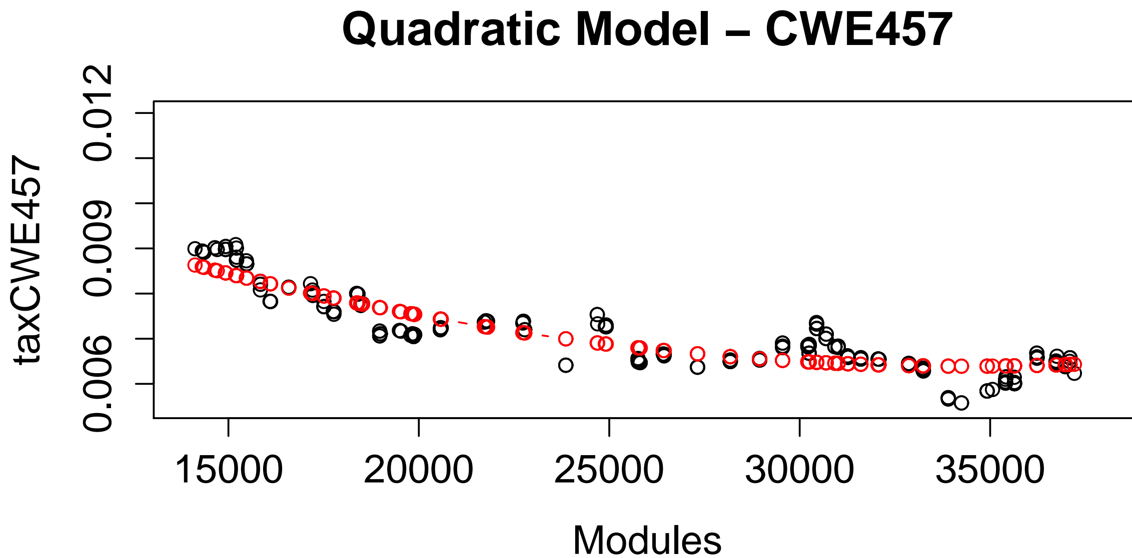


Figura 25 – Curva de predição do modelo quadrático dos dados da CWE457

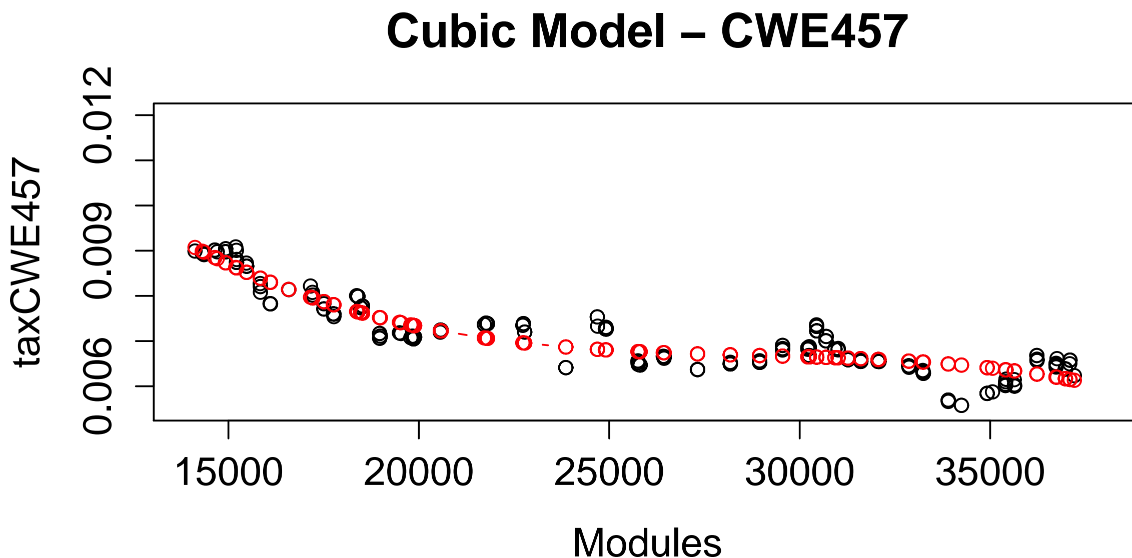


Figura 26 – Curva de predição do modelo cúbico dos dados da CWE457

6.3 Comparação e Escolha dos Modelos

Neste capítulo foram definidos dois modelos para cada uma das métricas de ameaças de vulnerabilidade de código-fonte, nesta seção esses modelos serão comparados, baseado no erro na predição do modelo e no método *K-fold* de validação cruzada, e então serão selecionados os melhores modelos. Nesta seção foi feita uma comparação e escolha entre os modelos desenvolvidos, podendo existir modelos melhores para cada uma das métricas.

Pensou-se em realizar uma Análise de Variância (*ANOVA*) para tentar auxiliar na comparação entre os modelos, mas não foi possível já que o conjunto de dados das taxas das CWE476 e CWE457 por módulo não respeitam as suposições a cerca da *ANOVA* apresentadas na Seção 4.5, como pode ser visto durante a análise exploratória dos dados nas seções 5.3.2.2 e 5.3.2.1.

Observa-se que quanto maior o grau do polinômio melhor serão os resultados obtidos, entretanto, o objetivo deste trabalho é encontrar um modelo que não seja muito custoso na realização de predições, onde o mesmo possa ser inserido sem muitos problemas no ciclo de desenvolvimento de software.

Tendo isso em vista, foram analisados e comparados os modelos de cada uma das métricas separadamente, como pode ser visto a seguir.

6.3.1 CWE476 - Referência de Ponteiros Nulos

Para iniciar a comparação foram confrontados todos os modelos em um único gráfico (Figura 27), que pode nos dar uma visão geral de como os modelos estão realizando as suas respectivas predições.

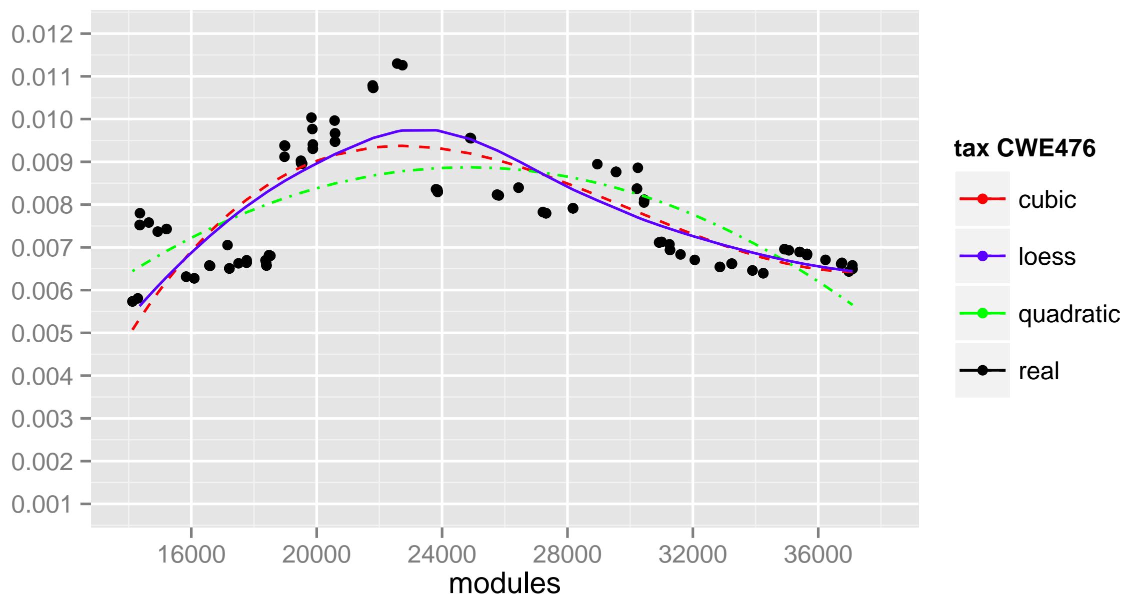


Figura 27 – Curva de predição dos modelos dos dados da CWE476

A Figura 27 nos mostra de maneira visual que o modelo cúbico gerado é o que melhor se aproxima da curva de predição do modelo de referência (modelo *LOESS*, não paramétrico), principalmente no que diz respeito a extrapolação, tanto para dados menores ou maiores ao intervalo de dados trabalhados nesta pesquisa. Essa extrapolação é importante para que seja possível a predição da métrica, por exemplo, para projetos com mais de 40.000 ou com menos de 14.000 módulos. O modelo cúbico acompanha de maneira

mais adequada nos limites do gráfico o modelo de referência, e o modelo quadrático tende a se distanciar nesses locais.

Saindo da parte visual e partindo para o erro residual padrão, sendo o erro residual padrão o desvio padrão da diferença entre o valor real e o predito, gerado por cada um dos modelos, que podem ser vistos na Tabela 4. Percebe-se que o erro referente ao modelo cúbico se aproxima bastante ao nosso modelo *LOESS*. O erro associado ao modelo quadrático é aproximadamente 20,00% maior do que o do modelo de referência, já o cúbico é aproximandamente 4,43% maior. Logo, o erro entre os modelos quadrático e cúbico é de aproximadamente 15%, sendo o cúbico mais próximo do desejável.

Modelo	Erro residual padrão
LOESS	0.0008044
Quadrático	0.0009653
Cúbico	0.0008400

Tabela 4 – Resumo do ERP dos modelos contruídos para as taxas da CWE476

Para finalizar a comparação entre os modelos foi realizado uma validação cruzada utilizando o método *K-fold* para analisar a performance de ambos o modelos em um conjunto de dados em que não foram treinados, ou seja, contra dados que o mesmo deveria predizer em uma situação real. Nesta pesquisa foi utilizado um $K = 10$, já que segundo Kohavi (1995) uma validação cruzada *ten-fold* pode ser mais eficiente até que uma validação cruzada *leave-one-out*, mais detalhes sobre o assunto pode ser visto na Seção 4.5. Os gráficos apresentados nas figuras 28 e 29 nos mostra um resumo do método aplicado.

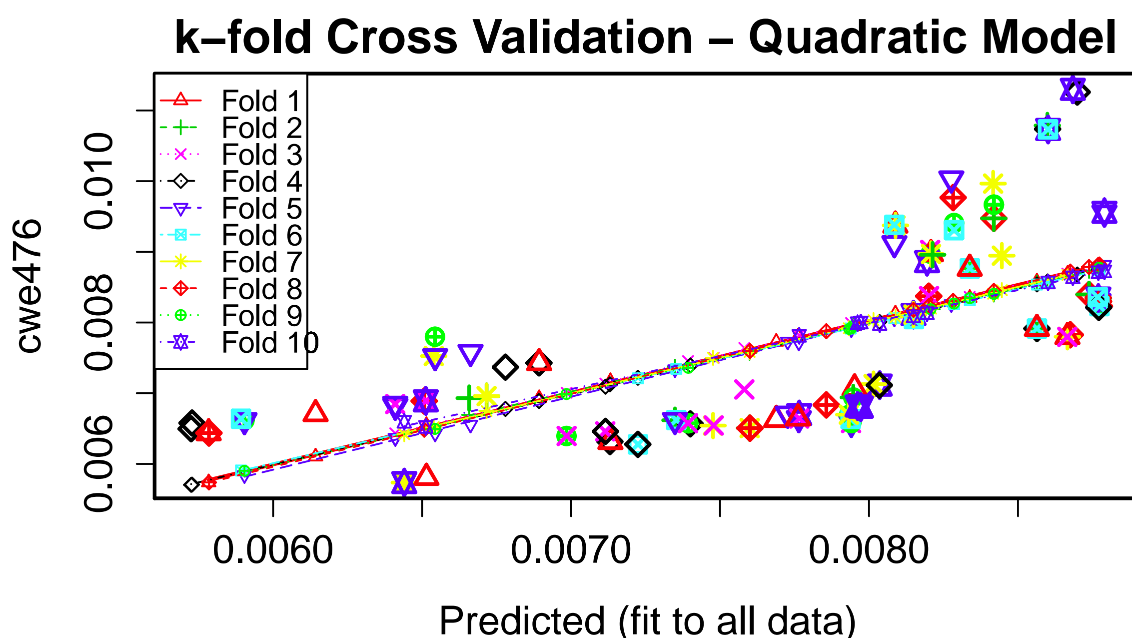


Figura 28 – Validação cruzada *Ten-fold* do modelo quadrático da taxa da CWE476

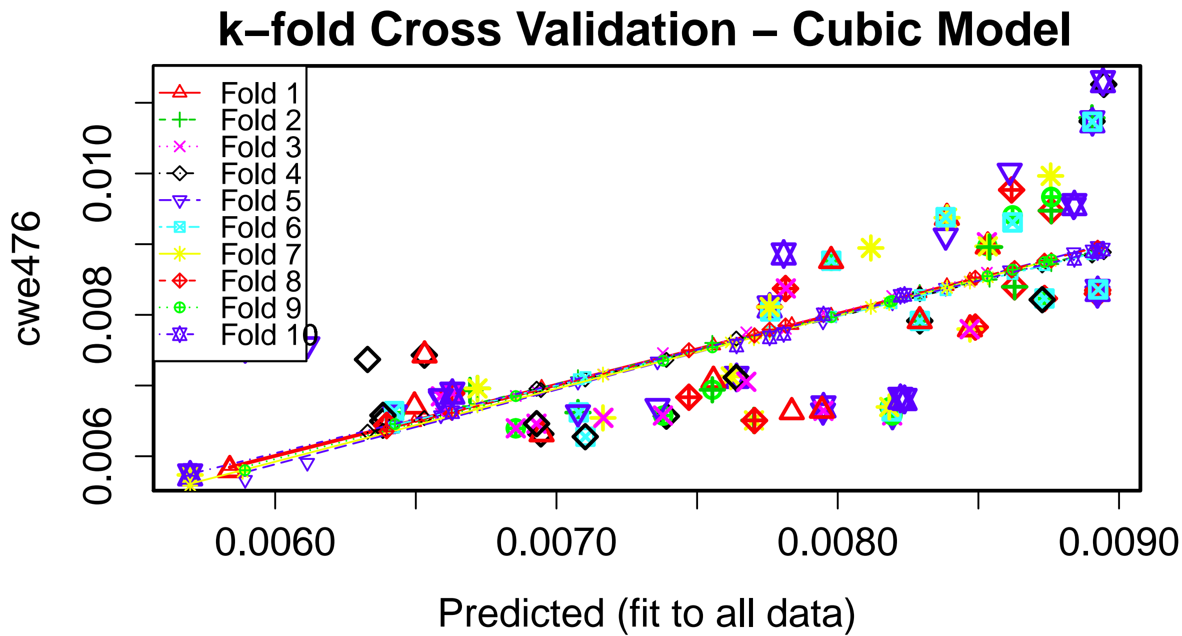


Figura 29 – Validação cruzada *Ten-fold* do modelo cúbico da taxa da CWE476

O gráfico apresentado possuem no eixo X os valores preditos pelo modelo em questão e no eixo Y o valor real. As retas traçadas representam o que seria o ideal, ou seja, o valor predito igual ao valor real, e cada um dos diferentes pontos apresentam o que foi gerado pelo modelo em cada um dos *folds*. Pode-se ver que em ambos os gráficos vários pontos ficaram distantes da reta, mostrando que existe um erro associado a predição. Na Tabela 5 são apresentados os erros médios quadráticos de cada um dos modelos aqui avaliados sobre o conjunto de teste. A definição de erro médio quadrático pode ser encontrada na Seção 4.5.

Modelo	Erro médio quadrático
Quadrático	0.000000958
Cúbico	0.000000862

Tabela 5 – EMQ dos modelos das taxas da CWE476 através da validação cruzada.

Pode-se ver que mais uma vez o modelo cúbico se sobressaiu ao modelo quadrático, sendo o erro médio quadrático associado ao modelo quadrático aproximadamente 11,14% maior do que o modelo cúbico em situações reais de análise.

Levando em consideração todos os aspectos trabalhados na comparação dos dois modelos, o modelo cúbico se apresentou como um melhor modelo para o acompanhamento, monitoramento e predição da taxa da ameaça de vulnerabilidade de código-fonte relacionada a referência de ponteiros nulos.

6.3.2 CWE457 - Variável não Inicializada

Para uma melhor visualização dos modelos construídos foi elaborado um gráfico contendo a curva de predição de todos eles sobre o *scatterplot* dos dados. Podendo o mesmo ser visto na Figura 30.

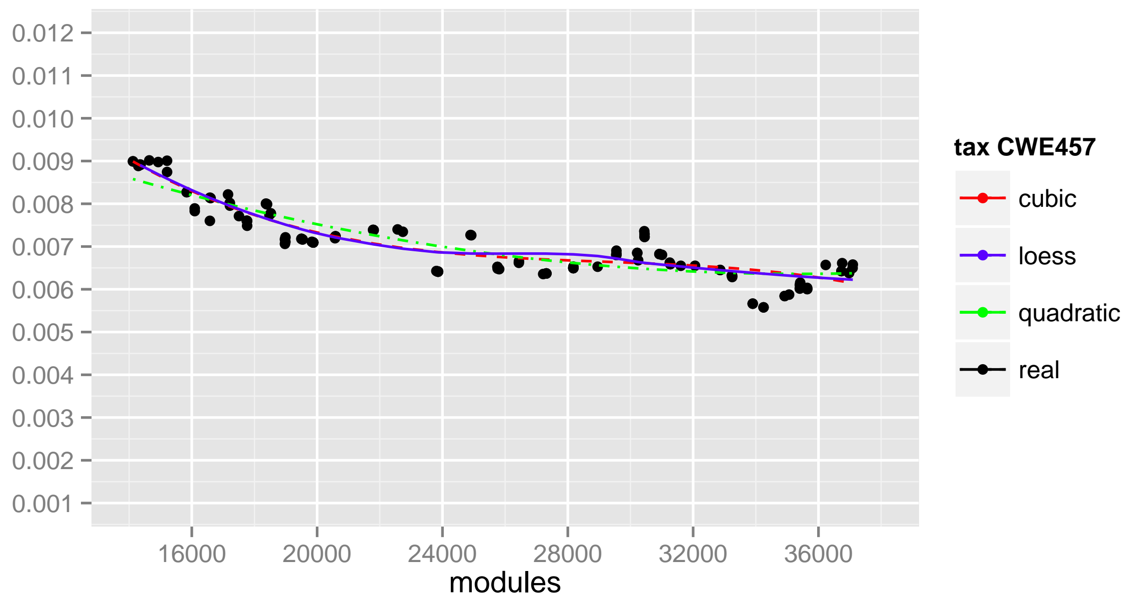


Figura 30 – Curva de predição dos modelos dos dados da CWE457

Todos os modelos apresentados na Figura 30 ficaram bem próximos uns dos outros, diferente do ocorrido com os modelos referentes a CWE476. Entretanto, o modelo cúbico continua mais próximo do modelo *LOESS* nos limites mínimo e máximo em relação ao modelo quadrático, sendo similar a comparação dos modelos na Seção 6.3.1, o que favorece predições de valores fora do intervalo de valores trabalhados neste estudo.

Visualmente também pode-se ver que provavelmente o erro residual desses modelos são bem menores do que os análisados na Seção 6.3.1. Na Tabela 6 são apresentados os erros residuais padrão de cada um dos modelos, mais sobre erro residual padrão pode ser visto na Seção 4.5.

Modelo	Erro residual padrão
LOESS	0.0003287
Quadrático	0.0003674
Cúbico	0.0003388

Tabela 6 – Resumo do ERP dos modelos contruídos para as taxas da CWE457

Como foi dito anteriormente, os erros residuais padrão dos modelos apresentados possuíram valores baixos e com uma pequena diferença entre os mesmos, apesar do modelo cúbico estar mais próximo do modelo de referência. O modelo quadrático teve um erro

11,77% maior do que o modelo de referência e o modelo cúbico 3,07% maior, logo, o modelo quadrático teve um erro 8,44% maior do que o modelo cúbico. Mais uma vez o modelo cúbico apresentado um erro menor do que o modelo quadrático.

Para verificar o desempenho de cada um dos modelos foi feita uma validação cruzada *K-fold*. Assim como na Seção 6.3.1, foi utilizado um $K = 10$. Os gráficos apresentados nas figuras 31 e 32 nos mostra um resumo dos testes realizados.

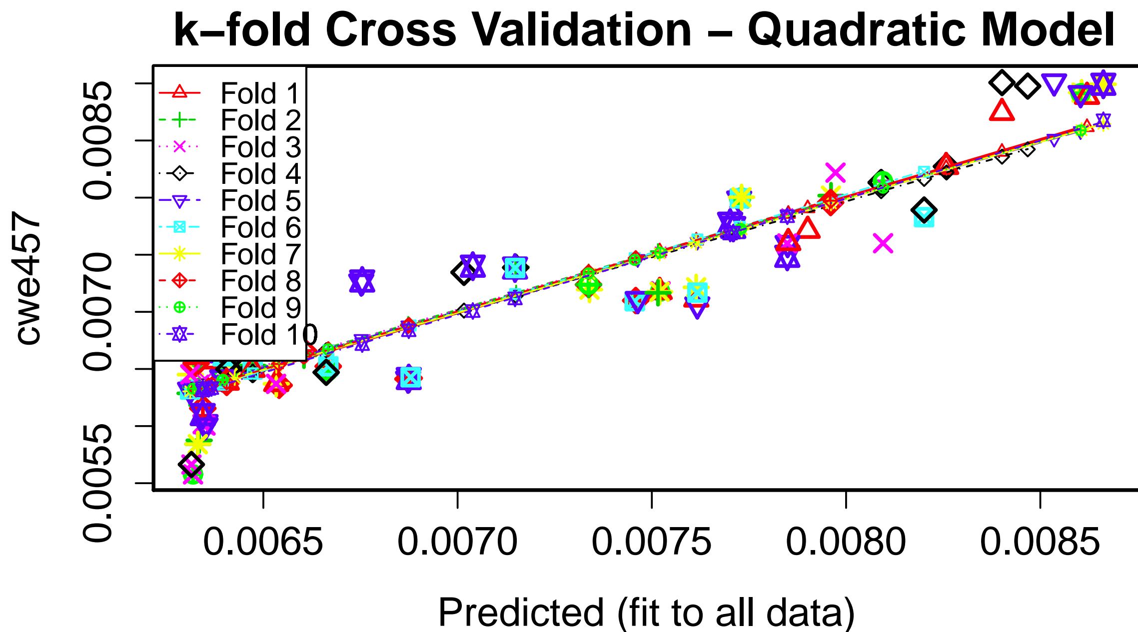


Figura 31 – Validação cruzada *Ten-fold* do modelo quadrático da taxa da CWE457

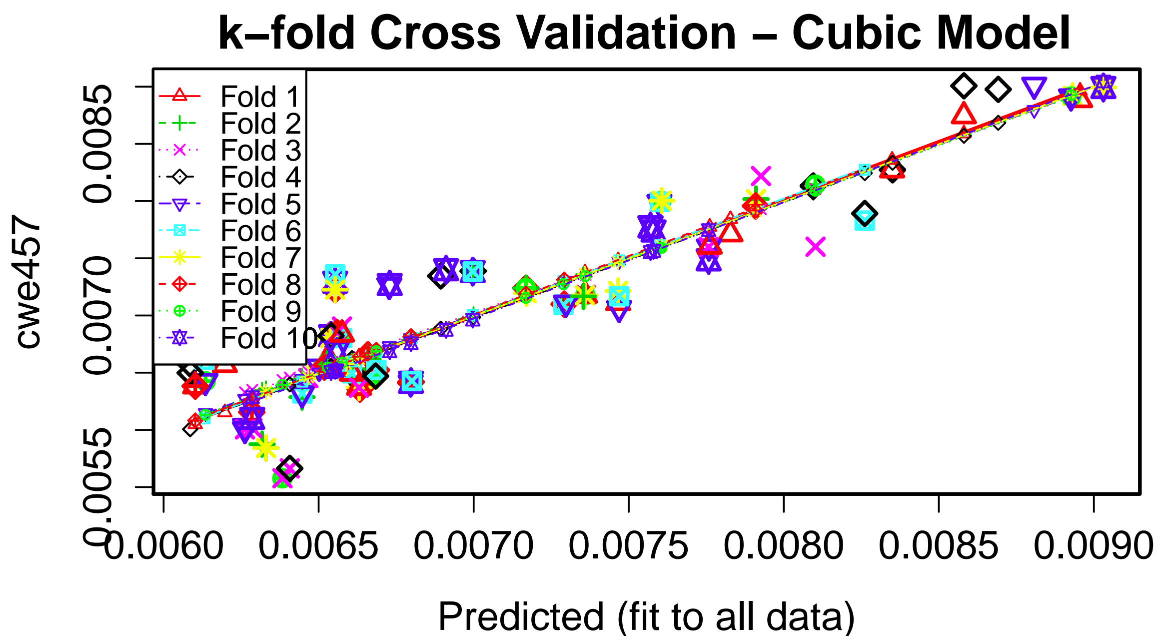


Figura 32 – Validação cruzada *Ten-fold* do modelo cúbico da taxa da CWE457

Definições acerca do método *K-fold* de validação cruzada podem ser encontradas na

Seção 4.5. A Tabela 7 apresenta os erros médios quadráticos obtidos através da validação cruzada.

Modelo	Erro médio quadrático
Quadrático	0.000000137
Cúbico	0.000000112

Tabela 7 – EMQ dos modelos das taxas da CWE457 através da validação cruzada.

O erro médio quadrático referente ao modelo cúbico se apresentou 22,32% menor do que do modelo quadrático. Apesar da diferença percentual aparentar ser grande, numericamente pode não ser tão significativo, essa diferença percentual é devido ao erro ser bem pequeno, onde qualquer pequena diferença representa um grande percentual.

Apesar do modelo cúbico se apresentar mais vantajoso durante a comparação realizada, o modelo quadrático também se mostrou como uma boa solução para o acompanhamento, monitoramento e predição desta métrica de ameaça de vulnerabilidade de código-fonte, já que a diferença no geral foi pequena. Logo, a seleção de qualquer um dos modelos seria uma boa escolha, o modelo quadrático tendo uma complexidade menor, e o modelo cúbico acompanhando de maneira mais adequado o modelo de referência nos limites apresentados, o que favorece a extrapolação dos valores. Tendo em vista que este trabalho visa principalmente a predição dos valores das métricas em questão, o modelo cúbico foi selecionado devido a maior aderência ao modelo de referência escolhido.

6.4 Consolidação dos Resultados

Com o que se diz respeito a hipótese *H3*, foram definidos dois modelos e selecionado um para cada uma das métricas de ameaças de vulnerabilidade de código-fonte. Essa hipótese também não foi negada, já que como foi apresentado na comparação entre os modelos (Seção 6.3), os mesmos se apresentaram bem diante de situações reais de análise, logo, atingindo o objetivo de acompanhar, monitorar e predizer valores das métricas de maneira satisfatória. Entretanto, como foi explicado neste capítulo, quanto maior o grau do polinômio melhor ele se adaptará aos dados e menor será o erro associado ao conjunto de treinamento.

Após a realização da comparação e seleção dos modelos para cada uma das métricas de ameças de vulnerabilidade de código-fonte, chegou-se a seguinte função para cada uma das métricas trabalhadas:

$$\begin{aligned} tax_CWE476(modules) = & \quad (1.911224e^{-15}) * modules^3 \\ - & \quad (1.72028e^{-10}) * modules^2 \\ + & \quad (4.857479e^{-6}) * modules \\ - & \quad 0.03460173 \end{aligned}$$

$$\begin{aligned} tax_CWE457(modules) = & \quad (-6.466983e^{-16}) * modules^3 \\ + & \quad (5.603787e^{-11}) * modules^2 \\ - & \quad (1.639652e^{-6}) * modules \\ + & \quad 0.02287291 \end{aligned}$$

Lembrando que as fórmulas matemáticas apresentadas acima representam os modelos desenvolvidos no início do capítulo, ambos sendo desenvolvidos a partir de uma regressão polinomial de terceiro grau das métricas obtidas a partir do código-fonte do projeto *Linux Kernel*.

Para apresentar os modelos definidos foram elaboradas as Tabelas 8 e 9 que apresentam os valores das referidas métricas em alguns determinados pontos já conhecidos, sendo eles a primeira versão, a última versão e a versão com o valor da métrica mais alto, e ao final um novo ponto predito pelo modelo construído.

Version	tax_CWE476	Modules
linux-v2.6.11	0.005735325	14123
linux-v2.6.39	0.008927095	30245
linux-v3.9-rc8	0.006460368	33899
X	0.008715918	43787

Tabela 8 – Alguns valores da taxa da CWE476 mais predição realizada.

Version	tax_CWE457	Modules
linux-v2.6.11	0.008992424	14123
linux-v3.16-rc3	0.006577706	37095
linux-v3.9-rc8	0.005663884	33899
X	0.004226781	43787

Tabela 9 – Alguns valores da taxa da CWE457 mais predição realizada.

Para selecionar o ponto que foi predito foi feita a média da diferença entre os módulos dos outros pontos selecionados, que coincidentemente foi o mesmo para ambas as métricas, apesar dos pontos conhecidos selecionados serem diferentes. Como pode-se

ver na Tabela 8, a métrica relacionada a taxa da CWE476 teve um pico na versão 2.6.39 e depois começou a diminuir, mas a partir do momento que o número de módulos voltar a crescer ela tende a crescer novamente, e em determinado ponto deve atingir um novo pico e voltar a diminuir o seu valor. Com a métrica relacionada a taxa da CWE457 já é diferente, pelo o que foi apresentado na Tabela 9 a métrica tende a diminuir sempre com o decorrer do aumento do número de módulos, sendo o número de módulos a variável independente.

Para exemplificar o uso dos modelos desenvolvidos serão apresentados na Seção 6.5 alguns exemplos de uso do mesmo com alguns dos projetos de software trabalhados na primeira etapa da pesquisa. Lembrando, que esses modelos baseados no projeto *Linux Kernel* visam servir de referência para outros projetos.

6.5 Exemplos de Uso

Para dar alguns exemplos palpáveis do uso dos modelos desenvolvidos em um projeto de softwares livre foram realizadas predições das métricas das taxas das CWE476 e CWE457 em projetos de software livre. Para isso foram selecionados dois projetos, sendo eles o *FreeBSD* e o *Android*, ambos inclusive tendo em comum com o *Linux Kernel* o desenvolvimento de um núcleo de um sistema operacional. Foi desenvolvida a Tabela 10, para tentarmos verificar o uso dos modelos de predição em projetos de software livre.

Projeto de Software	Módulos	tax_CWE476	tax_CWE457
FreeBSD	33203	0.0069897180	0.0065379640
Android	49431	0.0160103100	0.0006387576

Tabela 10 – Predição de métricas em projetos de software livre maiores.

Pode-ser ver na Tabela 10 que ambos os modelos se comportaram de maneira adequada e dentro do esperado, com valores de métricas aceitáveis, para projetos de software com a quantidade de número de módulos igual ou maior ao que os mesmos foram construídos baseado no projeto *Linux Kernel*. O projeto *FreeBSD* se apresenta em um estágio similar ao *Linux Kernel*, onde os mesmos possuem um número de módulos e valores de ambas as métricas similares quando se comparando os dados das Tabelas 10, 8 e 9, levando em consideração a última versão do *Linux Kernel*. Já o projeto *Android* se apresenta com uma quantidade de módulos muito maior do que os outros dois projetos mencionados, a primeira vista aparenta que talvez o projeto esteja em um estágio mais imaturo e necessite de uma refatoração para atingir o patamar dos outros, entretanto, essa diferença é grande devido ao repositório do projeto *Android* não conter apenas o núcleo do seu sistema operacional, mas sim vários outras coisas como ABI (*Application Binary Interface*), *frameworks*, suas próprias ferramentas de desenvolvimento entre outras coisas.

Portanto, com os modelos desenvolvidos foi possível realizar a predição das métricas em questão para projetos do mesmo porte da referência selecionada (*Linux Kernel*), sendo esses a principal contribuição desta pesquisa. Os resultados obtidos demostraram que foi possível realizar as predições das métricas de ameaças de vulnerabilidade de código-fonte propostas, sendo esse o objetivo maior deste trabalho.

6.6 Evolução dos Modelos de Predição

Tendo em vista os modelos definidos e comparados no decorrer deste capítulo, explorou-se uma outra abordagem, onde serão apresentadas curvas de erro de predição para modelos não lineares de diferentes graus. A partir da mesma base de dados utilizada anteriormente, foram definidos modelos de segundo até vigésimo grau. Tendo esses modelos, foi realizado o processo de validação cruzada *K-fold*, com $K = 10$. Com o erro de predição obtido através da validação cruzada foi plotado um gráfico para cada uma das métricas de ameaças de vulnerabilidade de código fonte trabalhadas neste capítulo. Sendo o eixo X o grau do modelo em questão, e o eixo Y o erro de predição do modelo. Como pode ser visto nas Figuras 33 e 34.

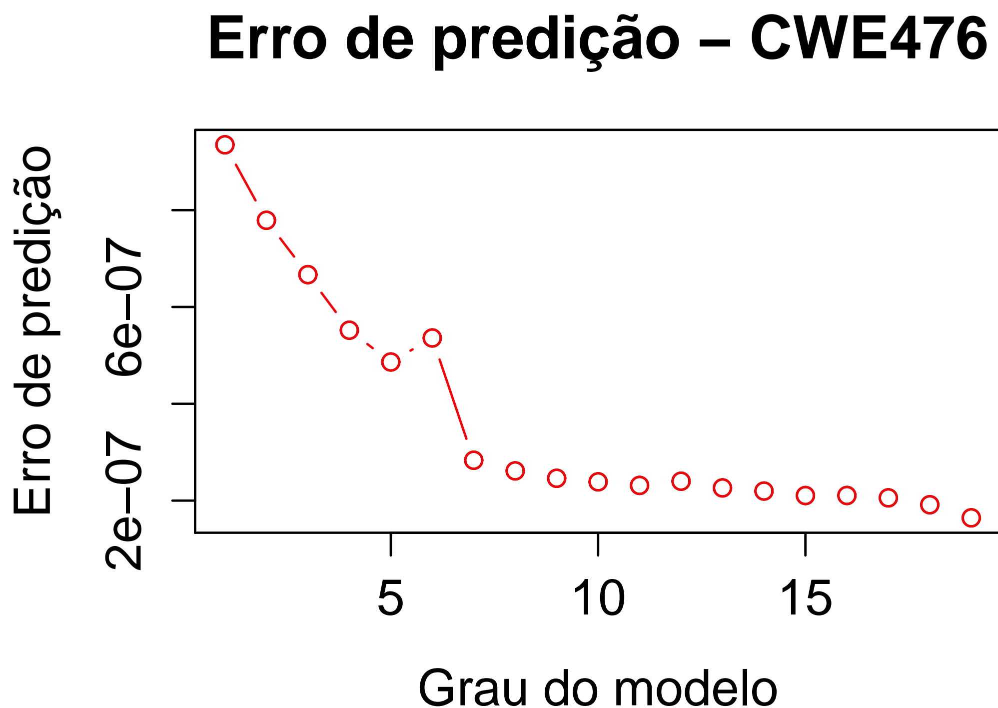


Figura 33 – Erro de predição de modelos não lineares para a CWE476

Como pode-se ver nas Figuras 33 e 34, e como foi discutido anteriormente, quanto maior o grau do modelo, menor será o erro de predição na maioria dos casos. Esta abor-

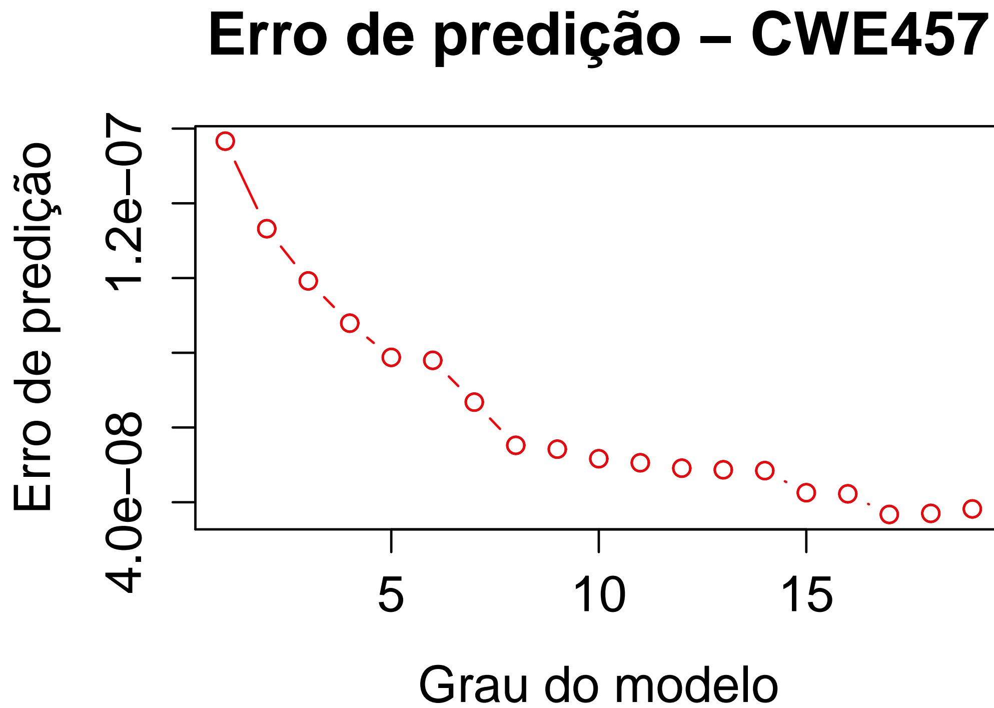


Figura 34 – Erro de predição de modelos não lineares para a CWE457

dagem pode auxiliar na seleção de um modelo não linear para cada uma das métricas, podendo ser uma continuidade do trabalho apresentado durante este capítulo. Pode-se considerar o custo benefício da diminuição do erro de predição do modelo com a complexidade do mesmo, o que deve aumentar o tempo de predição. Entretanto, a validação desta abordagem não faz parte do escopo deste trabalho.

7 Conclusão

Na primeira etapa deste trabalho foi realizado um estudo qualitativo das métricas de ameaças de vulnerabilidade de código-fonte, onde foram analisados um total de dez projetos de software livre, a fim de entender o comportamento das métricas em questão. Ao final, pode-se definir um subconjunto de métricas mais frequentes e responder algumas das hipóteses apresentadas na Seção 1.2. Na realização desse estudo foi possível identificar métricas de ameaças de vulnerabilidade mais recorrentes, sendo elas: referência a ponteiros nulos, variáveis não inicializadas e vazamento de memória, um resumo das métricas calculadas de cada um dos projetos pode ser visto no Apêndice B. Além disso, foi respondida a hipótese levantada na primeira parte deste trabalho, sendo essa a hipótese *H1*. A hipótese *H1* afirma que as métricas de ameaças de vulnerabilidade de código-fonte podem ser acompanhadas da mesma forma que as métricas de *design*. Após uma análise dos dados das métricas obtidos pode-se perceber que boa parte dos valores das métricas de ameaças de vulnerabilidades são nulos, como pode-se ver no Apêndice A, diferente das métricas de *design*, o que altera a forma de acompanhamento das mesmas, isso é explicado até pela natureza distinta das diferentes classes de métricas. Logo, pode-se negar a hipótese *H1*.

Na segunda etapa deste trabalho foi feito um trabalho estatístico para determinar modelos de predição para as métricas de ameaças de vulnerabilidade de código-fonte mais recorrentes. Para isso, foi usada uma abordagem de análise exploratória de dados, onde nela se definiu o conjunto de dados referentes as métricas que foram trabalhados, nesta etapa foi decidido a não definição de um modelo de predição para a métrica relacionada a vazamento de memória (CWE401) como foi explicado na Seção 5.3.2. Após a análise exploratória dos dados ter sido realizada não se pode negar a hipótese *H2*, que afirma que os valores das métricas de ameaças de vunerabilidade se comportam como uma distribuição estatística de cauda longa, não sendo similar a uma distribuição normal, a análise realizada pode ser vista na Seção 5.3.2. Com as informações obtidas através da análise pode-se realizar a definição de modelos de predição para as métricas de ameaças de vulnerabilidade de código-fonte relacionadas a referência de ponteiros nulos e variáveis não inicializadas (CWE476 e CWE457), no Capítulo 6 pode-se ver o processo de definição, validação e seleção dos modelos desenvolvidos, que apresentaram resultados satisfatórios apesar de possuírem uma baixa complexidade (polinômio cúbico), como pode-se ver nos exemplos de uso apresentados na Seção 6.5. E com isso também não foi possível negar a hipótese *H3*, onde foi afirmado que pode-se monitorar métricas de ameaças de vulnerabilidade através de um modelo baseado em uma função polinomial.

Tendo sido analisadas as três hipóteses inicialmente levantadas para responder a

questão-problema deste trabalho, sendo ela:

É possível o desenvolvimento de modelos preditivos de referência para acompanhamento e monitoramente de métricas de ameaças de vulnerabilidade de código-fonte em projetos de software?

Conclui-se que o desenvolvimento de modelos preditivos de referência, de baixa complexidade, para métricas de ameaças de vulnerabilidade de código-fonte é possível quando se possui um projeto de referência que possibilite a obtenção de uma base de dados considerável referente a valores das mesmas. Como foi discutido na Seção 6.4, quanto mais complexo for o polinômio referente ao modelo, melhor o mesmo se adaptará aos dados reduzindo o erro dentro do grupo de treinamento, entretanto, como o objetivo dos modelos definidos neste trabalho é predizer esses valores para outros projetos de software, o mesmo deve ter uma certa flexibilidade, evitando o *overfitting*. Além disso, modelos menos complexos tendem a gastar uma menor quantidade de tempo para realizarem as suas predições, o que pode auxiliar a inserção do mesmo no ciclo de desenvolvimento de software, tornando o processo de obtenção de valores de referência dessas métricas mais rápido, facilitando o monitoramento das mesmas em projetos de software.

7.1 Limitações do Trabalho

Uma possível limitação deste trabalho foi a identificação de *outliers* dentro do conjunto de dados das métricas trabalhadas, apresentada na Capítulo 6. Nesse caso, um método simples e conhecido foi puramente aplicado para a identificação de *outliers*, não sendo encontrada nenhuma evidência de que os valores realmente representavam algum evento fora do normal.

A principal limitação deste trabalho foi a não validação dos valores de referência obtidos através dos modelos preditivos desenvolvidos com os reais valores das métricas em diferentes projetos de software. Na Seção 6.5 foram apresentados alguns valores de referência para alguns projetos de software livre, entretanto, os mesmos não foram validados com os reais valores das métricas que poderiam ser extraídos através de uma ferramenta de análise estática de segurança de código. Esse seria um bom teste para os modelos desenvolvidos.

7.2 Trabalhos Futuros

Como trabalho futuro deve-se validar os valores de referência preditos pelos modelos desenvolvidos com os valores das métricas extraídas através de análise estática e comparar o desempenho dos modelos. Para auxiliar no acompanhamento das métricas de

ameaças de vulnerabilidades de código-fonte em projetos menores, com menor número de módulos, é interessante replicar este trabalho tomando como referência um projeto desse porte.

Este trabalho também pode ser expandido a fim de definir modelos de predição para outras métricas de ameaças de vulnerabilidade de código-fonte, aumentando o abrangência desses modelos e facilitando a inserção dessa classe de métricas no ciclo de desenvolvimento de software. Outro trabalho interessante seria desenvolver modelos de predição de complexidade maior e comparar o seu desempenho com os modelos de baixa complexidade desenvolvidos neste trabalho, e verificar o custo benefício dos mesmos, podendo continuar a abordagem apresentada na Seção 6.6.

Outro trabalho interessante seria tentar definir modelos de predição de métricas de ameaças de vulnerabilidade baseado em métricas de *design*, ou seja, as entradas do modelo seriam métricas de *design* e como saída teriamos valores de referência para métricas de ameaças de vulnerabilidade. Isso pode ser vislumbrado já que, como foi apresentado no início deste trabalho, existem trabalhos que tentam relacionar essas diferentes classes de métricas.

Referências

ABBOTT, R. et al. Security analysis and enhancements of computer operating system. Technical Report NBSIR 761041, National Bureau of Standards, 1976. Citado na página 24.

ALSHAMMARI, B.; FIDGE, C.; CORNEY, D. Security metrics for object-oriented class designs. In: . Queensland University of Technology, Brisbane, Australia: [s.n.], 2009. Citado na página 20.

ARAúJO, T. C. Apprecommender: um recomendador de aplicativos gnu/linux. Instituto de Matemática e Estatística, USP, 2011. Citado na página 41.

ASLAM, T.; KRSUL, I. V.; SPAFFORD, E. H. Use of a taxonomy of security faults. Proceedings of the 19th National Information Systems Security Conference, 1996. Citado na página 24.

BLACK, P. E. Static analyzers in software engineering. National Institute of Standards and Technology, Gaithersburg, USA, 2001. Citado na página 36.

BRINK, H.; RICHARDS, J. Real world machine learning. Manning Publications C.O, 2014. Citado na página 38.

BROWNLEE, J. Discover feature engineering, how to engineer features and how to get good at it. <http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/>, 2014. Citado na página 38.

CAETANO, M. A. L. Métodos quantitativos. Insper, Ibmec, São Paulo, 2012. Citado na página 42.

CHESS, B.; WEST, J. Secure programming with static analysis. In: *Software Security Series*. [S.l.: s.n.], 2007. Citado 6 vezes nas páginas 9, 31, 32, 33, 34 e 35.

CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. In: . MIT, Cambridge, MA, USA: [s.n.], 2002. Citado na página 19.

CLEVELAND, W. S.; DEVLIN, S. J. Locally weighted regression: An approach to regression analysis by local fittting. Journal of the American Statistical Association, Vol. 83, No. 403., 1988. Citado na página 40.

ERNST, M. D. Static and dynamic analysis: synergy and duality. MIT Lab for Computer Science, Cambridge, MA 02139 USA, 2005. Citado na página 23.

ESPOSTE, A. M. D.; BEZERRA, C. F. L. Tomada de decisões orientadas a métricas de software: observações de métricas de produto e vulnerabilidades de software via dw e plataforma de monitoramente de código-fonte. In: . Universidade de Brasília, FGA, Brasília: [s.n.], 2014. Citado 3 vezes nas páginas 20, 23 e 25.

FENTON, N. E.; PFLEENGER, S. L. Software metrics: A rigorous and practical approach. In: *Course Technology*. [S.l.: s.n.], 1998. v. 2. Citado na página 19.

FERZUND, J.; AHSAN, S.; WOTAWA, F. Empirical evaluation of hunk metrics as bug predictors. Software Process and Product Measurement, International Conferences IWSM 2009 and Mensura 2009, 2009. Citado na página 23.

GRéGIO, A. R. A. et al. Um estudo sobre taxonomias de vulnerabilidades. LAC/INPE, CenPRA/MCT, Cert.BR, 2005. Citado 2 vezes nas páginas 24 e 25.

HAWKINS, D. M. Identification of outliers. In: . Londres, Chapman and Hall: [s.n.], 1980. Citado na página 37.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *IEEE 1061*: Ieee standard for a software quality metrics methodology. [S.l.], 1998. Citado na página 23.

JANSEN, W. Directions in security metrics research. In: . U.S. Department od Commerce, National Institute of Standards and Technology: [s.n.], 2009. Citado na página 20.

KOHAVI, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. International Joint Conference an Artificial Intelligence (IJCAI), Stanford, CA, USA, 1995. Citado na página 67.

KRSUL, I. V. Software vulnerability analysis. Ph.D. thesis, Purdue University, 1998. Citado na página 24.

LANDWEHR, C. et al. Taxonomy of computer program security flaws. ACM Computing Surveys, 1994. Citado na página 24.

MARTINS, M. E. G. Percentis. In: . WikiCiências: [s.n.], 2013. Citado na página 43.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. In: . São Paulo: [s.n.], 2013. Citado na página 43.

MISRA, S.; BHAVSAR, V. elationships between selected software measures and latent bug-density: Guidelines for improving quality. Computational Science and Its Applications-ICCSA2003, 2003. Citado na página 23.

NAGAPPAN, N.; BALL, T.; ZELLER, A. Mining metrics to predict component failures. Proceedings of the 28th international conference on Software engineering, ACM, New York, NY, USA, 2006. Citado na página 23.

OKUN, V. et al. Effect of static analysis tools on software security: Preliminary investigation. Alexandria, Virginia, USA, 2007. Citado na página 44.

RAYMOND, E. S. The cathedral and the bazaar. Linux Kongress, Würzburg, Germany, 1997. Citado 2 vezes nas páginas 46 e 47.

ROCHA, A. R. C. da; SOUZA, G. dos S.; BARCELLOS, M. P. Medição de software e controle estatístico de processos. In: . Ministério da Ciência, Tecnologia e Inovação; Secretaria de Política de Informática; Brasília: [s.n.], 2012. Citado na página 19.

ROSSMAN, A. J. Workshop statistics: Discovery with data. Springer-Verlag, New York, USA, 1996. Citado na página 38.

RUTAR, C. B. A. N.; FOSTER, J. S. A comparison of bug finding tools for java. EEE Int. Symp. on Software Reliability Eng. (ISSRE'04), France, 2004. Citado na página 44.

SEACORD, R. C.; HOUSEHOLDER, A. D. A structured approach to classifying security vulnerabilities. In: . CMU/SEI-2005-TN-003: [s.n.], 2005. Citado na página 24.

SNEDECOR, G. W.; COCHRAN, W. G. Statistical methods. In: . 6th edition.: [s.n.], 1967. Citado na página 41.

TUKEY, J. The future of data analysis. Princeton University, New Jersey, USA, 1961. Citado na página 37.

TUKEY, J. W. Exploratory data analysis. Adisson-Wesley, 1977. Citado 4 vezes nas páginas 39, 40, 59 e 60.

WASIAK, R. Exploratory data analysis. United BioSource Corporation, USA, 2012. Citado 2 vezes nas páginas 9 e 37.

ZHENG, J. et al. On the value of static analysis for fault detection in software. IEEE Trans. on Software Engineering, 2006. Citado na página 44.

Apêndices

APÊNDICE A – Análise dos Percentis

Neste apêndice contém os gráficos relacionados com a análise de percentis de algumas métricas de ameaças de vulnerabilidades extraídas do projeto *Linux Kernel*. Nos gráficos abaixo pode-se perceber que a maioria das métricas possuem valor constante igual a zero.

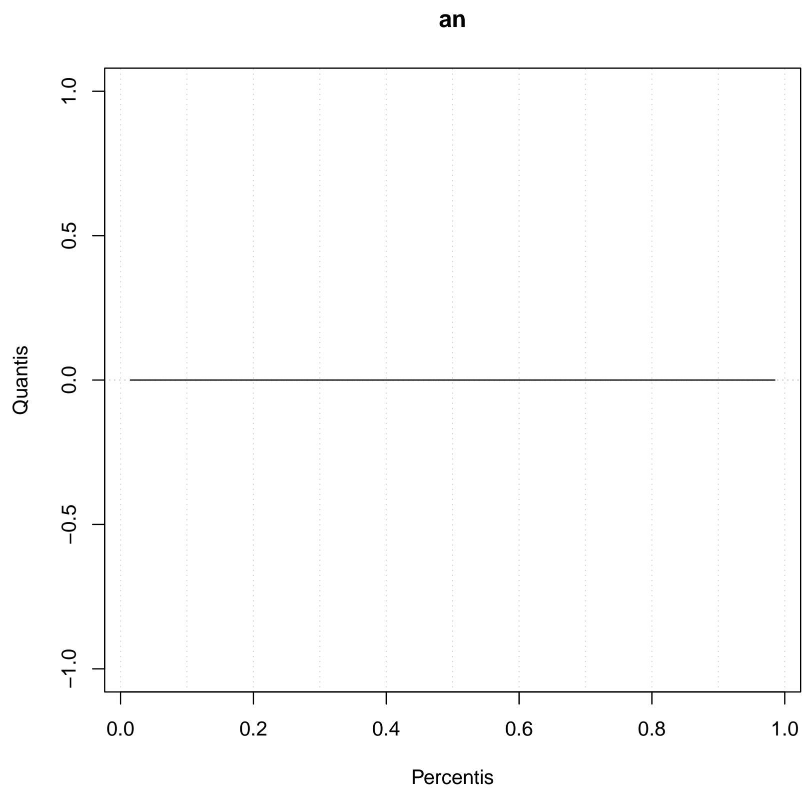


Figura 35 – Gráfico de Percentis da métrica AN

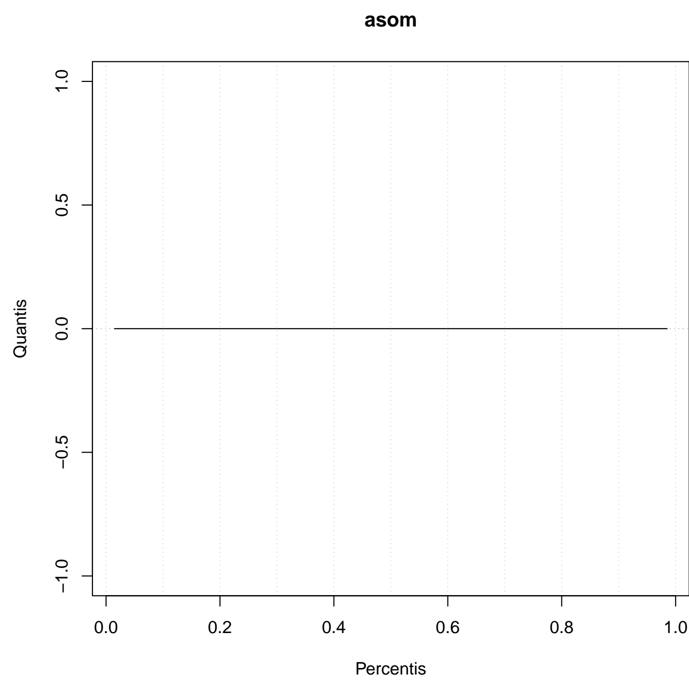


Figura 36 – Gráfico de Percentis da métrica ASOM

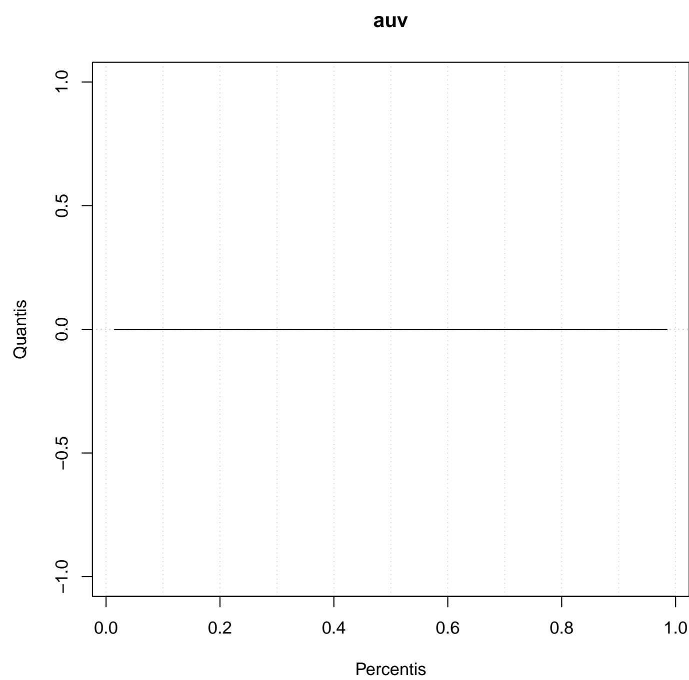


Figura 37 – Gráfico de Percentis da métrica AUV

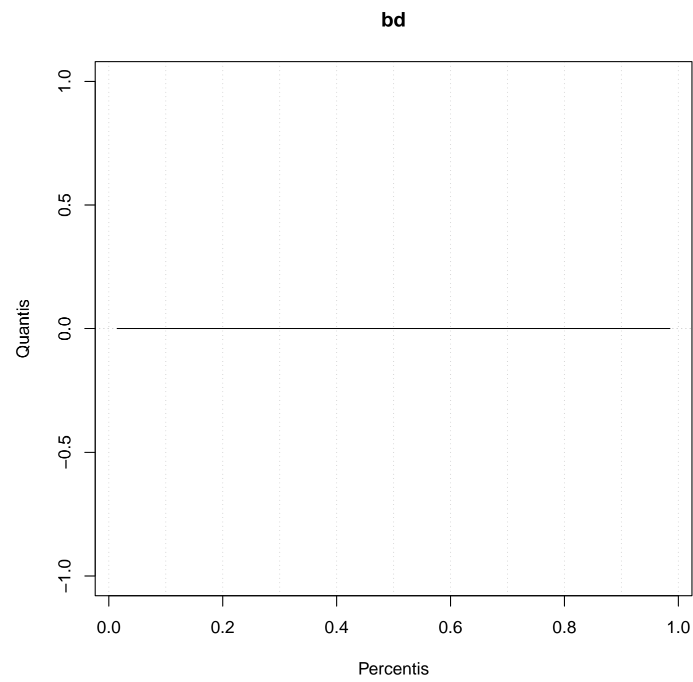


Figura 38 – Gráfico de Percentis da métrica BD

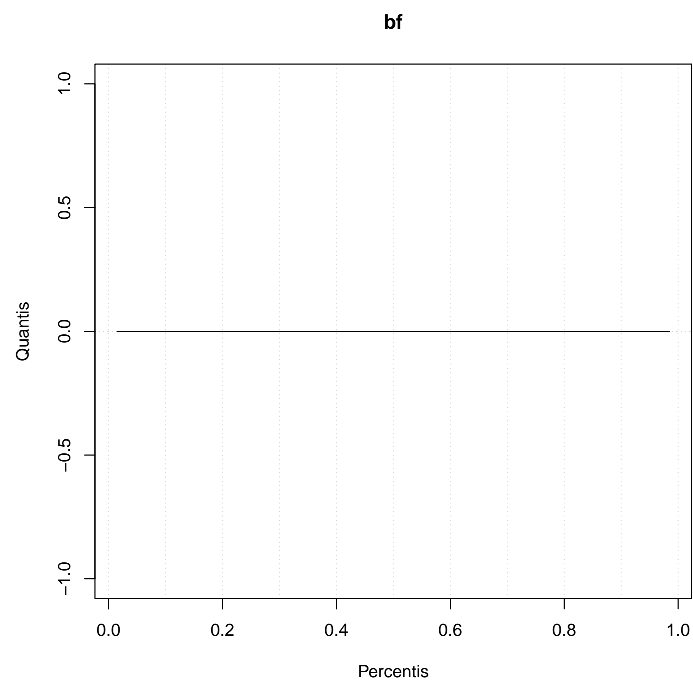


Figura 39 – Gráfico de Percentis da métrica BF

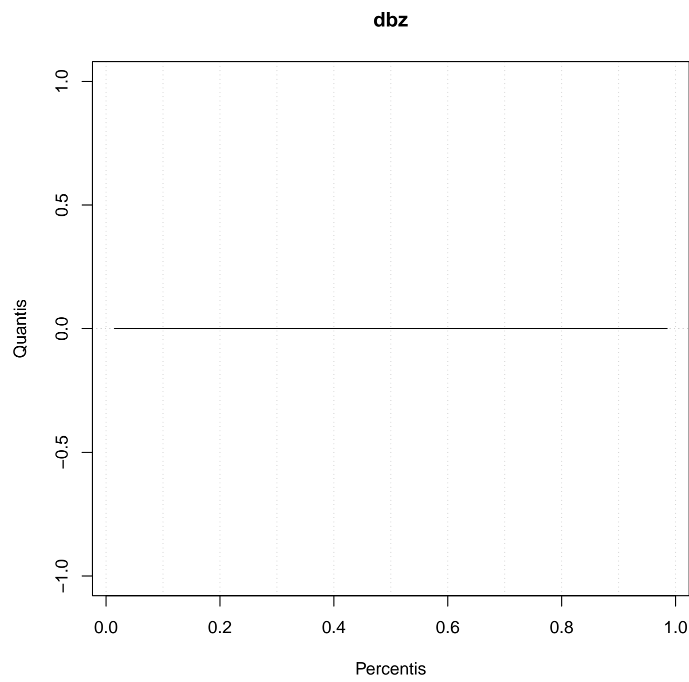


Figura 40 – Gráfico de Percentis da métrica DBZ

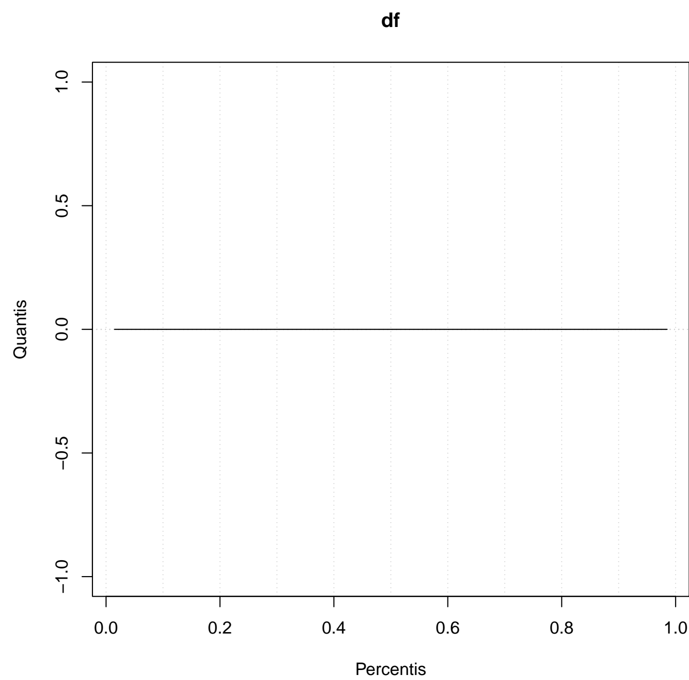


Figura 41 – Gráfico de Percentis da métrica DF

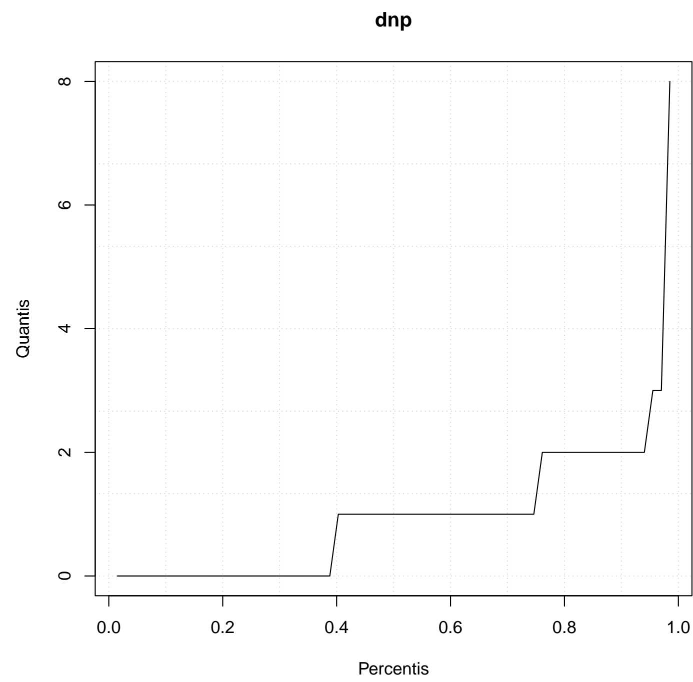


Figura 42 – Gráfico de Percentis da métrica DNP

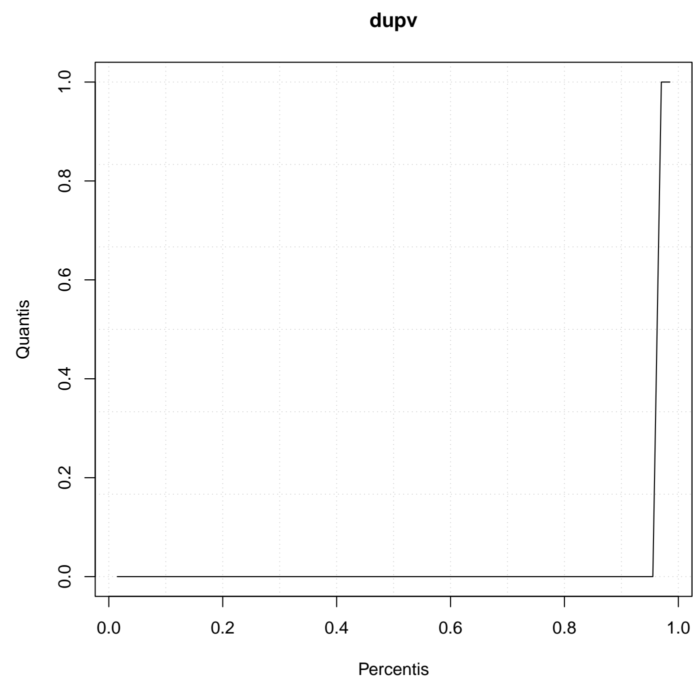


Figura 43 – Gráfico de Percentis da métrica DUPV

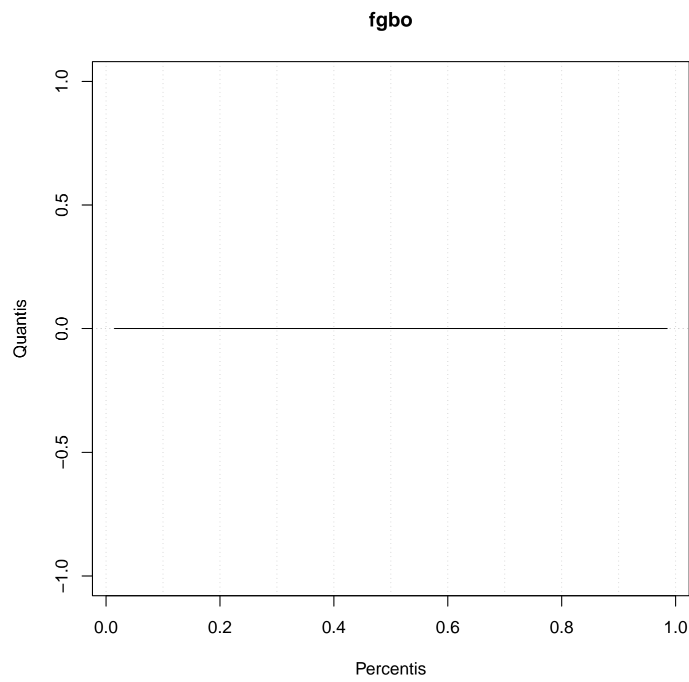


Figura 44 – Gráfico de Percentis da métrica FGBO

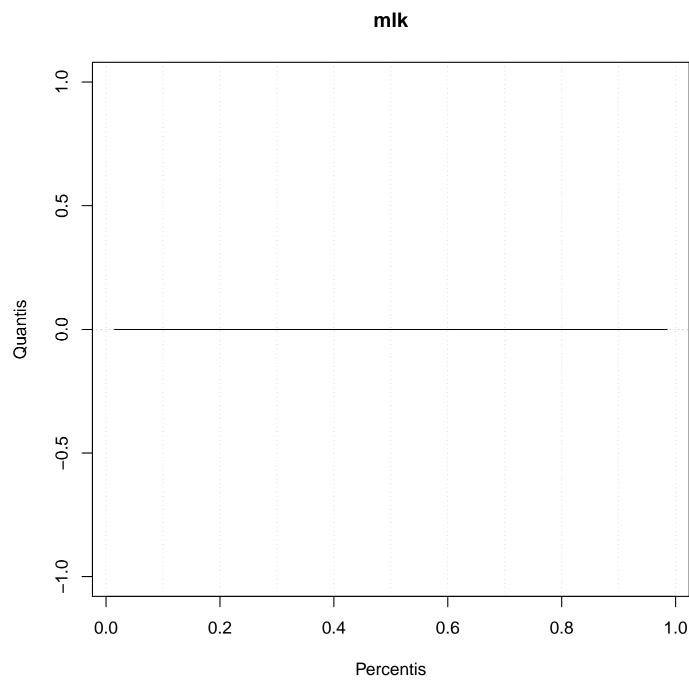


Figura 45 – Gráfico de Percentis da métrica MLK

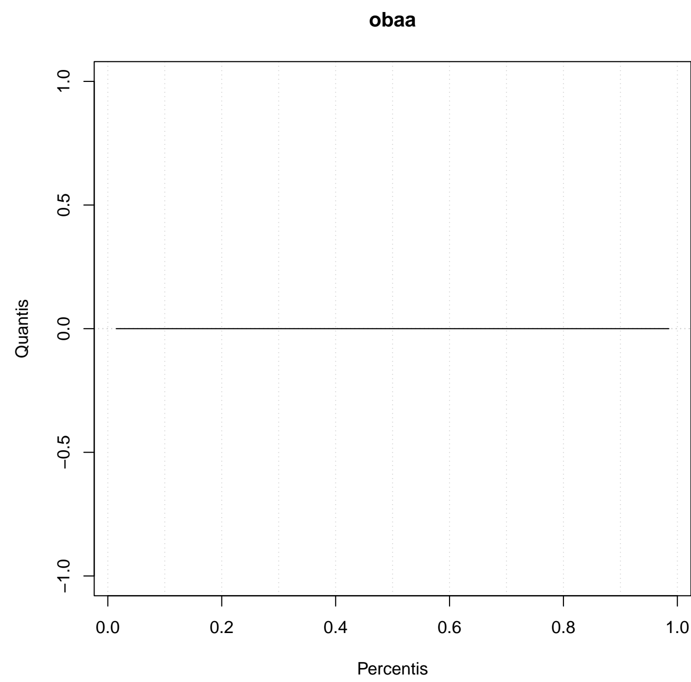


Figura 46 – Gráfico de Percentis da métrica OBAA

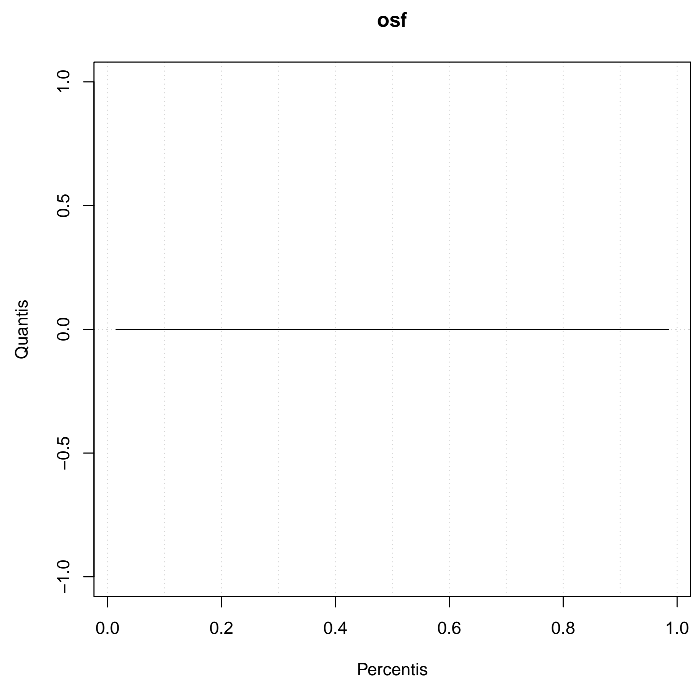


Figura 47 – Gráfico de Percentis da métrica OSF

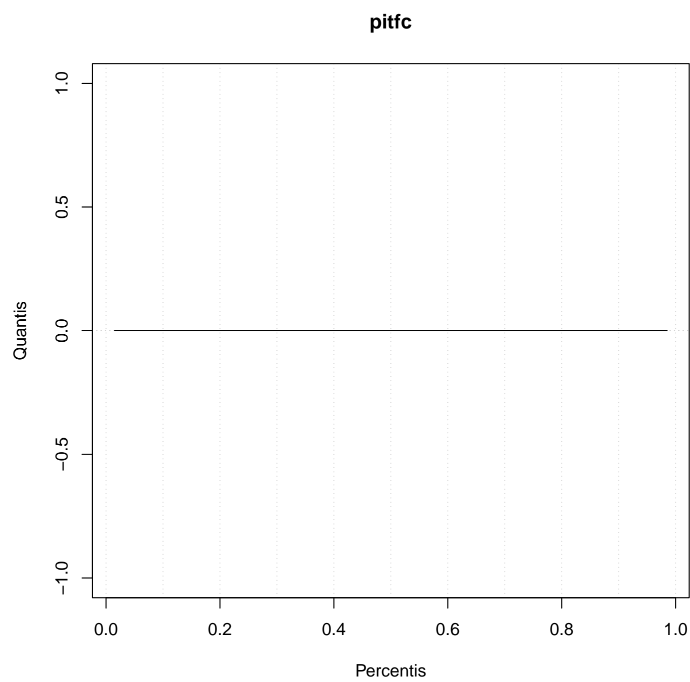


Figura 48 – Gráfico de Percentis da métrica PITFC

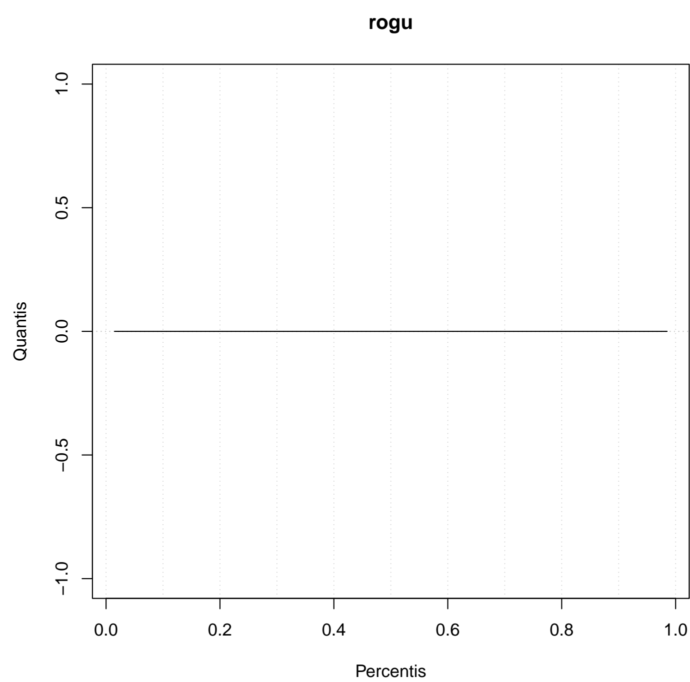


Figura 49 – Gráfico de Percentis da métrica ROGU

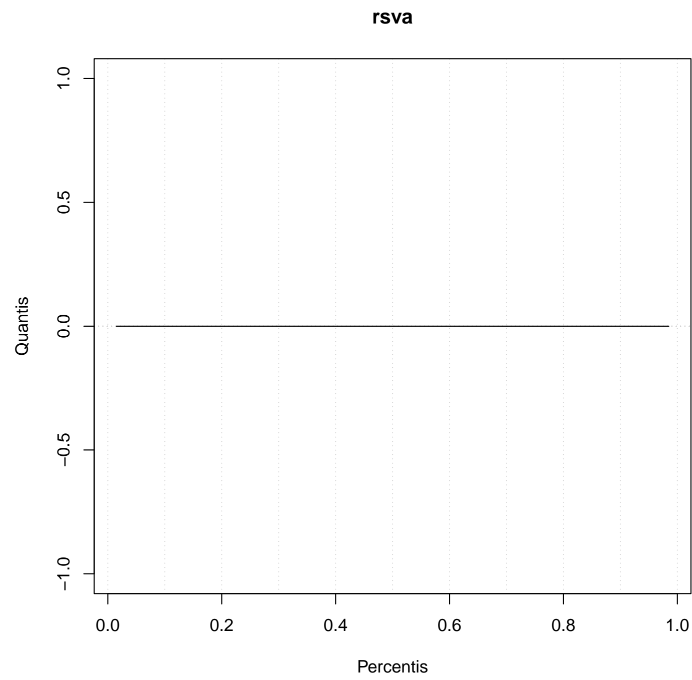


Figura 50 – Gráfico de Percentis da métrica RSVA

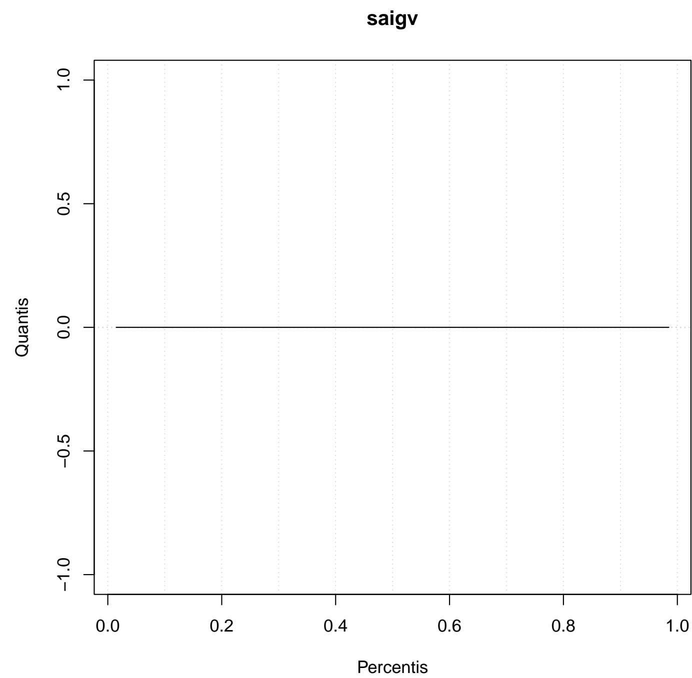


Figura 51 – Gráfico de Percentis da métrica SAIGV

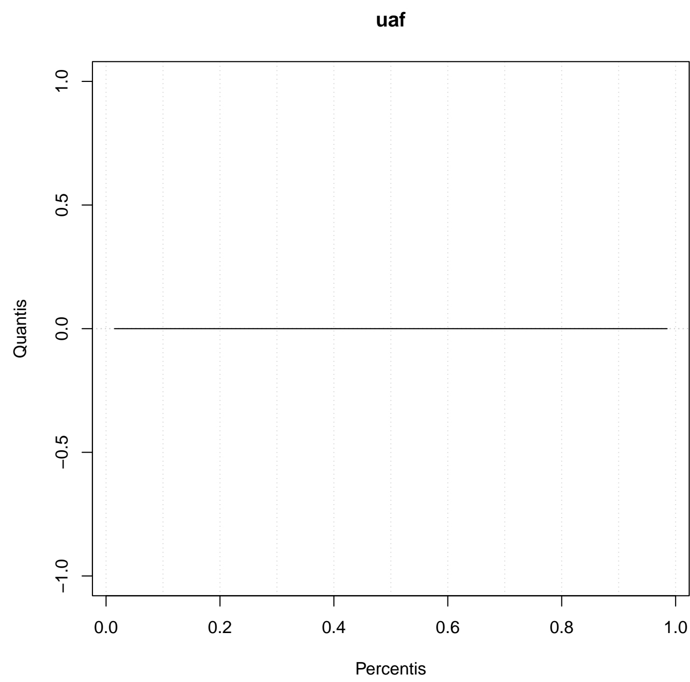


Figura 52 – Gráfico de Percentis da métrica UAF

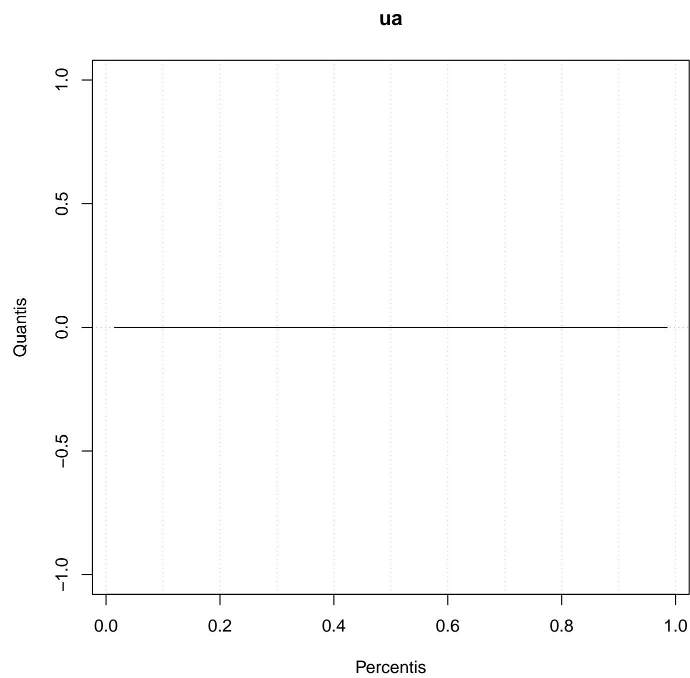


Figura 53 – Gráfico de Percentis da métrica UA

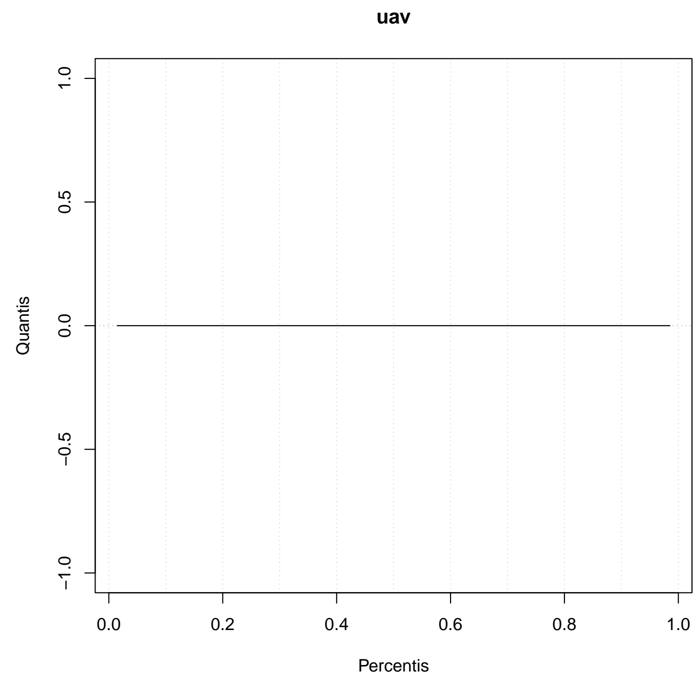


Figura 54 – Gráfico de Percentis da métrica UAV

APÊNDICE B – Análise Qualitativa

Neste apêndice possui as tabelas de análise das métricas de ameaças de vulnerabilidade de código fonte por módulo dos projetos utilizados para realização da análise qualitativa. A partir desta análise foi possível identificar um conjunto de métricas mais recorrentes em alguns projetos de software livre.

Projetos analisados:

- Bash
- Blender
- FFmpeg
- Firefox
- Gstreamer
- Inetutils
- OpenSSH
- OpenSSL
- Python2.7
- Ruby-2.1

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	11	4.3478
AUV	2	0.7905
DNP	22	8.6956
MLK	1	0.3952
ROGU	3	1.1857
UA	1	0.3952
UAV	3	1.1857

Tabela 11 – Projeto Bash

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	7	0.4755
AUV	9	0.6114
DNP	93	6.3179
DUPV	1	0.0679
ROGU	11	0.0563

Tabela 12 – Projeto Blender

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	10	0.5630
AUV	31	1.7454
DNP	37	2.0833
DUPV	4	0.2252
MLK	1	0.0563
ROGU	24	1.3513
UAV	13	1.1857

Tabela 13 – Projeto FFmpeg

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	29	0.2337
ASOM	7	0.0564
AUV	24	0.1934
DNP	218	1.7574
DUPV	10	0.0806
MLK	12	0.0967
OBAA	5	0.0403
ROGU	34	0.2741
SAIGV	2	0.0161
UA	6	0.0483
UAF	8	0.0644
UAV	22	

Tabela 14 – Projeto Firefox

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	2	0.7575
DNP	16	6.0606
ROGU	2	0.7575
UAV	4	1.5151

Tabela 15 – Projeto Gstreamer

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	3	0.1164
AUV	3	0.1164
DF	1	0.0388
DNP	6	0.2329
MLK	9	0.3493
ROGU	7	0.2717
SAIGV	1	0.0388
UA	2	0.0776
UAF	5	0.1940
UAV	3	3.2667

Tabela 16 – Projeto Inetutils

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
DNP	40	0.1734
DUPV	3	0.0130
SAIGV	1	0.0043

Tabela 17 – Projeto Linux Kernel

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	2	0.8064
DNP	7	2.8225
MLK	2	0.8064
UA	1	0.4032
UAF	4	0.1026

Tabela 18 – Projeto OpenSSH

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	6	0.6160
AUV	1	0.1026
DNP	14	1.4373
ROGU	6	0.6160
UAV	1	0.7472

Tabela 19 – Projeto OpenSSL

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	2	0.3629
AUV	4	0.7259
BF	1	0.1814
DNP	18	3.2667
DUPV	2	0.3629
MLK	2	0.3629
OBAA	2	0.3629
ROGU	6	1.0889
UA	1	0.1814
UAV	1	0.1164

Tabela 20 – Projeto Python2.7

Métrica	Nº total de Módulos vulneráveis	% de Módulos vulneráveis
AN	4	1.0498
ASOM	2	0.5249
AUV	2	0.5249
DNP	10	2.6246
MLK	1	0.2624
ROGU	2	0.5249
SAIGV	1	0.2624
UAF	1	0.2624
UAV	1	0.1814

Tabela 21 – Projeto Ruby-2.1